Integrating functional programming into C++.

Abdul Rauf, Rose Hafsah Binti

Integrating Functional Programming Into C++

Rose Hafsah Binti Ab. Rauf BSc. (Malaysia) MSc. (Malaysia)

A thesis submitted to the University of Wales in
candidature for the degree of Philosophiae Doctor

Department of Computer Science
University of Wales, Swansea

July 2008

Summary

C++ is a general purpose language that supports object-oriented programming as well as procedural and generic programming, but unfortunately not functional programming. We have developed a parser-translator program that translates simply typed λ-term to equivalent C++ statements so as to integrate functional programming. The program parses λ-terms and translate them into full language of C++. Our intention is to upgrade this to an extension of the language of C++ by λ-types and -terms together with a parser program which translates this extended language into native C++. For this purpose we introduce a syntax for representing λ-types and -terms in C++. We use functional style notation rather than overloading existing C++ notation, since we believe that this will improve readability and acceptability of our approach among functional programmers.

The translated code generated by the parser-translator program uses the object-oriented approach of programming that involves the creation of classes for the λ-term. By using inheritance, we achieve that the translation of a λ-abstraction is an element of a function type.

The most important advantage of our thesis is that we give a mathematical proof of the correctness of the translation, and to our knowledge the verification of the implementation of λ-calculus in C++ using a logical relation is new. We introduce a suitable fragment of C++ with a precise denotational semantics. We give a formal translation of λ-terms into this fragment and show that it preserves this semantics. We show as well completeness, i.e. essentially all programs in this fragment of C++ can be obtained by translating terms of the λ-calculus. We develop a mathematical model for the evaluation of programs in this model, and show that this evaluation is correct with respect to the denotational semantics.

We hope that our model of a fragment of C++ which includes a formal model of the heap, will have applications which go beyond the translation of the typed λ-calculus. We expect that extensions of this model can be used to verify formally the correctness of more complex C++ programs, including programs with side effects. We believe that if our approach is extended to cover full C++, we obtain a language in which the worlds of functional and object-oriented programming are merged, and that we will see many examples where the combination of both language concepts (such as the use of λ-terms with side effects), will result in interesting new programming techniques.

Declaration

This work has not been previously accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signed .... (candidate)

Date ....10/7/08..........

Statement 1

This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

Signed .. .. (candidate)

Date ....10/7/08..........

Statement 2

I hereby give my consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed .... (candidate)

Date ....10/7/08..........

Acknowledgements

This thesis would not have been possible without the support of many people. I would like to express my heartiest gratitude to both of my supervisors, Dr. Ulrich Berger and Dr. Anton Setzer. They were abundantly helpful and offered invaluable assistance, support and guidance. Without their guidance and knowledge, this thesis will not have been successful.

Special thanks also to my colleagues and friends who gave their support and assistance throughout my duration of study. I would also like to convey my thanks to the Department of Computer Science for their cooperation and facilities, and not forgetting to my sponsor, Department of Training and Scholarship, University of Technology MARA, Malaysia for their financial means.

I wish to express my love and infinite gratitude to my beloved family especially my husband, Abd. Ghafar and my children, Najwa, Nawal, Nadia, Nazira, Najat, Najla and Abd. Hafiz for their support and sacrifices. Also, my love to my parents for their endless support and love.

Finally, I would like to thank every individual that was involved directly and indirectly throughout the process of my thesis.

Contents

Chapter 1

Introduction

Ever since their inception in the 1950's [BG96], high-level programming languages have been a fascinating and productive area of study. Programmers endlessly debate the relative merits of their favourite programming languages, and researchers are looking for ways to design languages that combine expressive power with simplicity and efficiency. Looking at the history of programming languages we can clearly see a divergence of programming paradigms (for example, object-oriented, logic, and functional) despite the fact that a "universal" high-level programming language integrating all these paradigms would be highly desirable. Therefore, great efforts are being made to fight this divergence by creating such an integrated programming language. This thesis is not an exception, in the sense that it integrates functional programming concepts into the C++ language, with the longterm goal of completely merging the functional and object-oriented programming paradigms. Undoubtedly, this will result in a wealth of interesting new programming techniques such as lazy evaluation in C++ (see Chapter 4, section 4.6).

While the integration of functional and object-oriented programming concepts had been successfully attempted before ([SS00], [FA00], [Kis98], [Lau95], [MS00], [Vel95], [JP00], our thesis goes an important step further by giving a mathematical stringent proof of the correctness of the integration, based on mathematical model of a fragment of C++. Our research has produced four articles (jointly written by my supervisors and myself) which were published in the CIE 2006 (informal proceedings and postproceedings) [Ab06], [ABS08] and in the TFP 2006 (informal proceedings and postproceedings)[ABS06a], [ABS06b].

Before explaining more details of the results of our thesis, let us discuss some fundamental aspects of programming languages as well as the historical development of low- and high-level programming languages. According to Herbert G. Mayer [May87], the primary function of programming languages is to let the user communicate with the computer via a common interface, where programming languages, together with their compilers bridge the gap between low level, binary instructions that machines understand, and the higher level in which people express their thoughts. We can say that programming languages are the medium through which users communicate with a computer. Programming languages have a wide spectrum of levels spanning from the low-level machine and assembly languages to the high-level machine independent languages.

A low-level programming language is a machine dependent language. This dependency makes the program written not portable from one machine to another. A low-level programming language requires an additional transformation from the conceptual idea to the actual data structures and instructions. It takes a longer time to write a program in a low-level language than writing it in a high-level language. Although there are setbacks in coding a program in a low-level language, the good thing about it is that program written in a low-level language runs faster than a program in high-level language. This is because high-level language programs need to be translated by means of a compiler into machine language [May87].

A high-level programming language allows the programmer to express complex instruction sequences directly in the language used, and also allows the programmer to ignore the machine-specific details. The more the actual computer can be ignored, the higher is the language level, and the more convenient it becomes to code programs. Although we lose some control over resource utilization, such as data and code space, most of the time high-level language is preferable because memory space and a few seconds of machine time are less precious than a programmer's time.

In the following we will discuss the different levels of programming languages in greater detail and describe their development through the history of programming. We hope that this will give the reader a better understanding and appreciation of the achievements of this thesis.

1.1 A Brief History of Programming Languages

There are many kinds of programming languages on the market, which sometimes make people wonder why this is the case. Is it because of the ever evolving machine that is becoming more and more sophisticated or is it because of the demand of humans that needs everything to be automated? To understand more of why these programming languages spring out rapidly, we reconsider the development of programming languages, starting with low-level languages and moving on to high-level languages.

1.1.1 Development of Low-Level Languages

There are five generations of programming languages ranging from low-level to high-level. The five generations of programming languages start at the lowest level with the first generation which is the machine language. They then range up through the second generation - assembly language, third generation - high-level languages (procedural language), and the fourth generation - very high-level language (problem-oriented language). At the highest level are the fifth generation languages which are the languages close to natural language. Beginning in 1945, the five generations have evolved over the years, as programmers adopted the later generations. The birth of the generations are as follows [WS03]:

- First generation, 1945
- Second generation, early 1950's

- Third generation, mid 1950's
- Fourth generation, early 1970's
- Fifth generation, early 1980's

1.1.1.1 First Generation

The first-generation languages are machine languages. They are primitive languages where the program consists of sequences of instructions called machine code . This machine code is addressed to the hardware of the computer and is written in binary notation which consists of binary digits i.e. 0 and 1. The instructions are made of strings of binary digits which represents operations such as add, subtract and compare. A later improvement of the language is allowing the use of octal, decimal or hexadecimal representation of binary strings. Writing the machine language programs is tedious and error prone. Due to these impracticalities of the language, a second generation language is introduced in the early 1950's.

1.1.1.2 Second Generation

Second generation languages are called assembly or symbolic languages. These languages use mnemonics to represent operations such as ADD for add or SUB for subtract. The assembly language program when compiled is translated to machine language by an assembler. All computers operate using a machine language. If programs are written in other than machine language, they have to be translated to a machine language by a compiler or an interpreter that is specific to that language.

One setback to this low-level language is that it is machine dependent, which means that each one only work on one specific type of computer.

1.1.2 Evolution of High Level Languages

Programs developed in the low-level language is too specific in following the low-level details of computer's hardware and they lack portability between different computers. These disadvantages of low-level languages lead to the development of high-level languages. High-level languages allow programmers to ignore low-level details of computer hardware and the nearer the language resembling the 'natural language' the less likely errors could be made by the programmer.

1.1.2.1 Third Generation

In the mid 1950's, the third generation of languages were in use. They are algorithmic or procedural languages that are used to solve a particular type of problem. There are many different kinds of high-level languages produced due to the different attitude in solving the problems involved [Hig73].

The first high-level language is Fortran (FORmula TRANslation). It was developed in 1956 by John Backus at the IBM Corp., for scientific and engineering applications. The Fortran compiler was not only the first compiler, but was also the best optimizing compiler in years to come. Over the years, Fortan was developed into Fortran-II, Fortran-IV, Fortran-66, and Fortran-77 [May87]. In the early 1950's, John McCarthy at the Masachusetts Institute of Technology developed LISP (LISt Processing) and it was implemented in 1959. LISP handled recursive algorithms better and become the standard language for the artificial intelligence community. It began as a purely functional language but soon acquired some important imperative features that increased its execution efficiency. But, it is still the most used functional languages. But, ML and Haskell have widespread use. More on history of functional programming in Chapter 3. However, LISP is gradually being replaced or challenged by Prolog in the artificial intelligence applications.

COBOL (COmmon Business Oriented Language) is the first language designed for commercial application and it is still widely used now. It was developed in 1959 by a navy programmer Captain Grace Mary Hopper and her committee of computer manufacturers and users. It is used for a certain type of applications such as applications that involved processing of dollars and cents. It is advanced in the use of file processing and handling of character string data.

In Europe at about 1958, ALGOL (ALGOrithmic Language) was developed as an improvement over Fortran. It was redesigned and improved further until it was completed and published in 1960 as ALGOL-60. Even though it was said to be the most ingenious language effort in the early days of programming languages, it never gained widespread acceptance [May87]. It is used primarily in mathematics and science as is APL. APL (A Programming Language) is published in the United States in 1962 by Kenneth Iverson at Harvard University.

In 1966, PL/1 (Programming Language 1) is introduced by IBM Corp. It was intended as a replacement for all previous programming languages and has features from all other programming languages. Another important language is ADA. Its name was taken to honour Ada Augusta, the countess of Lovelace. She was the biographer of Charles Babbage and considered as the first computer programmer, since she wrote programs for Babbage's machine. Ada was developed in 1981 by the U.S. Deptarment Of Defence. It was designed as a language for military applications, in order to have one uniform language in which most software for US military applications should be written in future.

BASIC (Beginner's All-purpose Symbolic Instruction Code) was designed by two professors from Dartmouth College , John Kemeny and Thomas Kurtz in 1966 as an easy to learn interactive programming language. It became the primary language used in microcomputers for a while, but has lost its importance. In 1971, a more structured language for teaching that was named Pascal after Blaise Pascal, a French mathematician, was developed. It was designed by Nicholas Wirth, a Swiss professor. It is one of the few very well designed languages which is widely used. Then in 1982, Wirth introduced Modula-2. It is a Pascal-like language for commercial and mathematical applications. Modula-2 is a general purpose programming language which is also designed for systems programming.

Around 1972, Dennis Ritchie of Bell Laboratories produced a language called C to implement the UNIX operating system. It is a general purpose language that is mainly suited for operating system implementations. Systems written in C are more portable than the ones written in assembly language. C++ is an extension of C is developed by Bjarne Stroustrup of Bell Laboratories. C++ has become the most widely used general purpose language because of its speed and its capabilities to deal with object-oriented programming. Java is an object-oriented language which was developed specifically as a network-oriented language where writing programs can be safely downloaded from the internet and can be executed immediately without fear of any threat from computer viruses.

1.1.2.2 Fourth Generation

Very high-level or problem-oriented languages, also called fourth generation language (4GLs), are much more user-oriented and allow users to develop programs with fewer commands compared with procedural language, although they require more computing power. These languages are called problem-oriented because they are designed to solve specific problems, whereas procedural languages are more general purpose languages.

There are three types of problem-oriented languages. They are report generators, query languages and application generators. A report generator which is also called as report writer is a language to produce a report, where the report can be a printout or a screen display in a certain format specified by the user. A query language is an easy-to-use language for retrieving data from a database management system. The query may be expressed in the form of a sentence or near-English command. An application generator is the programmer's tool consisting of modules that have been programmed to accomplish various tasks. The benefit of this generator is that the programmer can generate application programs from descriptions of the problem rather than by traditional programming, in which the processing of the data have to be specified. Programmers use application generators to help them create parts of other programs such as to construct onscreen menus or types of input and output screen formats.

FORTH is the first fourth generation language developed in 1970 by the American Astronomer Charles Moore. FORTH is used in scientific and industrial control applications. Besides FORTH, NOMAD and FOCUS are database management systems which include application generators. Other examples of application generators are Mathematica, MATLAB, Progress 4GL, Maple SPSS (which are data manipulation, analysis and reporting languages) , APE, AVS (are data-stream languages) and Coldfusion (a web development language). RPGIII, Quest, Report Builder, GEMBase, Oracle Reports, PostScript are examples of report generators, and SQL, Informix-4GL, SB+/SystemBuilder, and Genero are examples of query languages.

High-level, domain-specific programming languages were earlier often mentioned as fourth-generation languages, while expert systems were called fifth-generation programming languages. In later years this distinction has blurred, as many very high-level general purpose programming languages like Python, Haskell and Common Lisp have emerged.

Domain-specific languages are languages tailored to a specific application domain. For a specific domain, they offer substantial gains in expressiveness and ease of use compared to general-purpose languages. They sacrifices generality and provides notations and constructs tailored to a particular application domain. (E)BNF and Excel are representatives of domain-specific language which are for syntax specification and spreadsheet application respectively [HM07]. The term domain-specific language has become popular in recent years in software development to indicate a programming language or specification language dedicated to a particular problem representation technique, and/or a particular palestine technique. The concept isn't new - special-purpose programming language and all kinds of modelling/specification languages have always existed, but the term has become popular due to the rise of domain-specific modelling. The opposite is a general-purpose programming language, such as C or Java, or a general-purpose modelling language such as the UML. Creating a domain-specific language (with software to support it) can be worthwhile if the language allows a particular type of problems or solutions to them to be expressed more clearly than pre-existing languages would allow, and the type of problem in question reappears sufficiently often. In comparison with the domain specific language with our project we can clearly say that the main goal of our project is to develop a general purpose language extension of C++ not a domain specific extensions. Thus we did not consider in integrating the functional programming into C++ as creating a domain specific language.

1.1.2.3 Fifth Generation

Fifth generation language is an outgrowth of artificial intelligence research. Artificial intelligence (AI) is a group of related technologies used for developing machines to emulate human qualities, such as learning, reasoning, communicating, seeing and hearing. In the early 1970s, PROLOG (PROgramming LOGic) was designed by French computer scientist Alain Colmeraur and logician Philippe Roussel. PROLOG is useful for programming logical processes and allows to automatically deduce programs from declarations. Prolog received a major boost in 1981, when the Japanese for New Generation Computing Technology selected logic programming as its enabling software technology, and launched a ten year project to provide complementary hardware technology in the shape of fast logical inference machine [Wat90].

Today, the main areas of artificial intelligence are virtual reality, robotics, natural language processing, fuzzy logic, expert systems, neural networks, genetic algorithms and cyborgs. Virtual reality, a computer generated virtual reality projects a person into a sensation of three dimensional space. Other than using virtual reality in arcade-type games, its more important uses are in simulators for training. Robotics is the development and study of machines that can perform work normally done by people and natural language processing is the study of ways for computers to recognize and understand human language. LUNAR, developed to help analyze moon rocks, answers questions about geology on the basis of an extensive database is an example of natural language processing. Fuzzy logic is a method of dealing with imprecise data and uncertainty, with problems that have many answers rather than one. Unlike classical logic, fuzzy logic is more like human reasoning: it deals with probability and credibility. Expert system is an interactive computer program used in solving problems that would otherwise require assistance of a human expert. Such program simulates reasoning process of experts in certain well-defined areas and incorporates not only the expert's surface knowledge ("textbook knowledge") but also deep knowledge ("tricks of the trade"). Artificial intelligence and fuzzy logic principles are being applied to the development of neural networks. Neural networks use physical electronic devices or software to mimic the neurological structure of the human brain where they learn from example and don't require detailed instructions. A genetic algorithms is a program that uses Darwinian principles of random mutation to improve itself. As in Darwin's rules of evolution, many chunks of code compete to see which can best fulfil the goal of the program where some chunks will become extinct and the survived ones will combine with other survivors to produce offspring programs.

Artificial intelligence research has led to many advances of programming languages including LISP and its dialects , Planner, Actor, the Scientific Community Metaphor, production systems and rule-based languages. According to Hewitt [Hew06], Planner was the first language to feature procedural plans that were called by pattern-directed innovation using goals and assertions. A subset called Micro Planner [Bau72] was implemented by Gerry Sussman, Eugene Charnak, and Terry Winograd [Lig73] and was used in Winograd's natural language understanding program SHRDLU and other projects. Several researches then introduced other subsets of Planner such as PICO-PLANNER [And72] and Popler [Dav73]. Bob Kowalski [Kow88], who had been one of the principal members of the logic paradigm community, then adapted, in collaboration with Alain Colmerauer, some theorem proving ideas into a form similar to a subset of Micro Planner called Prolog. Using Prolog, Kowalski hoped to save the logic paradigm as a suitable approach to artificial intelligence.

There may yet be a spring of a new discipline of programming that can be considered as the sixth generation programming language. Trygve Reenskaug, a researcher at the University of Oslo, created and explored a possible new discipline of programming in his BabyUML project [Ree07] which is still experimental. He regard BabyUML [Ree04] as a sixth generation programming language because it combines the algorithmic capabilities of the third generation with the semantic modelling of the fourth generation language. BabyUML replaces the idea of a closed application with an open module that is created within a running context. Current programming technology involves a four stage process which includes modelling, coding, loading and execution. But, BabyUML merges them into one, making programming a question of dynamically modifying a running system.

1.2 Languages Evolved from other Programming Languages

There are several programming languages that evolved from other programming languages to improve the language in fulfilling the demands of system development where software are becoming more and more complex. OCaml (Object Caml) is the implementation of the Caml dialect and of ML extended with class based object and powerful module system in the style of SML. It is a general purpose programming language which combines functional, imperative and object oriented programming. It is suited to medium advanced programmers as a tool to boost their productivity through type inference. OCaml does something similar to what we aim at, but coming from the functional programming side. It is the extension of ML by objects. It lacks the full power of C++ concepts, especially pointers, a rich object-oriented structure, explicit memory management. However it is a very clean language. It is like a functional programming language, with objects added to it, whereas the language, this project was aiming at (we haven't achieved it in full yet, but some steps towards it) we have in mind is an object-oriented (or in fact multi-paradigm language) with features from functional programming added to it. So OCaml is intended for functional programmers who need some object orientation, whereas the program this project was aiming at is intended for imperative or object-oriented programmers, who need some concepts from functional programming.

F# is the implementation of the core of the Caml programming language for the .Net framework. Its aim is to work together with C#, Visual Basic SML.Net and other .Net programming languages. C# is derived from C and C++ and developed by Microsoft. It is a Java like language for web programming and was specially designed to operate within the .Net framework. Pizza is an extension of Java with important features like parametric polymorphism, function pointers and algebraic types. However, its encoding of λ-terms is extensive. But the generic part of Pizza has been developed further to an extension of Java called Generic Java (GJ). Most ideas of GJ have been incorporated into Java 1.5.

Purely imperative programming languages such as C or Pascal do not provide a satisfying mechanism such as abstraction and data manipulation. C++ is an extended version of C where it supports object-oriented programming and templates (see Chapter 2). Purely object oriented languages like SmallTalk are excellent with dynamic application but do not provide static guarantees. Typed class based programming languages such as C# and Java contain a very large number of constructs and it is sometimes difficult for programmers to choose how to model their program and sometimes one obtains a large program for a simple problem.

1.3 Outline of Thesis

Since the beginning of evolution in software development, programmers or more precisely computer scientists are trying to find ways or techniques in improving how programs are designed or structured. There are several approaches in designing programs. They are known as programming paradigms. The most prominent ones are imperative, procedural, module-based, generic, declarative, functional and object-oriented programming. These programming paradigms are discussed in Chapters 2 and 3.

By combining the advantages of functional programming and object-oriented programming, it is hoped that a general purpose object-oriented language like C++ can enhance the efficiency of developing a program. Since functional programming is based on the λ-calculus, it is appropriate to embed the typed λ-calculus into C++. This extension of C++ is developed by creating a parser that can parse a C++ program and translate any typed λ-terms in it to equivalent C++ statements. This integration of functional programing into C++ is to simplify the coding of the typed λ-terms so as making it a simple task to define λ-terms in C++. The syntax of defining these typed λ-terms was decided based upon simplification and ease of use for programmers or users.

A discussion on the approach that we use in integrating functional programming into C++ and the design, specification and development of the program that parses and translates λ-terms into equivalent C++ code can be seen in Chapter 4. The implementation of the parser-translator program is discussed in greater detail in Chapter 5. In this chapter, the parsing and translation of the simply typed λ-terms are discussed. The simply typed λ-terms are translated by using the object-oriented approach of programming that involves the creation of classes for the λ-term. The translation of a λ-abstraction is an element of a function type, where the concept of inheritance plays the main role. The execution of the translated code in C++ is discussed by showing how the classes and variables are allocated on the heap in the memory. The evaluation strategy of the translated code is call-by-value.

One thing that is new in our approach is that we have correctness proof of our C++ implementation of the λ-calculus. We proof the correctness of our implementation with respect to the usual (set-theoretic) denotational semantics of the simply typed λ-calculus and a mathematical model of sufficiently large fragment of C++ using the Kripke-style logical relation. Complete proofs are given in Chapter 6. Related work in integrating functional programming into C++ is discussed in Chapter 7. Summary of the thesis is discussed in Chapter 8 and future work is recommended. It becomes our believe that if our approach is extended to cover full C++, we can obtain a language in which the worlds of functional and object-oriented programming are merged.

As mentioned earlier, we have produced papers from our research which are both refereed at usual journal standards and are quite different from the thesis. Papers in the Theory Of Computing System (Appendix C) and Trends in Functional Programming 2006 (Appendix D) use monadic concepts to define the model, and the latter paper (TFP) added the lazy data structures.

Chapter 2

From Imperative Programming to Object-oriented and Generic Programming

A programming paradigm is defined as a paradigmatic style of programming. This can be compared with the notion of programming methodology, which is a paradigmatic style of carrying out software engineering. A programming paradigm provides a view of how the program is being represented. It determines the style and the design method the programmer would use in developing software.

Programming languages are tools for writing software. They are the tools we use to communicate not only with computers but with people. They have been an active field of computer science throughout the decades. As discussed earlier, there are many programming languages , beginning with the lowest to the higher hierarchy of programming languages (refer to Chapter 1, section 1.1). Computer programmers or researchers/computer scientists are still trying to find a better programming language that can be used with ease in writing software efficiently. The pros of different languages are sometimes combined to create a new language or an extension of an existing language.

Just as different groups of software engineering, support different methodologies, different programming languages support different programing paradigms. There need not be a one-to-one relationship between programming languages and their paradigms. Some languages are designed to support one particular paradigm. Such languages are called paradigm-oriented, for example Java and Smalltalk support object-oriented programming while Haskell and Scheme support functional programming. Other programming languages support multiple paradigms and are therefore paradigm-neutral like C++, which is designed to support elements of procedural programming, object-oriented programming and generic programming. The design abstractions can easily be directed to program components if the design method and the language paradigm are the same or the language is paradigm-neutral [GJ98].

Many programming paradigms are well known for what techniques they forbid or enable. For example, pure functional programming disallows the use of side effects and structured programming disallows the use of goto's. Object-oriented paradigm is the most common style of programming nowadays. It is certainly the key programming methodology for the next decade [DD01].

Before going further into the object-oriented paradigm of programming, we think that it is important first to go through some of the paradigms in programming that is relevant to this research. We discuss imperative programming first because it is the basis of most programming not including functional programming. The project in this thesis applied object-oriented programming in developing the program and in the translation of the λ-expression. Structured programming is also discussed because the objects in the object oriented programming have internal structures which is usually built using structured programming techniques and also the manipulation of the objects is best expressed with this technique. The concept of generic programming make possible the existence of Standard Template Library(STL) [STL00]. Especially containers make use of this concept. We will use generic programming when creating the translation of the function type of a λ-term. (see Chapters 4, 5, 6)

2.1 Imperative Programming

The imperative programming paradigm is an abstraction of the principles for executing programs in real computers which in turn are based on the Turing machine and the von Neuman machine. A diagram of the von Neuman machine is given in Figure 2.1 [GJ98]. This architecture consists of a memory, that contains data and instructions, a CPU and an I/O unit.

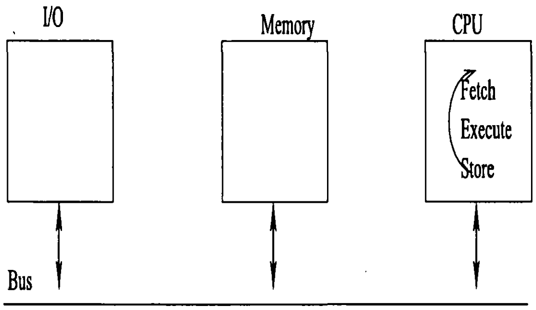

Figure 2.1: A von Neumann computer architecture

The CPU is responsible for fetching instructions one at a time. Since machine instructions are very low-level , they require the data to be taken out of memory and manipulated through arithmetic and logic operations with the result being copied back to the memory. Execution of instructions result in the change of the state of the machine which is reflected by the contents of the memory.

An abstraction is a model that highlights the relevant aspects of a phenomenon and ignores its irrelevant details [GJ98]. In other words, conventional programming languages adopt the underlying von Neumann architecture as their computational model but abstract away from the details of each sequential step of execution. This model consists of a sequential, step by step execution of instructions which change the state of computation by modifying the repository of values. Sequential execution of language reflects the sequential fetch and execution of machine instruction performed by the hardware. A variable of the language which can be modified by the assignment statements, reflects the behaviour of the memory cells of the computer architecture. Higher levels of abstractions such as procedures and functions, data types, exception handlers and classes have been developed from time to time until now by language designers to overcome the ever increasing needs of programmers. Even though higher level languages have been designed to make programming much easier, the concept of the languages are still based on the von Neumann architecture.

The state in an imperative language is the logical model of storage which is an association between memory locations and values. It consists of collection of names and the associated values and the location of control in the program. In imperative programming, a name may be assigned to a value which in turn can be reassigned to another value. The execution of a program generates a sequence of states abbreviated as S. The transition from one state to the next is determined by assignment operations and sequencing commands that is abbreviated as O in the expression below:

$$S_0 \xrightarrow{O_0} S_1..... \longrightarrow S_{n-1} \xrightarrow{O_{n-1}} S_n$$

Imperative programs are characterized by sequences of bindings i.e. state changes. So, a name has two bindings which is a binding to a location and to a value. The location is called the *l-value* and the value is called the *r-value*. For example, the statement :

```
    y := y + 1
```

indicates that the y on the left(*l-value*) denotes the location while the y on the right(*r-value*) denotes the value. Assignment changes the value at a location. A variable and value are bound by an assignment. The assignment statement typically has the form :

```
   V := E
```

Varieties of notations are used in a programming language to indicate the binding of a variable V and the value of an expression E. Examples are shown as follows :

```
Pascal   V := E
C++      V = E
APL      V <- E
Scheme  (setq V E)
```

The assignment is not the same as a constant definition because it permits redefinition. For example,

```
y := 2;
y := y + 1;
```

reads as: assign y to 2 and then reassign y to the value of the expression y + 1 which is 3.

Several kinds of assignments are possible. A multiple assignments

```
V0 := V1 := ... := Vn := E
```

causes several names/variables to be assigned to the same value. A simultaneous assignment of the form:

```
V0, V1, ..., Vn := E0, E1, ..., En
```

causes several assignments of names to values to occur simultaneously. This allows the swapping of values without explicit use of an auxiliary variable.

For the point of view of denotational semantics, the assignment is a function from states to states and for the point of view of operational semantics, the assignment changes the state of an abstract machine.

When imperative programming is combined with subprograms, it is called **procedural programming**. An imperative programming can only be understood in terms of its execution behaviour. This is because during the execution of the code, any variable maybe referenced, control may be transfered to any arbitrary point and any variable binding may changed. Hence, the whole program need to be examined in order to understand even a small portion of the program. In view of this, sequence control are very important in an imperative programming. Considerable efforts have been given to find an appropriate control structures. Figure 2.2 gives a minimal set of basic control structures.

```
command   ::=  identifier := expression
             | command; command
             | label : command
             | GOTO label
             | IF boolean_expression THEN GOTO label
```

Figure 2.2: A set of unstructured commands

The unstructured commands include the assignment command, sequential composition of commands, a provision to identify a command with a label, and an unconditional and conditional GOTO commands. The programs are flat without hierarchical structure thus making the code difficult to read and understand. The set of unstructured commands contain one of the most powerful and highly criticized command GOTO, when used in abundance in a program will result in a 'spaghetti' like code which is difficult to understand and read. Due to this, structured programming (known as programming without GOTO) comes into picture where structured programming provides control structures that make it easier to reason about imperative programs.

2.1.1 Structured Programming

Structured programming is a term that describe a style of programming that emphasizes hierarchical program structures in which each has one entry point and a few clearly marked exit points. Its goal is to produce a program that is easy to read and understand hence easy to maintain. A minimal set of structured commands are as in Figure 2.3

```
command ::=  SKIP
           | identifier := expression
           | IF guarded_command [ [ ] guarded_command ]* FI
  | DO guarded_command [ [ ] guarded_command ]* DO
           | command ; command
guarded_command ::=  guard --> command
guard           ::=  boolean expression
```

Figure 2.3: A set of structured commands

At a low-level , structured programs are composed of simple, hierarchical program flow structures. These structure can be regarded as single statements or combination of simpler statements that can be of primitive statements such as the assignment statement or procedure calls. Djikstra identified three types of structures i.e concatenation, selection and repetition. Concatenation refers to a sequence of statements executed in order whereas selection is alternatives or choices given in order to execute an operation which is usually expressed with keywords such as if .. then [else} .. endif, switch or case. Repetition is execution of a statement depending on the state of the program where the statement can be executed 0 or several times depending on the condition given.

The general structure of selection and repetition is shown in the Figure 2.3 as IF..FI and DO..OD respectively. The IF and DO commands defined in the Figure 2.3 are in terms of guarded commands.

IF guard → command FI is equivalent to if condition then command and

DO guard → command is equivalent to while condition do command.

A command preceded by a guard can only be executed if the guard is true. Generally, the semantics of IF - FI and DO - OD commands require that only one command corresponding to the guard is true be selected for execution. The DO command can be represented with keywords such as while, repeat or for.

At a high level structure, programmers should break larger piece of code into shorter subroutine (functions, procedures, blocks or others) that are small enough to be understood and maintained easily. In general, global variables should be used sparingly and local variables should be used instead by subroutines where the arguments can be passed by value or reference. This is to make subroutines or small pieces of code easier to understand without having to go through the whole program.

Structured program is usually designed using the ”top down” approach where large scale structure of a program are mapped out into smaller operations. This smaller operation are implemented and tested and then tied together to form a whole program.

Imperative programming languages have a rich assortment of control structures, which represent Djikstra’s control structures.

2.1.2 Sequential Composition

Sequential composition specifies a linear ordering of command execution. Usually it is indicated by placing textual sequence separated by a line or a symbol (most commonly a semicolon). This symbol usually used as a termination point for the commands or a command separator(for example in C++). At an abstract level, composition of commands is indicated by using composition operator such as semicolon ($C_0;C_1$).

2.1.3 Selection or Alternation

Selection permits the specification of a sequence of commands by cases. The selection of a particular sequence is based on the value of an expression. The most common representative of alternation are the commands `If` and `Case`. For `If` command the condition is a boolean expression, while `Case` command permits any scalar expression. The `Case` statement is best used when the selection is from many statements.

2.1.4 Iteration

Iteration specifies that a sequence of commands may be executed zero or more times (repeatedly). Most programming languages provide different loop constructs. This loop constructs define an iteration of certain action which is called the loop body. It also has an expression which determines when the execution will ceased. Often, one distinguishes between loop based on whether the number of repetitions are known at the start of the loop or the repetitions continue until a certain condition is met. The former kind of loop is usually called a 'for' loop and the latter is often called the 'while' loop.

The 'for' loop define the control variable which takes on all values of a given predefined sequence. For every value the loop body is executed. The general appearance of a for loop is shown as follows :

```
for loop_ctr_var := lower_bound to upper_bound do statement
```

The 'while' loop describe any number of iterations of the loop body, including zero. The semantics of this loop require the testing of the condition or expression before the body is executed. They have the following general form :

```
while condition do statement
```

Some languages provide a similar kind of loop as 'while', where the condition is checked at the end of the body (i.e. the loop body is executed at least once). In Pascal, the construct has the following general form:

```
repeat statement until condition
```

In the 'repeat' loop, the body is executed as long as the condition is false. It will terminate when the condition becomes true. C++ provides the 'do-while' statement that behaves in a similar way which has the following general form :

```
do statement while expression
```

The body of the 'do-while' statement is executed repeatedly until the value of the expression becomes zero (i.e. the condition is false).

2.1.5 Side Effects

Side effects are a feature of imperative programming languages that make the reasoning of the program difficult. Side effects are used to provide communication among program units, but when undisciplined access to global variables are permitted, the program becomes difficult to understand. The whole program needs to be scanned to determine which program unit that access and modify the global variables since a call command doesn't really reveal which variables are affected by the call. The change to a global variable is called **side effect**.

For example:

```
integer f(a:integer)
{
   b := b +1
   f := b + a
}
```

This function computes a value as well as changing the global variable b. This causes side effects. In addition of it changing the global variable, the function is difficult to reason with itself. For example, if at some point in the program it is known that b = y = 0, then the call f(y) will return a value 1. But, should the following expression:

```
1 + f(y) = f(y) + f(y)
```

occurs at that point in the program, then the expression will be false.

2.1.6 Aliasing

Aliasing is another feature that makes programs harder to understand and difficult to reason about. Two names are aliases if they denote the same data object during a unit activation. One way aliases occurs is when two or more arguments to a subprogram are the same. When a data object is passed by reference, it is referenced both by its name in the calling environment and its parameter's name in the called environment. In the following subprogram, the parameters are in-out parameters (which are parameters that acts as inputs and outputs for the subprogram):

```
Aliasing(x, y : in out integer)
{
   y := 1
   y := x + y
}
```

For the call Aliasing(i, i), the two parameters are used as different names for the same object giving i the value 2. But, in the call Aliasing(a[i], a[j]), the result will depend on the values of i and j with aliasing occurring when they are equal. This later call illustrates that aliasing can occur at run time, so the detection of aliasing may be delayed until run time, thus compilers cannot be relied on to detect aliasing.

Aliasing interferes with the optimizing phase of a compiler. Optimization sometimes requires the reordering of steps or the deletion of unnecessary steps. The following assignments which appear to be independent of each other illustrate an order of dependency.

```
x := a + b
y := c + d
```

If x and c are aliases for the same object, the assignments are interdependent and the order of evaluation is very important.

Other ways that aliasing can occur:

- A data object may be a component of several data objects (referenced through pointer linkages)
- Formal and actual parameters share the same data object
- Procedure calls have overlapping actual parameters
- A formal parameter and a global variable denote the same data object

Pointers are intrinsically generators of aliasing. When a programming language requires programmers to manage memory for dynamically allocated objects and the language permits aliasing, an object returned to memory may still be accessible through an alias and the value may be changed if the memory manager allocates the same storage area to another object. For example, in the following code, the pointer r is left pointing to a non-existent value.

```
type pointer = *Integer
var r : pointer;

procedure FreePointer:
var s : Pointer;
begin;
new(s);
s* := 10;
r  := s;
dispose(s)
end;

begin
new(r);
FreePointer(r)
```

Many times optimizers have to make conservative assumptions about variables in the presence of pointers. For example, a constant propagation process which knows the value of y is 1 will not be able to keep this information after an assignment e.g. `*x = 2` because maybe that`*x` is an alias of y (in the case after an assignment such as `x = &y`). The value of y will be changed as well after the effect of the assignment to `*x`. Thus, propagating the information that y is 1 to the statements following `*x = 2` would be wrong if `*x` is indeed an alias of y. However, if we have information about pointers, the constant propagation process could make a query like: Is y an alias of *x?. Then if the answer is no, then y = 1 can be propagated safely.

Another optimization that is as an effect of aliasing is code reordering. If the compiler decides that y is not an alias of *x, then the code that uses and changes the value of y can be moved before the assignment *x = 2, if this improves scheduling or enable more loop optimizations to be carried out. In order to enable such optimizations to be carried out in a predictable manner, the ISO standard for the C language specifies that it is illegal (with some exceptions) for pointers of different types to reference the same memory location. This rule is known as strict aliasing. It allows impressive increases in performance but has been known to break some valid code.

The problem of aliasing arises as soon as language supports variables and assignments. If more than one assignment is allowed on the same variable, the fact that x = y cannot be used at any other point in the program to infer a property of x from a property of y. The use of aliasing and global variables magnifies the issue more.

Imperative constructs jeopardize many of the fundamental techniques for reasoning about mathematical concepts. For example, the assignments axiom of axiomatic semantics is valid only for languages without aliasing and side effects. Much work has been tempted to explain the "referential opaque"' features of programming languages in terms of well defined mathematical constructs. By providing descriptions of programming language features in terms of standard mathematical concepts, programming language theory makes it possible to manipulate programs and reason with them using rigorous and precise techniques. But the resulting descriptions are complex and the notational machinery is difficult to use. One strong motivation for functional and logic programming is that it avoids this complexity of imperative programming.

2.2 Procedural Programming

In the history of computer programming, most programs were written sequentially where programs consist of series of steps that take place one after the other where these steps are executed based on the condition determined by the programmer. The major setback in the sequentially written program that does not involve any procedures, is that some part of the program had to be rewritten in more that one place if the same task has to be done in a different part of the program. This involves duplication of statements. To overcome this, programming languages allow methods to be used making writing programs becomes easier because statements that are used frequently in the program such as the task of printing are grouped together in a method. This method can be called whenever needed. Method in programming languages are known as functions, procedures, methods, subprograms, subroutines or simply routines.

For example the task of printing a message. In C++, this task can be done by the statement :

```
cout<<message;
```

where this message can be printed out depending on the content of the variable message. This statement can be a part of a function `prtmessage` shown below :

```
void prtmessage(string message){
   cout<<"The message is "<<message<<endl;
}
```

So in the main program, the use of this function `prtmessage` can be seen as follows :

```
main(){

string messg;
messg = "good evening";
prtmessage(messg);
prtmessage("goodbye");
}
```

The output of the above segment program is

```
The message is good evening
The message is goodbye
```

As mentioned the use of functions in a program contribute in structuring a program provided the coding of the function follows the structuring techniques.

Procedural programming is a conventional programming style that is based upon the concept of modularity and scope of program code. Programs are decomposed into computation steps that perform complex operations. Routines are used as modularization units to define the computation steps. These modules are either coded by the same programmer or precoded by someone else and provided in the code library.

Each module consists of one or more subprograms whereby these subprograms can be composed of procedures, functions, subroutines or methods depending on the programming language used. Most languages distinguish between two kinds of routines i.e procedures and functions. A procedure is an abstract command that is called to alter some desired state and it does not return a value, while functions are the mathematical counterparts which will return a value when activated depending on the arguments or parameters passed.

An example of a function (AVERAGE) that averages three numbers and a procedure (CALCULATE) that calculates the total of three numbers and squaring it written in Fortran are as follows:

```
REAL  FUNCTION AVERAGE(X, Y, Z)
REAL  X, Y, Z
AVERAGE = (X + Y + Z) / 3.0
RETURN
END

SUBROUTINE CALCULATE(A, B, C, TOTAL, TOTSQUARE)
REAL  A, B, C, SUM, TOTSQUARE
TOTAL = A + B + C
```

```
TOTSQUARE = TOTAL ** 2
RETURN
END
```

The function and subroutine defined above can be invoked as shown below:

```
REAL  A, B, C, TOTAL, TOTSQUARE, AVG
CALL CALCULATE(A, B, C, TOTAL, TOTSQUARE)
AVG = AVERAGE(A, B, C)
```

We can say that the function and procedure provides a service or they can be called a service provider and the one that uses them is a client. If the service is provided as a function, then the client has to use it in an expression. On the other hand, if the service is provided as a procedure, the client are forced to use an imperative style. It is also possible for a procedural program to have multiple levels or scope, with subprogram defined inside other subprograms. Each scope can contain name that cannot be seen in outer scopes.

Procedural programming offers more benefit over a simple sequential programming because procedural code is easier to read hence more maintainable, it is more flexible and facilitates the practice of good program design. The canonical example of a procedural programming language is ALGOL. Others are Fortran, PL/1, Modula-2 and Ada.

2.3 Generic Programming

Generics in computer science is defined as a construct that allows one value to take different data types as long as certain contracts such as subtypes and signature are kept. Generic programming is a programming style that emphasizes the use of this technique. Generic modules may be instantiated either during compile-time or run-time to create the entities such as data structures, functions and procedures that is needed to build a program. This programming approach encourages the development of high-level of generic abstractions as units of modularity.

A simple example of using generic technique in creating a list is by declaration the list as `List<T>`, where `T` represents the type of the list. When instantiated, one can create `List<Integer>` or `List<String>`. The list is then treated as whichever type specified.

Polymorphism is the fundamental mechanism for generic programming. Generic programming is best suited to parametric polymorphism where the example on list given earlier is an example of parametric polymorphism. More about polymorphism will be discussed in the section 2.4.2.

The generic programming paradigm does not exist in isolation. It exists jointly with other programming paradigm. For example it exists with object oriented paradigm as in Eiffel and later versions of Java, with functional programming as in ML and also with languages which provide more than one paradigm such as C++ and Ada.

However, it was , templates of C++ that popularized the concept of generics.

2.3.1 Templates

As mentioned above, the concept of generics is popularized by the templates of C++. Templates allow code to be written without concerning much of the data type that eventually will be used in the program. Template in C++ is of great utility to programmers especially when it is combined with multiple inheritance and operator overloading. The C++ Standard Template Library provides many useful functions within the framework of connected templates. For example , the C++ STL contains the function template `max(x,y)` which will return x or y whichever is larger. This template could be defined as :

```
template <class T>
T max(T x, T y)
{ if(y > x)
      return y;
else
      return x;
};
```

It can be called just like a function such as :

```
            cout<<max(24,80);          //outputs 80
```

The call to `max(24,80)` makes the compiler examine the arguments to determine that this call is a call to `max(int,int)` and instantiate a version of the function where the type `T` is `int`. The function `max()` works for all types of arguments as long as the type is applicable to the condition `y > x`. In the example function template `max` accepts two arguments of the same type but one can use a user defined data type. If a user defined data type is used, one can use the operator overloading to define the meaning of `'>'` so as the `max( )` function can be used. Even though the use of operator overloading seems to be a minor benefit for this example, but in the context of a comprehensive library like STL, it allows the programmer to get extensive functionality for a new data type just by defining a few operators for it.

A class template extends the same concept to classes. Class templates are often used to create generic containers such as vectors, lists, deques, stacks and queues, sets and many more. These containers have a set of standard functions associated with it, which works well with whatever matter that you put in between the brackets. For example in C++, has a container class List which contains functions such as `add()`, `detach()` and `getIterms()`.

Previously, some uses of templates like `max( )` function were filled by the function-like preprocessor macros. Macros and templates are expanded during compile-time where macros are always expanded inline while templates can also be expanded as inline function when the compiler deems it appropriate. Therefore, both function templates and function-like macros have no runtime overhead.

However templates are considered far more better than macros because of the following reasons :

- Templates are type safe

- Templates avoid some of the errors that occur for the code that uses many function-like macros
- Templates were designed to be applicable to much larger problems than macros.

But, templates also have their disadvantages. There are three drawbacks to the use of templates which are:

- Historically, many compilers have very poor support for templates making the code using them less portable. However, most modern compilers now have fairly robust and standard template support and the new C++ standard , C++0x, is expected to further address the issue of portability.
- Almost all compilers produce confusing, unhelpful error messages when errors are detected in a template code, thus making the templates difficult to develop.
- A C++ compiler uses the code specialization approach in translating its templates. Every use of the template may cause the compiler to generate extra code for the instantiation of the template leading to code bloat when they are indiscriminately used, thus resulting an excessively large executables. Also the extra instantiation generated by the templates can cause debuggers to have difficulty working with templates. For example, when setting a debug breakpoint within a template from a source file where this setting may be missed set in the actual instantiation desired or may set a breakpoint in every place the template is instantiated. Note that code bloat is not inevitable in C++ and can generally be avoided by an experienced programmer.

The term concept has emerged to denote specifically the interface description for templates that are at the heart of C generic programming frameworks [Aus99]. Back then, although concepts play an obviously critical role in generic programming, they are typically used implicitly since there is no language supporting it.

2.3.2 C++ Concepts

In C++, template classes and functions necessarily impose restrictions on the types that they take. In the case of the function, the requirement an argument must meet is clear, but in the case of a template the interface an object must meet is implicit in the implementation of that template. Concepts provide a mechanism for codifying the interface that a template parameter must meet. The primary motivation of the introduction of concepts is to improve the quality of compiler error messages. If a programmer attempts to use a type that does not provide the interface a template requires, the compiler will generate an error. However such errors are often difficult to understand, especially for novices. The two main reasons for this are that error messages are often displayed with template parameters spelled out in full which leads to extremely large error messages and that the compiler does not immediately refer to the actual location of the error. In an attempt to resolve this issue, C++0x adds the language feature of concepts [RS06]. Similar to how object-oriented programming use a base-class to define restrictions on what a type can do, a concept is a named construct that sepecifies what a type must provide. Unlike object-oriented programming, however, the concept definition itself is not always associated explicitly with the type being passed into the template, but with the template definition itself.

One example of the idea of concept is to avoid the problem that we saw lots of times when for instance Spirit (object-oriented parser generator, see Section 4.2) gives an unreadable error messages. What actually happens is that somewhere in the code a template was wrongly used, but the compiler doesn't see this wrong use of a template, and instead starts to unfold the templates until an error in the unfolded code is found. With Concepts one can specify the template parameter which has these properties. If this template is now applied to a parameter which does not have these properties, an error message should be displayed at this point. This will make the template mechanism of C++ more type safe. In Java these problems have been avoided by demanding that for template parameters one has to specify which interface they need to implement.

In some sense C++ Concepts make a similar step to what was done in Java. This makes C++ superior to Java, because Concepts are more flexible, since one can demand arbitrary logical combinations of guards whereas the mechanisms in Java and other templates mechanism in object-oriented languages only demand that a certain interface is implemented by the template parameter.

The first version of the concept checking system was developed by Jeremy Siek while working at SGI STL in their C++ Compiler and library group which is now part of the SGI STL distribution [SL00]. The definition of concept checking classes in the system originally introduced in the Boost concept checking library was greatly simplified at the price of less helpful error messages. This differs from the concept checking in SGI STL. At the moment, concepts are planned to be added as a language construct to C++. More details on this can be found in the articles [RS06] and [Str03].

2.3.3 Generic Programming Features in Other Languages

Some C++ based languages such as Java and C# left out templates due to the problems with templates. These languages have adopted other methods in dealing with these problems. C# is currently adopting generic programming features comparable to templates. Java supports generic as of J2SE 1.5.0. Generics in Java supports template programming as advanced as C++ but less powerful. In Java, generics are checked at compile time for type correctness, and the generic type information is then removed through a process called type erasure which is unavailable at runtime. Ada's generics predate templates. Ada has had generics since it was designed in 1977-1980. The standard library uses generics to provide many services.

In Haskell, some language extensions have been developed for generic programming and in the language itself contains some generic aspects. In Haskell [Hut06] itself, for example, a user-defined data type of binary trees with labels of type a attached to the nodes and leaves as follows :

```
data BinTree a = Leaf a | Node (BinTree a) a (Bintree a)
        deriving (Eq, Show)
```

The keyword `deriving` followed by the two type classes `Eq` and `Show`, will make it possible for the programmer to automatically have an equality function defined `BinTree (==)` as well as a way to transform them into printable output. The Haskell compiler can in a generic fashion generate instances of particular functions for any given data type. Other instances that can be generated are `Ord` and `Read`.

PolyP was the first generic programming extension for Haskell where the generic functions are called polytypic. This extension introduces a special construct in which such polytypic function can be defined through structural induction over the structure of the pattern functor of a regular datatype. *Generic Haskell* is another extension to Haskell which is developed at the Utrecht University. It provides type-indexed value which are values indexed over the various Haskell type constructors such as unit, primitive types, sums, products and user defined type constructors. The resulting type-indexed, can be specialized to any type like the kind-indexed types, generic application, generic abstractions and type-indexed types. The *Scrap your boilerplate* approach is a lightweight generic programming approach for Haskell. In this approach programmers can write generic functions such as traversal scheme as well as generic read (`gread`), generic show (`gshow`) and generic equality (`geq`). This approach is based on just a few primitives for type-safe cast and processing constructor applications.

2.4 Object-oriented Programming

Quotes from Samuel P. Harbison "*The surest way to improve programming productivity is so obvious that many programmers miss it. Simply write less code*" [HS02]. One way of achieving this is by implementing the object-oriented paradigm of programming where emphasis is on generality and reusability. In object-oriented programming, reusability is supported by inheritance and polymorphism. Object-oriented programming is characterized by programming with objects, messages, and hierarchies of objects [Cox86]. This section will start off by giving a glimpse of the history of object-oriented programming and what is meant by object-orientation in programming before discussing further on its concepts and usage.

2.4.1 History of Object-oriented Programming

The first two object-oriented languages are SIMULA I and Simula 67 which were introduced in the 1960s. The Simula languages were developed at the Norwegian Computing Center in Oslo, Norway, by Ole-Johan Dahl and Kristen Nygaard. Simula 67 introduced most of the key concepts of object-oriented programming such as objects and classes, subclasses and virtual procedures, combined with safe referencing and mechanisms for bringing into a program collections of program structures described under a common class heading (prefixed blocks). SIMULA I got a reputation as a simulation language but it turned out to be a general programming language due to it possessing interesting properties of a general programming language.

Starting in the early 1970s, Simula concepts have been important in the discussion of abstract data types and of models for concurrent program execution. Simula was used as a platform for the development of Smalltalk extending object-oriented programming by the integration of graphical user interfaces and interactive program execution. In 1980, "C with Classes" was released as an enhanced version of C which included classes for data abstraction. It was designed so that a preprocessor could make direct conversion from classes to struct. In 1982, Bjarne Stroustrup began working on a better version of "C with Classes" which would be a more true object-oriented superset of C. In 1983, the first version of C++ was released and more advanced object-oriented features were rapidly introduced until 1985, when the first commercial version was released. More features including templates were continually added until 1982, at which time C++ obtained some level of stability and an ISO version of C++ was finalised in 1998. In the late 1990s, object-oriented programming became the dominant style for implementing complex programs with large number of interacting components. A large variety of object-oriented programming languages have been developed, among them are Eiffel, CLOS (object-oriented enhanced version of LISP), Object Pascal, Ada 95 (Ada2005 still in the process of enhancement) and particularly the internet-related Java which has in particular gained popularity now.

Due to the initiative of programmers in searching better ways for people working with computers, object-oriented programming techniques have evolved from procedural programming techniques.

In procedural languages, object-oriented programming appears as a form where data types are extended to behave like a type of an object, similar to an abstract data type with an extension such as inheritance in object-oriented programming, and each method is actually a subprogram which is syntactically bound to a class. Object-oriented programming is an abstraction and generalization of imperative programming. Imperative programming involves a state and a set of operations that changes the state whereas object-oriented programming involves collections of objects where each object has a state and a set of operations to transform the state. Thus, we can say that object-oriented programming focuses on data rather than on control. In an object-oriented language, programming requires the programmer to think in terms of a hierarchy of objects and the properties possessed by the objects where emphasis is on generality and reusability. Object-oriented programming uses the metaphor of message passing to capture the interaction of objects [Laf94]. Before going further, we will first discuss concepts that are emphasized in object-oriented programming.

2.4.2 Concepts in Object-oriented Programming

An **object** models the entity of concern in an application. It encapsulates its structure and behaviour through its data structure and functions. In conventional programming, an object is referred to as a variable which is an instance of a type. This is similar to an object as an instance of a class. A class describes a group of similar objects. It names and types the components of data structure of each object in the class and declares the function that can be applied to them [Eck00]. The structure of an object is described by member fields and the behaviour is described by member functions. The member function and member fields are not the description of an individual object but for a group of similar objects or **class**. James Rumbaugh(1991) define a class as a group of objects with similar properties(attributes), common behaviour (operations), common relationships to other objects and common semantics(meaning). A class is depicted in a diagram in Figure 2.4.

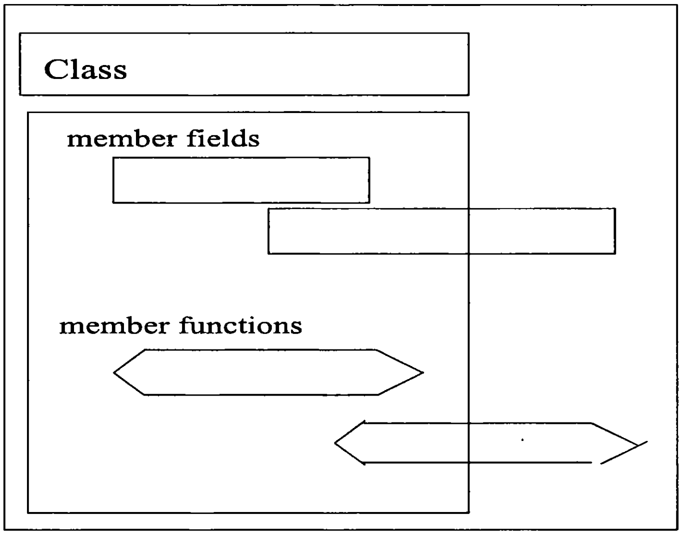

Figure 2.4: Class Diagram

We can depict an object as a box which denotes the boundaries between the inside and outside of the object. Inside the box are the local variables i.e. the member fields and functions. Everything that is completely inside the box is hidden from the outside meaning they are encapsulated. **Encapsulation** is one of the major features of object-oriented methods. By hiding both the data and method within an object, a level of encapsulation that no earlier methods can approach is achieved, resulting in stability and portability [Cox86] Stability here means that future changes to the system designed using the object techniques will only involve in reusing the classes that have been defined and maybe few changes to the reusable objects. Portability is increased from the ability to reuse a class in a new project or a new platform. New fields or new methods are added to the objects at each reuse making it more and more reusable.

Fields and functions that extend outside the box make up the object interface and are accessible. Interface makes possible any access to the object's member features. All the variables (functions and fields) that are declared under the keyword `public` in C++ are accessible. An object is depicted in the Figure 2.5.

An example of a class is the `ObjShapes` class. It is a class of shapes that can consist of circles, rectangles and etc. It contain shape and colour of the objects and has member functions `print` for printing the attributes of the shape objects and `setfields` for setting the attributes of the objects. The coding of the class in C++ is shown below :

```
class ObjShapes {
public:
   string shape;
   string colour;
   void setfields(string s,c);
   void print();
```

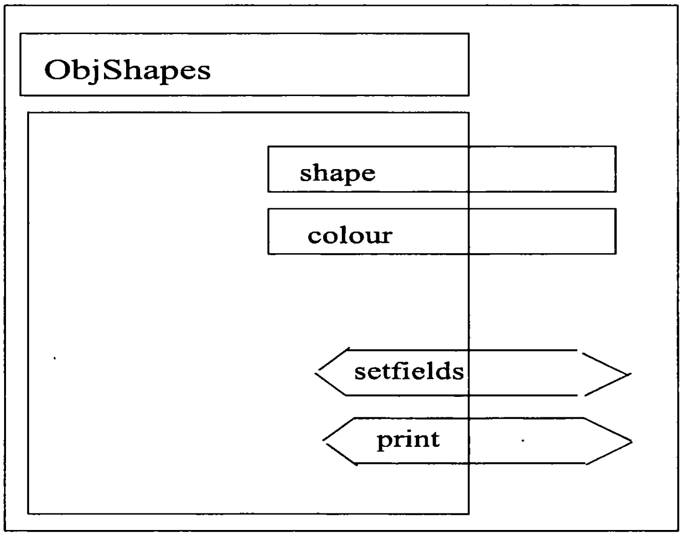

Figure 2.5: Object Diagram

```
}
```

An object is referenced by a variable or a data field. There are several ways in referencing an object i.e. by :

- a variable that contain an object ,eg. `ObjShapes obj1` where `obj1` is an object of a class `ObjShapes`.
- a variable that is a pointer to an object, eg.`ObjShapes *obj_ptr`
- a variable that is a reference to an object, eg. `ObjShapes &obj_ref` where we can say it is a second name of the object. A complete declaration of this reference variable example is

  ```
  ObjShapes obj1;
  ObjShapes &obj_ref = obj1;
  ```

Inside object, computation is achieved by sending messages to other objects which is called "message passing". An object executes one of its methods as a result of receiving a message. A message states what should be done by the object whereas a method expresses how it will be done. Message passing is similar to a function call in conventional programming. In order to print the attributes of the object `ObjShapes`, the message `print` must be passed to the object identifier or variable. If the object variable is a complete object (`obj1`), the message passing is executed by the statement

```
obj1.print();
```

Message passing for a variable that is a pointer(`obj_ptr`) to an object can only be executed after the object is created which is shown below:

```
obj_ptr = new ObjShapes;
```

The message `print` is sent by the code:

```
obj_ptr->print();
```

where the object react by executing the method print(). If the message sent is coded in the main program, so the main program is the sender of the message. In responding to a message sent, an object has to lookup the appropriate method. The binding of the method name to it's body is done routinely by a compiler in conventional programming languages. There are two types of binding i.e. **static** and **dynamic binding** where the former is done during compile time whereas the latter is done during runtime [AU01]. If the method exist during compile- and run-time, the result will be the same. The difference can be seen when the method does not exist, where static binding will report a compile error, whereas dynamic binding will result in a run-time error. C++ is a strongly typed programming language, so if an object is of a certain class it will always be of that type. For example, obj1 will always refer to ObjShapes. Therefore, obj1 will not change its class between compile- and run-time.

Dynamic binding plays an important part in the context of class hierarchies where a class inherits member functions from its superclass. Eventhough dynamic binding incurs a performance penalty due to an extra lookup at run-time, it is negligible due to optimisations carried out by the current object-oriented compiler technology and also because of the rapid increase in hardware performance.

Object-oriented programming languages use classes to categorize entities that occur in an application. Related categories form hierarchies that has "is a" relationship. This idea of relationship is used in relating classes in an object-oriented programming languages. For example a class ObjShapes can be a circle or rectangle. In other words, class Circle and Rectangle are derived classes or subclasses of ObjShapes making ObjShapes a superclass or base class. Circle and Rectangle inherits all the features of ObjShapes. This concept is called **inheritance** which plays an important role in defining object-oriented programming languages. The class ObjShapes and its descendants are coded in C++ as shown below:

```
class ObjShapes {    //Class definition for ObjShapes
public:
   string shape;
   string colour;
   //consructor
   ObjShapes(){shape = " ";colour = " ";};
   //constructor
   ObjShapes(string s,string c){shape = s; colour = c;};
   //member function
   void print();
};

void ObjShapes::print(){
   cout<<"Shape is "<<shape<<endl;
   cout<<"and the colour is "<<colour<<endl;
};
```

```
\\Class definition for the descendants Circle and Rectangle
class Circle::public ObjShapes{
public:
      real  radius;
      \\constructor
      Circle(string shape, colour):ObjShapes(shape,colour){};
      void setradius(float);
      void print();
};

void Circle::setradius(float r){
     radius = r;
};
void Circle::print(){
    ObjShapes::print();
    cout<<"It's radius is "<<radius<<endl;
}

class Rectangle::public ObjShapes{\\class definition for Rectangle
public:
      float length, breadth, area;
      \\constructor
      Rectangle(string shape, colour):ObjShapes(shape,colour){};
      void calcarea(float, float);
      void print();
}

void Rectangle::calcarea(float l, float b){
     length = l;
     breadth = b;
     area = length * breadth;
}

void Rectangle::print(){
    ObjShapes::print();
    cout<<"It's sides are "<<length<<" and "<<breadth<<endl;
    cout<<"It's area is "<<area<<endl;
};
```

In the example above, class `Circle` and `Rectangle` inherits the field `shape` and `color`, and the method `print`. In order to show how the message passing between objects of the classes above, consider the following program segment:

```
int main(){

  Circle c1("circle","blue");
  c1.setradius(4.5);
  Rectangle rl("rectangle","green");
  rl.calcarea(5.1,6.2);
  cl.print();
  rl.print();
}
```

The Figure 2.6 shows the message passing and method lookup for the above code segment.

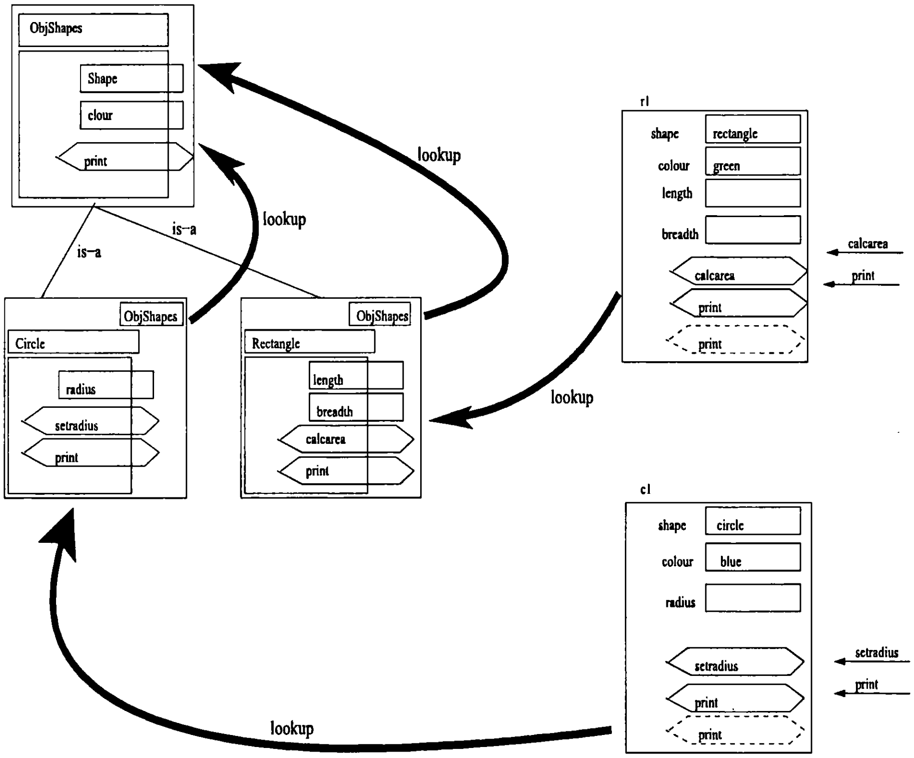

Figure 2.6: Message Passing and Method Lookup

The program declares two objects, `cl` and `rl`, set the radius for `cl` and calculate the area of `rl`. The first invocation of `print` refers to the `print` member function of `Circle` and the second invocation of `print` refers to the `print` member function of `Rectangle`. The output of this program segment is the same whether with static or dynamic binding. If one wants to express a member function to be bind dynamically, the member function has to be designated as `virtual`. The keyword `virtual` signals the intention to use dynamic binding for designated member function. For example to enable dynamic binding for the member function `print` in class `ObjShapes`, the declaration of the member function `print` is coded as follows:

```
virtual void print();
```

Dynamic binding must be a major criterion in calling a language an object-oriented programming language. C++ supports dynamic binding making it a truly object-oriented programming language. Another kind of inheritance is **multiple inheritance**. Multiple inheritance is when a child class or subclass is derived from more than one base class. Details of it will not be discussed here.

Suppose a set of shape classes such as `Circle, Triangle, Square` and etc are derived from base class `Shape` . Let say we need each shape classes to be treated generically as objects of base class `Shape` so that to draw a shape we could simply call function `draw` in each of base class `Shape` and let the program determine dynamically which derived class `draw` function to use. Thus we should declare `draw` in the base class as virtual function and we override `draw` in each of the derived classes to draw the appropriate shape. For example,

```
        virtual void draw() const;
```

may appear in base class `Shape` which declares that function `draw` is a constant function that takes no argument, returns nothing and is a virtual function.

A class that has direct instances are called a concrete class meaning that it has instances of its class not of its subclasses. A class that does not have direct instances is called an **abstract class**. An abstract class serves as a common base but will not have any instances. A class is made abstract by declaring one or more of its virtual functions to be "pure". A pure virtual function is one with an initializer of = `0` in its declaration as shown below:

```
    virtual void draw() const = 0; //pure virtual
```

The sole purpose of an abstract class is to provide an appropriate base class from which classes may inherit interface and/or implementation. They are too generic to define real objects.

A hierarchy does not need to contain any abstract class but there are many good object-oriented systems that have class hierarchies headed by an abstract class. An example is a shape hierarchy where it is headed by abstract class `Shape` and the next level down the hierarchy, there are two or more abstract base class `TwoDimenShape` and `ThreedimShape`. The next level concrete classes are defined such as for two dimensional shapes will be circles and squares and for three dimensional shapes will be spheres and cubes. The usage of the concept of abstract classes and virtual function can be seen in the translation of the function types in C++ (see Chapter 4).

Polymorphism is another key concept in object-oriented programming. According to Webster's dictionary, the word polymorphism means "occuring in various forms". But in the context of object-oriented programming, polymorphism refers to behaviours that have the same name and meaning but actually are different depending on the class concerned. Polymorphism is the ability to write several versions of method(function, subroutine) in different classes of a subclass hierarchy with the same name and rely on the object-oriented environment to establish which version should be executed depending on the class of the target object when the method is invoked [DD01]

Polymorphism means a behaviour may be inherited either unchanged, or totally different between the superclass and the subclass, or it is specialized for a particular subclass. Polymorphism is implemented through virtual functions. When a request is made through a base class pointer (or reference) to use a virtual function. C++ chooses the correct overridden function in the appropriate derived class associated with the object. Sometimes a non virtual function is defined in a base class and overridden in a derived class. If such a member function is called through a base class pointer to the derived class object, the base class version is used. If the member function is called through a derived class pointer, the derived class version is used. As we can see from the example given, the method `print` is coded specially for certain subclasses which are Circle and Rectangle. The method `print` in `Rectangle` and `Circle` class overrides the method `print` in the superclass `ObjShapes`. Generally we can say that the subclass version of an attribute or operation/method is said to **override** the version from the superclass because it is executed in preference to the superclass version.

Through the use of virtual functions and polymorphism, one member function call can cause different action to occur depending on the type of the object receiving the call which gives programmer tremendous expressive capability [DD01]. With virtual functions and polymorphism, it is possible to design and implement systems that are more easily extensible. Programs can be written to generically processed objects of existing classes in a hierarchy that derived from a base class objects. Classes that do not exist during program development can be added with little or no modifications to the generic part of the program as along as those classes are part of the hierarchy that is being processed generically .

There are two fundamentally different kinds of polymorphism which was originally described informally by Christopher Strachey in 1967. They are ad-hoc and parametric polymorphism. Ad-hoc polymorphism is when the range of actual types that can be used is finite and the combinations must be specified individually prior to use, while parametric polymorhism is when all code is written without mention of any specific type and thus can be used transparently with any number of new types. Ad-hoc polymorhism is generally supported in object-oriented programming through object inheritance which was described earlier. Parametric polymorhism is widely supported in statically typed functional programming languages and in the object-oriented community, programming using parametric polymorphism is often called generic programming (see section 2.3).

2.5 C++ as an Object Oriented Programming Language

In the previous sections we have discussed thoroughly what are the concepts that are needed in writing an object-oriented program. We also have gone through the history of object-oriented programming where we now know how it started. However, we have not yet mentioned the definition of object oriented programming. Traditionally, we can say a language or technique is object-oriented if and only if it directly supports abstraction (providing some form of classes and objects), inheritance (providing the ability to build new abstractions out of existing ones) and run-time polymorphism (providing some form of run-time binding). This definition includes all major languages which are referred to as object-oriented such as Ada95, Beta, CLOS, Eiffel, Simula, Smalltalk, Java and C++ [Laf94]. As mentioned, C++ is a paradigm-neutral language meaning that it was designed to support a range of styles that are considered fundamentally good and useful. By sharing a common type system, a common toolset and etc., significant benefits can arise from it such as enabling groups with moderately differing needs to share a language rather than having to apply a number of specialized languages.

The range of facilities or properties that C++ supports whether they were object-oriented or otherwise, can be listed below:

i) Abstraction is the ability to represent concepts directly in a program and hide incidental details behind well defined interfaces. This ability is the key to every flexible and comprehensible system of any significant size.
ii) Encapsulation is the ability to provide guarantees that an abstraction is used only according to its specification which is crucial in defending abstractions against corruption.
iii) Polymorphism is the ability to provide the same interface to object with differing implementations. Polymorphism is crucial in simplifying code using abstractions.
iv) Inheritance is the ability to compose new abstractions from existing ones. It is one of the most powerful ways of constructing useful abstractions.
v) Genericity is the ability to parameterize types and functions by types and values. It is essential for expressing type-safe containers and a powerful tool for expressing general algorithms.
vi) Coexistense with other languages and systems and this feature is essential for functioning in real world execution environments.
vii) Runtime compactness and speed which is essential for classical systems programming
viii) Static type safety is an integral property of languages of the family to which C++ belongs to and it is valuable both for guaranteeing properties of a design and for providing run-time and space efficiency.

The list of properties and facilities listed are taken from [Str95]. These facilities and general properties can be supported in several alternative ways such as supporting them in the core language or in a library.

C++ supports all the facilities and properties that defines an object-oriented programming language such as abstraction, encapsulation, polymorphism and inheritance, thus we can say that C++ is truly an object-oriented programming language, even though it is also a general purpose language due to its design which supports multiple styles of programming.

Chapter 3

λ-Calculus and Functional Programming

Generally speaking, functional programming is a style of programming in which the basic method of computation is the application of functions to arguments [BW88]. The definition of a function in functional programming is an expression rather than a sequence of commands and execution of a functional program means the evaluation of the expression. Expressions in a functional language can be constructed, manipulated and reasoned about, like any other kind of mathematical expression using more or less familiar algebraic laws for the operators.

It has been said that functional programs do not use variables but this is not exactly true because there are variables as arguments of functions and also in the let expressions. However, variables get their value only once, so the value never changes. This avoids the aliasing problem. Furthermore, this applies only to pure functional programming languages like Haskell. In ML and Lisp, side effects do occur. The idea of executing commands sequentially (like in an imperative program) in functional programs is meaningless since the sequence of commands does not make any difference because there is no state to mediate between them. Functions in a functional program can be used in more sophisticated ways such as they can be passed to other functions as arguments and returned as results and generally can be calculated with. Functional languages use recursive functions (functions that are defined in terms of themselves) instead of sequencing and looping.

Functional programming is declarative in the sense that we say what we want rather than how to get it. A characteristic feature of functional programming is that if an expression possesses a well-defined value, then the order in which a computer may carry out the evaluation does not affect the outcome [CM98]. However, this feature is true only for pure functional programming language such as Haskell but not for ML. We can say that the meaning of an expression is its value and the task of a computer is simply to obtain it.

Functional programs correspond more directly to mathematical objects making it easier to reason about them. Most functional programming languages are based on a simple and elegant mathematical foundation i.e. the λ-calculus. Alonzo Church [FH88] defined a calculus that can express the behaviour of function as an effort to capture the computational meaning of mathematical functions. The history of functional programming will be discussed in the next section in which we will discuss the functional languages since the beginning until now. Then we will discuss details of λ-calculus, since it is, together with combinatory logic, one of the roots of functional programming.

3.1 History of Functional Programming

One of the main roots of functional programming are the λ-calculus and combinatory logic, which were introduced by Alonzo Church, Haskell Curry and Moses Schönfinkel in the 1920s and 30s. Schönfinkel developed a simple theory of functions in the year 1924 and at about ten years after that Church introduced the λ-calculus and used it to formalize the syntax of Whitehead and Russell's Principia Mathematica. In the 1940's, Haskell introduced combinatory logic which is a variable free theory of functions. In the late 1950's Church's λ-notation for functions led to the first version of LISP by McCarthy. LISP was extremely successful and is still being used. Dialects of LISP include Common Lisp, Scheme and elisp for emacs. LISP had many innovations which was influential on both theoretic and practical aspects of functional programming which include the use of garbage collection as a method of disposing of unused cells, implementing static scoping by using closures, invention of conditional expression in writing recursive functions (which involves lazy evaluation) and the use of higher order operations on list. In 1978 Backus defined FP in his Turing Award lecture. His lecture gave a significant impact on the functional language field [Tan04].

Modern functional languages have more advanced features such as static type systems, polymorphism, type inference, algebraic data type, pattern matching and lazy evaluation. These features contribute a great deal in making functional programming more practical. Examples of modern functional languages are ML, Miranda and Haskell. ML (meta language) was defined by researches Gordon, Miller et al. for the use in describing proof search strategies. Later (1978) they found out that ML could also be used as a general programming language. ML was the first language to use the Hindley-Milner type system (now known as type inference) which is the basis for the type system for most modern functional language. Now there are two important dialects of ML that is Standard ML and CAML. Miranda [MV97] was developed by David Turner in 1985. Turner implemented Miranda using the idea of combinators (fixed set of basic functions). Miranda is a language with lazy evaluation. A committee was formed in 1987 as an effort to define a standard functional language with modern features resulting in the development of Haskell named after the logician Haskell B. Curry [Hug89]. Haskell has all the modern functional language features such as higher-order functions, type inference, lazy evaluation and user defined data types.

3.2 λ-Calculus

The λ-calculus was developed by mathematicians before the development of computers in order to obtain a notation for writing down functions. One way of describing functions mathematically is through their extension which can be a list of pairs of input, output values or as a graph from one domain to the other. But not all functions are computable even though they are describable. λ-calculus is an attempt to write down functions that could actually be evaluated in the real world. The following is the definition of λ-calculus [Pau00]:

Definition: The terms of the λ-calculus, known as λ-terms, are defined inductively from a given set of variables $x, y, z, \ldots$ as follows:

$$\begin{array}{ll} x, & \text{where } x \text{ is a variable} \\ c, & \text{where } c \text{ is a constant} \\ (\lambda x.r & \text{(abstraction), where } r \text{ is a term, and } x \text{ is a variable,} \\ (r\ s) & \text{(application), where } r \text{ and } s \text{ are terms} \end{array}$$

We define $\lambda x, y.r := \lambda x.\lambda y.r$, similarly for $\lambda x, y, z.r$ and similar expressions. The symbol λ is completely arbitrary bearing no significance meaning to it. The symbol arose by a completed process of evolution. Originally, the 'hat'(∧) notation $t[\widehat{x}]$ is used by Principia Mathematica for the function of x yielding $t[x]$. Church modified it to $\widehat{x}.t[x]$, but it turned out as $\wedge x.t[x]$ due to the fault of the typesetter which could not place the hat on top of the x. The symbol then mutated into $\lambda x.t[x]$ [Har97].

3.2.1 Variable Binding

The λ-term $\lambda x.r$ refers to a variable defined by surrounding context. For this term, the λ-abstraction defines a new function with argument variable x and body r. We call x as the bound variable to the abstraction. Any occurrences of x in r is bound by the abstraction. For example, in the λ-abstraction $\lambda x, y.(x\ y)z$, x and y are bound variables because they are bound to the abstraction whereas z occurs free. A closed term is one which all variables/identifiers are bound and we will consider a program in the λ-calculus to be any closed term.

The concept of free and bound variables can be defined as sets in the λ abstraction :

- The set of all bound variables $\mathrm{BV}(r))$ in r is given by:

$$\begin{array}{lcl} \mathrm{BV}(x) & = & \emptyset \\ \mathrm{BV}(\lambda\ x.r) & = & \mathrm{BV}(r) \bigcup \{x\} \\ \mathrm{BV}(r\ s) & = & \mathrm{BV}(r) \bigcup \mathrm{BV}(s) \end{array}$$

- The set of all free variables in $r(\mathrm{FV}(r))$ is given by:

$$\begin{array}{lcl} \mathrm{FV}(x) & = & \{x\} \\ \mathrm{FV}(\lambda\ x.r) & = & \mathrm{FV}(r) \setminus \{x\} \\ \mathrm{FV}(r\ s) & = & \mathrm{FV}(r) \bigcup \mathrm{FV}(s) \end{array}$$

A λ-term r is called *closed* if it has no free variables, i.e. $\mathrm{FV}(r) = \emptyset$. Closed λ-terms are also called *combinators*.

The application $r\ s$ defines the application of function r to the argument s. An example of this is shown below:

$$(\lambda x, y.y + x)3\ 4$$

where λ abstraction $\lambda x, y.y + x$ is first applied to 3 then to 4 i.e x takes the value 3 and y takes 4. The function application is left associative. How the application is evaluated will be discussed in Section 3.2.4.

3.2.2 Substitution

The function f such that $f(x) = r$ is represented by the λ-abstraction $\lambda x.r$ and when applied to s yields the result of substituting s for all free occurrences of x by r. Examples are as follows:

$(\lambda x.x)$	The identity function which returns its argument unchanged and is usually called I.
$(\lambda x.y)$	A constant function that returns y when applied to any argument.

Substitution of term t for all free occurrences of y in r, denoted $r[t/y]$, is defined as follows:

$$x[t/y] \equiv \begin{cases} t & \text{if } x = y \\ x & \text{otherwise} \end{cases}$$

$$(\lambda x.r)[t/y] \quad \equiv \begin{cases} (\lambda x.r) & \text{if } x \equiv y \\ (\lambda x.r[t/y]) & \text{if } x \notin \{y\} \cup \mathrm{FV}(t) \\ \lambda x'.r[x'/x][t/y] & \text{otherwise, where } x' \text{ is "fresh"} \end{cases}$$

$$(r\ s)[t/y] \quad = \quad (r[t/y]\ s[t/y])$$

λ-calculus would be inconsistent if we had defined substitution for λ-abstractions (second clause) naively, i.e. without replacing x with x' in the last case (α-conversion). For instance, the term $\lambda x, y.x$ when applied to an argument s should return the constant function $(\lambda y.s)$. However, in case $s\ \equiv y$; if we carried out the substitution directly, we obtain $(\lambda y.y)$ instead , which is the identity function. The free occurrence of x turns into a bound occurrence of y which is an example of variable capture. The substitution $r[s/x]$ is safe provided the bound variables of r are disjoint from the free variables of s :

$$\mathrm{BV}(r) \bigcap \mathrm{FV}(s) = \emptyset$$

In order to avoid a clash in variables, the bound variables of r might need to be renamed. This renaming is called β-conversion, and is defined in more detail in the next subsection. For example, we could change $\lambda y.x$ into $\lambda z.x$. Then an allowed substitution $(\lambda z.x)[y/x] \equiv \lambda z.y$ can be carried out. The result obtained is in this case a constant function.

3.2.3 Conversion

λ-calculus is based on three conversions which transform one term into another equivalent term. The conversions are α-conversion, β-conversion and η-conversion. Each of these conversions can be applied as well to any subterm. The formal definition of these conversions are as follows:

- α-conversion is the result of replacing a subterm of the form $\lambda x.r$ by $\lambda y.r[y/x]$, where y might not occur free or bound in r. We write then $s \longrightarrow_\alpha s'$ if s is obtained by applying this reduction to s.

  For example, $\lambda x.(x\ z) \longrightarrow_\alpha \lambda y.(y\ z)$.

- A β-redex of a term is a subterm of the form $(\lambda x.r)\ s$. A β-redex $(\lambda x.r)\ s$ reduces to $r[x/s]$. A term t β-reduces to t', written as $t \longrightarrow_\beta t'$, if t' is obtained by replacing a β-redex in t by its reduct.

  For example, $(\lambda x.(x\ x))(y\ z) \longrightarrow_\beta (y\ z)(y\ z)$.

- An η-redex of a term is a subterm of the form $\lambda x.r\ x$, where $x \notin \mathrm{FV}(r)$. An η-redex $\lambda x.r\ x$ reduces to r. A term t η-reduces to t', written as $t \longrightarrow_\eta t'$, if t' is obtained by replacing an η-redex in t by its reduct. For example,$(\lambda x.((z\ y)x)) \longrightarrow_\eta (z\ y)$

Among the three conversions, β-conversion is the most important since it represents the evaluation of a function on an argument. α-conversion is just a technical device to change the names of variables, while η-conversion is a form of extensionality. We do not consider the α-conversion in the thesis.

We demonstrate β-reduction by using the following example:

$$(\lambda x, y.y + x)3\,4$$

After the 1^{st} β-reduction , we will get:

$$(\lambda y.y + 3)4$$

Applying another β-reduction yields $3\ +\ 4$ which gives the result 7.

3.2.4 Reduction

Reduction corresponds to a systematic attempt to evaluate a term by repeatedly evaluating combinations $f(x)$ where f is a λ-abstraction. We say that the term is in normal form when no more reduction is possible except for α-conversion. For example, $\lambda x, y.y$ and $(x\ y)z$ are normal form. But many λ-terms cannot be reduced to normal form. As an example take $\Omega := (\lambda x.x\ x)(\lambda\ \ x.x\ x)$. The only reduction of Ω is to itself ($\Omega \longrightarrow \Omega$), β-reduction of Ω does not terminate and Ω does not have a normal form.

We define what it means for two terms to be α, $\alpha\beta$ and $\alpha\beta\eta$-equivalent. For this we need first the following auxiliary definitions:

- $r \longleftrightarrow_\alpha s$ if and only if $r \longrightarrow_\alpha s$ or $s \longrightarrow_\alpha r$
- similarly for $r \longleftrightarrow_\beta s$, $r \longleftrightarrow_\eta s$, $r \longleftrightarrow_{\alpha\beta} s$, $r \longleftrightarrow_{\alpha\beta\eta} s$, where $r \longrightarrow_{\alpha\beta} s$ if and only if $r \longrightarrow_\alpha s$ or $r \longrightarrow_\beta s$, similarly for $r \longrightarrow_{\alpha\beta\eta} s$

Furthermore, $r \longrightarrow^*_\beta s$ means that $r \equiv r_0 \longrightarrow_\beta r_1 \longrightarrow_\beta \cdots \longrightarrow_\beta r_n \equiv s$ for some $r_0, \ldots, r_n$, which means that r reduces to s in 0 or more steps. Notations like $r \longrightarrow^*_\alpha s$, $r \longleftrightarrow^*_\beta s$ etc. are to be understood similarly. By r and s being α ($\alpha\beta$, $\alpha\beta\eta$) equivalent, written as $r =_\alpha s$, ($r =_{\alpha\beta} s, r =_{\alpha\beta\eta} s$), we mean that $r \longleftrightarrow^*_\alpha s$ ($r \longleftrightarrow^*_{\alpha\beta} s, r \longleftrightarrow^*_{\alpha\beta\eta} s$). We identify r and s, if they are α-equivalent. Therefore we will omit in the following $\longrightarrow_\alpha$ steps, and write as a subscript of $\longrightarrow, \longleftrightarrow$ β instead of $\alpha\beta$, $\beta\eta$ instead of $\alpha\beta\eta$.

There are two main reduction strategies for β-reduction (note that we ignore intermediate α-reduction steps): *Normal-order reduction* is the strategy, in which the leftmost *outer*-most redex, is chosen. In contrast, an *applicative-order reduction* is a sequential reduction in which the leftmost *inner*-most redex is chosen first [Hug89]. Normal-order reduction corresponds to the principle of passing the arguments to a function initially unevaluated, whereas applicative-order strategy means that a function's arguments are evaluated before the function is applied.

The Church-Rosser Theorem states that β-reduction is confluent. The theorem says that whenever we reduce a λ-terms in two different ways (i.e. $r \longrightarrow^* s$, $r \longrightarrow^* s'$), then the two reducts can be joined together (i.e. there exists s'' so that $s \longrightarrow^* s''$, $s' \longrightarrow^* s''$). As a consequence we obtain uniqueness of normal forms: If r has normal forms s and s' then s and s' are equal up to α-equality.

However, not every reduction strategy will find the normal form. As an example of the difference between the applicative order reduction and the normal-order reduction we consider the following example:

- Applicative-order reduction:

$$\begin{aligned}(\lambda\, x.y)((\lambda\, x.x\, x)(\lambda\, x.x\, x)) &\Longrightarrow (\lambda\, x.y)((\lambda\, x.x\, x)(\lambda\, x.x\, x)) \\ &\Longrightarrow \vdots\end{aligned}$$

- Normal-order reduction:

$$(\lambda\, x.y)((\lambda\, x.x\, x)(\lambda\, x.x\, x)) \Longrightarrow y$$

From the example above, we can say that applicative-order reduction is not always adequate and the strongest completeness and consistency result can be achieved with normal-order reduction.

Let $t := y + x$. The abstraction $(\lambda y.t)$ contains x as free and each x it stands for a function over y. The abstraction $\lambda x, y.t$ contains no free variables and when applied to the arguments r and s, the result is obtained by replacing x by r and y by s. In other words we perform two β-reductions which can be shown symbolically as follows:

$$((\lambda x, y.t)r)s \longrightarrow_\beta (\lambda y.t[r/x])s \longrightarrow_\beta t[r/x][s/y]$$

This technique is called currying after Haskell B. Curry. An example would be the function $(\lambda x, y.x + y)$ which can be applied to 3 to yield the function $\lambda y.3 + y$ and then to 4 in order to obtain $3 + 4$.

As mentioned previously, the order of reduction can be applicative and normal. For a function application $(\lambda x.f)e$, the normal-order reduction strategy will reduce the redex $(\lambda x.f)e$ first before reducing e (being a subterm of the reduction $f[x/e]$ of $(f\ \ e)$ to a value. Due to this it is called **call-by-name** parameter passing. The applicative-order reduction strategy will reduce e to a value v before carrying out the reduction $(\lambda x.f)\ \ v \longrightarrow_{\beta} r[x/v]$. Therefore it is called **call-by-value** parameter passing.

3.2.5 Lazy evaluation

In the previous subsection call-by-value and call-by-name were introduced in terms of reduction strategies. In this section we will investigate call-by-name in more detail. Evaluation means reducing a λ-term until one obtains a normal form. There are two main ways of evaluating λ-terms: call-by-name and call-by-value. Call-by-name evaluation corresponds to lazy evaluation, where expressions are passed around unevaluated for as long as possible. Therefore in lazy evaluation function arguments are not evaluated, until needed in order to compute the result of the functions. On the other hand, call-by-value evaluation corresponds to eager evaluation where all expressions are evaluated before being passed as function arguments. Hence call-by-value requires that function arguments be reduced to values before the function is processed.

In a call-by-value setting functions are strict, which means if the result of one of the arguments is undefined, the result of applying this function to its argument is undefined as well. For instance, if c is a constant, $\lambda x.c$ applied to the undefined argument Ω is undefined. In a call-by-name setting functions can be non-strict, which means that they can have a defined value even if one of its arguments is undefined. In call by name $\lambda x.c$ applied to Ω has the defined result c.

The evaluation order can have an effect not only on execution speed but on program correctness as well. A program that encounters a dynamic semantic error or an infinite loop under applicative-order evaluation may terminate successfully under normal-order reduction. Expressions in a strict language can safely be evaluated in applicative-order but not for a non-strict language. A language is said to be strict if it requires all functions to be strict. It is a non-strict language, if it allows the definition and use of non-strict functions.

One possible problem of normal-order evaluation is inefficiency, since we obtain duplication of computation. But this inefficiency can be overcome without sacrificing its terminating property by using pointers to arguments. The idea is when reducing an application $(\lambda x.r)s$, we can first create a pointer to expression s and then reduce $(\lambda x.r)s$ to r', which is r with all x replaced by the pointer to s. If we need to reduce the pointer when reducing r', we can reduce the expression s pointed by the pointer. The point here is that every time we encounter this pointer in r', s need not to be reduced again since it has already been reduced the first time. This strategy can also be called call-by-need since s is evaluated whenever needed and it will be evaluated at the most once.

Lazy evaluation gives the advantage of normal-order evaluation (not evaluating unneeded subexpression) while running within a constant factor of the speed of applicative-order evaluation for expressions in which everything is needed. The principle problem with lazy evaluation is its behaviour in the presence of side effects [Sch00]. When using constants with side effects, the order of evaluation matters. For instance if we allow the statement $x := x + 1$, which has the side effect of incrementing the value of variable x by 1, the evaluation of another expression t, which refers to x, depends on, whether it is evaluated before or after the side-effect took place. When using call-by-value evaluation, it is easy to predict the evaluation order – the arguments of a function are evaluated first, then the function is evaluated, whereas with call-by-name and call-by-need, the order is difficult to predict. That is the reason why constants with side effects are usually not used in lazy languages.

The advantage of lazy evaluation is that it uses sometimes less reduction steps than applicative-order reduction (although with more implementation and runtime cost) and that it guarantees to find the normal form of an expression if there is one, whereas eager evaluation might not find the normal form even if it exists.

Lazy evaluation is particularly useful for infinite data structure such as infinite list. It is used for all arguments in Miranda and Haskell and also available in Scheme through explicit use of `delay` and `force`. The problem with side effects in lazy evaluation do no arise in Miranda and Haskell because they are pure functional language and Scheme leaves the problem up to the programmer to tackle. ML provides no mechanism for lazy evaluation, but it can be encoded.

3.2.6 Recursion

Recursion is essential in functional programming. Recursive or self-referential definitions are not needed to write recursive functions in the λ-calculus, since the function **Y** gives the effect of recursion. Y is known as the paradoxical combinator or as the fixed point operator. This Y combinator is realized based on the Fixedpoint Theorem and its simple proof. This theorem states that every λ-expression e has a fixed point e' such that $e' \longrightarrow e\ e'$, in particular, e' and $e\ e'$ are β-equivalent. In fact we can define $e' := e_0\ e_0$ where $e_0 := \lambda x.e\ (x\ x)$ for some $x \notin \mathrm{FV}(e)$, and one immediately sees that $e' \longrightarrow e\ e'$.

By replacing e with a variable y and λ abstracting y we obtain the famous fixed-point combinator

$$\mathrm{Y} := \lambda y.(\lambda x.y\ (x\ x))(\lambda x.y\ (x\ x))$$

which computes for every term e a fixed point $Y\ e$. Indeed,

$$\mathrm{Y}\ e \longrightarrow (\lambda x.e\ (x\ x))(\lambda x.e\ (x\ x)) \longrightarrow e\ ((\lambda x.e\ (x\ x))(\lambda x.e\ (x\ x))) \longleftarrow e\ (\mathrm{Y}\ e),$$

so $\mathrm{Y}\ e =_{\alpha\beta} e\ (\mathrm{Y}\ e)$.

Any recursive function can be written nonrecursively using Y. How is this done? Consider the recursive function F defined by

$$F\ =\ \ldots\ F\ \ldots$$

which can be rewritten as

$$F = (\lambda f. \ \ldots \ f \ \ldots) \ F$$

The equation above essentially says that F is a fixed point of the λ-expression $(\lambda f \ \ldots \ f \ \ldots)$, but Y exactly computes that. Hence, the recursive equation can be solved by the following nonrecursive definition for F:

$$F = Y(\lambda f \ \ldots f \ldots)$$

For example, the factorial function

$$F = \lambda n. \quad \text{if} \ (n = 0) \ \text{then} \ 1 \ \text{else} \ (n * F(n - 1))$$

can be written nonrecursively as

$$F = Y(\lambda f, n. \quad \text{if} \ (n = 0) \ \text{then} \ 1 \ \text{else} \ (n * f(n - 1)))$$

The ability of the λ-calculus to simulate recursion in this way is the key to its power and accounts for its persistence as a useful model of computation. This power is best expressed in Church's famous thesis which in its original form states that *effectively computable functions from positive integers to positive integers are just those definable in the λ-calculus.* Even though no proof can be given for his thesis but it gained support from Kleene who in 1936 showed that λ-definability was precisely equivalent to Gödel and Herbrand's notions of recursiveness. In 1937 Turing showed that Turing computability was also precisely equivalent to λ-definability.

In parallel with the development of the λ-calculus, Schöfinkel and Curry developed combinatory logic [Hug89]. Schönfinkel discovered that any function could be expressed as the composition of only two simple functions, K and S. Curry proved the consistency of a pure combinatory calculus and along with Feys, elaborated the theory considerably [Hug89]. Combinatory calculus plays a big role in the implementation of functional languages.

3.2.7 Higher-order Functions

In functional programming, higher-order functions; i.e., functions which take other functions as arguments, are treated as first class values, which can then be stored in data structures, passed as arguments, and returned as results. Let us consider the term for squaring integers which is defined as follows:

$$t \overset{def}{=} \lambda x.x * x$$

If we want to compute x^8 then this could be achieved by squaring x three times: $x^8 = ((x^2)^2)^2$. In the λ-calculus, this can be defined as the 'power-8' function:

$$P_8 \overset{def}{=} \lambda x.t(t(t \ x))$$

So we can see that taking a number to power 8 amounts to applying the squaring function Q three times. A λ-term which applies any function three times can be defined as follows:

$$t' \stackrel{def}{=} \lambda f, x.f\ (f\ (f\ x)))$$

So $t'\ f = \lambda x.f\ (f\ (f\ x))$, which is the function which applies f to x three times. The term P_8 can now be written as t' t, and 5^8 is t' t 5.

3.2.8 Typed λ-calculus

Types are a way of distinguishing different sorts of data such as booleans, natural numbers and functions so as to making sure that these distinctions are respected, for example by ensuring that functions cannot be applied to arguments of the wrong type. There are several reasons why types are added to λ-calculus. The main reason for introducing the typed λ-calculus is that the typed λ-calculus is strongly normalizing, so every reduction sequence terminates. From a logical point of view, one reason for considering type is that we would have a clearer picture of what sort of functions λ-terms denote if we knew exactly what their domains and codomains were, and only applied them to arguments in their domains. These considerations inspired Russell originally to introduce types in Principia Mathematica. Another reason for types is the fact that a compiler can generate more efficient code, and use storage more effectively by knowing more about a variable. As time went by, types also began to be appreciated more and more for their value in providing limited static checks on programs. Moreover types often serve as a useful documentation in programming and also they can be used to achieve better modularization and data hiding by 'artificially' distinguishing some data structure from its internal representation.

The basic idea of a typed λ-calculus is that every λ-term in the typed λ-calculus has a type. If A, B are types, then A → B is a type. A term s can only be applied to a term t, if the type of s is a a function type A → B and the type of t is A. The result $s\ t$ has then type B. This is strong typing where term t must have exactly the type A; there is no notion of subtyping or coercion. We will use t : A to mean t has type A. This is the standard mathematical notation where function spaces are concerned, because $f : \mathrm{A} \rightarrow \mathrm{B}$ means that f is a function from the set A to the set B. One property of types is that a type cannot be the same as any proper syntactic subexpression of itself.

There are two approaches in defining typed λ-calculus which are Church's approach (explicit) and Curry's approach (implicit). We will show both approaches of defining typed λ-calculus.

In Church's approach variables are typed, i.e. they are of the form v^{A} which means a variable is a pair consisting of a symbol v and a type A. In the case of constants, the type is preassigned. The generation rules for valid terms t in Church's style, together with their types C, written t : C, are:

$$\frac{}{v^{\mathrm{A}} : \mathrm{A}}$$

$$\frac{\text{Constant } c \text{ has type } \mathrm{A}}{c : \mathrm{A}}$$

$$\frac{s : \mathrm{A} \to \mathrm{B} \quad t : \mathrm{A}}{s\, t : \mathrm{B}}$$

$$\frac{t : \mathrm{B}}{\lambda v^{\mathrm{A}}.t : \mathrm{A} \to \mathrm{B}}$$

In contrast, in Curry's approach to typing, the terms are exactly as in untyped case, and a term may or may not have a type. But some purists would argue that this isn't properly speaking typed λ-calculus but rather untyped λ-calculus with a separate notion of type assignment. Curry-style of type assignment does not merely define a relation of typability in isolation but with respect to a context, i.e. a finite set of typing assumptions about variables. We write $\Gamma \vdash t : \mathrm{A}$ to mean 'in context Γ , the term t can be given a type A'. The elements of Γ are of the form $v : \mathrm{A}$, that is they are themselves typing assumptions about variables, typically those that are components of the term. We assume Γ never gives contradictory assignments to the same variable; if preferred we can think of it as a partial function from the indexing set of variables into the set of types. We write $\Gamma \vdash r : \mathrm{A}$ for $r : \mathrm{A}$ holds in context A. The Curry style typability rules are as follows:

$$\frac{v : \mathrm{A} \in \Gamma}{\Gamma \vdash v : \mathrm{A}}$$

$$\frac{\text{Constant } c \text{ has type } \mathrm{A}}{c : \mathrm{A}}$$

$$\frac{\Gamma \vdash s : \mathrm{A} \to \mathrm{B} \quad \Gamma \vdash t : \mathrm{A}}{\Gamma \vdash s\, t : \mathrm{B}}$$

$$\frac{\Gamma \cup \{v : \mathrm{A}\} \vdash t : \mathrm{B}}{\Gamma \vdash \lambda v.\, t : \mathrm{A} \to \mathrm{B}}$$

A special context is the empty context $\emptyset$, which makes no assumptions about the types of variables. Note that a context is a set of expressions of the form $(v : A)$. So in the last rule, Γ might contain $v : A$.

The rules above are to be regarded as an inductive definition of typability relation, so a term only has a type if it can be derived by the above rules. For example the identity function can be typed by first looking at the rule for variables, we have

$$\{x : \mathrm{A}\} \vdash x : \mathrm{A}$$

and therefore by the last rule we get:

$$\varnothing \vdash \lambda x.\, x : \mathrm{A} \to \mathrm{A}$$

This example illustrates the need for context, because without it we could not deduce x : B for any B. In the last step we derived $\lambda x.x$: A → A without any context. Note that we obtain $\lambda x.x$: A → A for any type A, so a λ-term can have many types. This problem does not arise in Church typing, since in that either both variables have type A or else the two x's are actually different variable, since their types differ and the types are a component of the term. $\lambda x^{\mathrm{A}}.x^{\mathrm{B}} : \mathrm{A} \to \mathrm{B}$ is reasonable for A ≠ B, however in this term x^{B} is a variable different from x : A, and x^{B} occurs free. In fact the second term is α-equivalent to $\lambda y^{\mathrm{A}}.x^{\mathrm{B}}$.

Type preservation is the property that if a term reduces to another term, its type is preserved. In the context of Curry typability it says that if $\Gamma \vdash t : \mathrm{A}$ and $t \to t'$, then we have $\Gamma \vdash t' : \mathrm{A}$. The Curry typing system gives a form of polymorphism in that a given term may have different types. In polymorphism, all types bear a systematic relationship to each other and all types following the pattern are allowed. For example, the identity function has types A → A, B → B or (A → B) → (A → B), but all instances have the same structure.

There exists a third style of typing which looks very similar to Curry style typing, because it uses contexts, but which is in fact rather a variant of Church style typing: The type of a variable is declared in a context, but in the rule for λ-abstraction, the type of the abstracted variable was given by the context is kept in the λ-abstraction. Hence, the rules as in the Curry system, except for the abstraction rule which becomes

$$\frac{\Gamma \cup \{v : \mathrm{A}\} \vdash t : \mathrm{B}}{\Gamma \vdash \lambda v : \mathrm{A}.\, t : \mathrm{A} \to \mathrm{B}}$$

It is this variant of the Church-style typing which is actually used in the next chapter.

3.3 Functional Programming as an Implementation of λ-calculus

As we mentioned earlier, λ-calculus is the basis of functional programming. Through the history of functional programming, we can see how from λ-calculus evolved to a family of modern functional programming languages that all have the characteristics of the calculus we discussed. For example, LISP, which is one of the first major programming languages was inspired by the λ-calculus. Many functional languages such as ML consist of little more than the λ-calculus with additional syntax. λ-calculus is important to functional programming languages and computer science. Through it variable binding and scoping in block structured languages can be modelled as well as several functions calling mechanism such as call-by-name, call-by-value and call-by-need. As discussed earlier, the λ-calculus is Turing universal, and probably the most natural model of computations and Church's Thesis asserts that the 'computable' functions are precisely those that can be represented in the λ-calculus.

The λ-calculus notions of confluence (Church Rosser property), termination and normal form, can be used as notions in rewriting theory. The λ-calculus and its extensions can be used to develop better type system, such as polymorphism, and to investigate theoretical issue such as program synthesis. The two main implementation methods , the SECD machines (for strict evaluation) and combinator reduction (lazy evaluation) exploit properties of

λ-calculus. SECD machine was invented by Landin as an interpreter (byte code interpreter) for the λ-calculus in order to execute ISWIM (If you See What I Mean) programs. ISWIM was the model for ML and it was designed to be extended with application-specific data and operations. It consisted of the λ-calculus plus a few more additional constructs and could be translated back into pure λ-calculus. Denotational semantics, which is an important method for formally specifying programming languages, employs the λ-calculus for its notation.

3.4 Denotational Semantics

Semantics is the assignment of meaning to the sentences of a programming language. Semantic definition methods are valuable to implementors and programmers for they provide a precise standard for a computer implementation, a useful user documentation and a tool for design and analysis. The standard guarantees that the language is implemented exactly the same on all machines. A formal semantic definitions can be read by a trained programmer and use it as a reference to answer subtle questions about the language. The semantics of programming languages is not as well developed as their syntax. This is because semantical features are much more difficult to define and describe and a standard method for writing semantics is still evolving. The first versions of programming language semantics used machines and their actions as their foundation. There are three main methods for semantics specification: operational, denotational and axiomatic semantics. In this thesis we will work with denotational semantics. Before giving a detailed definition of denotational semantics, we briefly give an overview of the other forms of semantics.

Operational semantics method uses an interpreter to define a language where the meaning of a program is the evaluation history (a sequence of internal interpreter configurations) that the interpreter produces. One of the disadvantage of this semantic is that there is no machine-independent definition exists because the language can only be understood in terms of interpreter configurations. Furthermore, if the interpreter's algorithm is simple and written in an elegant notation, the interpreter can give an insight of the language, but unfortunately, interpreters for nontrivial languages are large and complex, and the notation used to write them is often as complex as the language being defined.

In axiomatics semantics, the properties of a language are expressed with axioms and rules to construct a formal proof of the property. The character of an axiomatic definition is determined by the kind of properties that can be proved. The meaning of the program is not explicitly given at all with the axiomatic semantics method. For example, a very simple system may only allow proofs that one program is equal to another and a more complex system allows proofs about a program's input and output properties. Axiomatic definitions are more abstract than denotational and operational semantics method. The properties proved about a program may not be enough to completely determine the program's meaning [All87]. The format in axiomatic semantics is best used to provide preliminary specifications for a language or to give documentation about properties that are of interest to the users of the language.

Denotational semantics is an approach to formalizing the semantics of computer systems by constructing a mathematical object which expresses the semantics of these systems. The mathematical objects are called denotations or meanings. Further elaboration of the denotational semantics will be discussed in the next section.

Each of the three methods of formal semantics definition has different areas of application, and together they provide a set of tools for language development. Designers of a new programming system might first supply a list of properties that they wish the system would have. Since a user interacts with the system through an input language, an axiomatic definition is constructed in defining the input language and how it achieves the desired properties. Then a denotational semantics is defined to give the meaning of the language where a formal proof is constructed to show that the semantics contain the properties that the axiomatic definition specifies. Finally the denotational definition is implemented using an operational semantics.

3.4.1 Definition of Denotational Semantics

Denotational semantics has traditionally been described as the theory of true meanings for programs, or as the theory of what programs denote. The denotation is usually a mathematical value such as a number or a function and a valuation function maps a program directly to its meaning. A denotational definition is more abstract than an operational definition, for it does not specify computation steps.

Denotational semantics originated in the work of Christopher Strachey and Dana Scott in the 1960s. Denotational semantics originally developed by Strachey and Scott interpreted the denotation (meaning) of a computer program as a function that mapped input to output. But for programs that included elements such as recursively defined functions and data structures, the definition of denotation is limited. To overcome this, Scott introduced a generalized approach to denotational semantics based on domains [SS71].

An effort to address difficulties with the semantics of concurrent system, researches later on introduced approaches based on power domains. An alternative view point for denotational semantics is that it is seen as a translation from one formal system to another. However, the pragmatics of denotational semantics is essentially unaffected by the foundational stance one takes. The aims, hopes, and concrete uses of denotational semantics are the same. We can say that the purpose of denotational semantics are to bring out subtle issues in language design, to derive new reasoning principles, and to develop an intuitive abstract model of the programming language under consideration so as to aid program development.

3.4.2 Semantic Algebra

Before studying the semantics of programming languages, we must establish a suitable collection of meanings for programs. For this we need the notion of a semantic algebra. A semantic algebra is given by a semantic domain and a set of operations defined on elements of the domain. Semantic domains are the sets that are used as value spaces in programming language semantics. In practice not all of the set and set building operations are needed for building domains. The set of operations are functions that map elements from the domain to other elements of the domain. Operations are defined in two parts: first the functionality of the operations is defined, then the description of the operation's mapping is given. The functionality of an operation is given by the operation's domain and codomain. For an operation f, its functionality $f : D_1 \times D_2 \times \ldots \times D_n \rightarrow A$ says that f needs argument from domain D_1, D_2 until D_n to produce an answer in domain A. The description of the operation's mapping is usually an equational definition but a set graph, table or diagram may be used as well.

A primitive domain is a set that is fundamental to the application being studied and its elements are atomic and are used as answers or semantic outputs. For example the natural numbers:

- Domain Nat = N
- Operations

zero	:	Nat
one	:	Nat
two	:	Nat
...		
add	:	Nat × Nat → Nat
subtract	:	Nat × Nat → Nat
multiply	:	Nat × Nat → Nat
div	:	Nat × Nat → Nat

Note that constants (here zero, one, two, . . .) are treated as functions with zero arguments, and zero, one, two, . . . return the usual natural numbers. The operations add, subtract and multiply are addition, subtraction and multiplication of natural numbers, respectively and they are written in infix format. Natural number subtraction needs to be clarified further: if the second argument is larger than the first, the result is zero, otherwise normal subtraction is applied. add, multiply are defined as usual. By using the algebra, we can construct expression that represent members of Nat. An example is as follows:

(two multiply five) subtract (one add three)

This expression computes as follows:

(two multiply five) subtract (one add three)
= (two multiply five) subtract four
= ten subtract four
= six

Other examples of primitive domains are truth values (Boolean -B), character strings (C) and etc. Compound domains are domain building constructions for creating new domains from existing ones. The four basic constructions of forming compound domains from semantic domains A and B are :

- The product domain A×B has as members ordered pairs of the form (a, b), for ($a \in$ A and $b \in$ B).

- Sum domains A + B has as members elements from A and B, labeled to mark their origins. The classic representation of this labeling is the ordered pair (zero, a) for an $a \in$ A and (one, b) for a $b \in$ B.
- The members of the function domain A $\rightarrow$ B is the collection of functions from domain A to domain B.
- The lifted domains $\mathrm{A}_\perp$, has members $\mathrm{A}_\perp \overset{def}{=} \mathrm{A} \cup \{\perp\}$. '$\perp$' denotes an undefined element (often standing for nontermination) or 'no value at all'. If one wants to introduce a function f, which applied to an argument $a \in$ A may yield an element in B or no answer at all, then we can introduce f as having functionality A $\rightarrow$ $\mathrm{B}_\perp$. Then $f(a) = \perp$ means that $f(a)$ is undefined.

Including $\perp$ as a value is an alternative to using a theory of partial functions. A partial function is a function that may not have a value associated with each argument in its domain.

3.4.3 Denotational Definition

A denotational definition of a language consists of three parts i.e the abstract syntax definition of the language, the semantic algebra and the valuation function. The valuation function is actually a collection of functions, one for each syntax domain. A valuation function **D** for a syntax domain D is listed as a set of equations, one per option in the corresponding BNF rule for D. For example, the denotational definition of binary numerals are shown in Figure 3.1.

In the algebra only multiply and add are listed because the others are not used in the valuation functions. From the denotational definition in Figure 3.1, we can determine the meaning of the binary numeral $[\![101]\!]$ as follows:

$$\begin{array}{rcl}
\mathbf{B}(101) & = & (\mathbf{B}(10) \text{ multiply two}) \text{ add } \mathbf{D}(1) \\
 & = & (((\mathbf{B}(1)\text{multiply two}) \text{ add } \mathbf{D}(0))\text{multiply two})\text{add } \mathbf{D}(1) \\
 & = & (((\mathbf{D}(1) \text{ multiply two}) \text{ add } \mathbf{D}(0)) \text{ multiply two}) \text{ add } \mathbf{D}(1) \\
 & = & (((\text{one multiply two}) \text{ add zero}) \text{ multiply two}) \text{ add one} \\
 & = & \text{five}
\end{array}$$

Thus we can see that the meaning of the binary numeral 101 from the derivation tree is five.

We make use of denotational semantics in the proof shown in Chapter 6 where we give the denotational semantics for the functional programs and for the object-oriented programs. Then we show that the semantics of the functional programs and of the programs obtained from translating them into object-oriented program coincide. The denotational semantics of the functional program is constructed based on the abstract syntax of the simply typed λ-calculus shown in the section 6.2.4.

- Abstract syntax:

  B ∈ Binary-numeral
  D ∈ Binary-digit
  B ::= BD | D
  D ::= 0 | 1

  The notation D := 0 | 1 means D := $\{0, 1\}$ and the elements of B are either elements of D or an element b ∈ B followed by an element d ∈ D written as *bd*. Thus, for instance 101 ∈ B which is obtained by having: 1 ∈ D, so 1 ∈ B, 0 ∈ D, so 10 ∈ B, 1 ∈ D so 101 ∈ B. One then writes in the following B for elements of B and D for elements of D, so B(BD) stands for an element of B applied to the result concatenating an element b of B to an element d of D.
- Semantic Algebra

  I. Natural numbers
  Domain Nat = N
  Operations
  zero, one, two, . . . : Nat
  add, multiply : Nat × Nat → Nat
- Valuation Functions :

  B : Binary-numeral → Nat
  B(BD) = (**B**(B) multiply two) add **D**(D)
  B(D) = **D**(D)

  D : Binary-digit → Nat
  D(0) = zero
  D(1) = one

  The operation multiply and add are written in infix format.

Figure 3.1: Denotational definition of binary numeral

Chapter 4

Integrating Functional Programming into C++

C++ is a general purpose programming language which supports object oriented programming as well as procedural and generic programming. It is a paradigm-neutral language [GJ98]. Unfortunately, C++ does not support functional programming which can give great benefits in developing a program especially in order to create mathematical functions. As discussed in the previous chapter, functional programming have several features that made it practical such as first class values, high-order functions, lazy evaluation and other features that are usually absent from imperative languages. By integrating functional programming into C++, the advantages of object oriented programming and functional programming can combine making C++ a more powerful language.

We are using C++ code itself in order to integrate functional programming into C++. More precisely we have written a C++ program, which parses λ-terms, which are given in a specific syntax, and translates them into their equivalent C++ statements. This is an important step towards embedding functional programming into C++, since the λ-calculus is the basis of functional programming.

In this chapter we will discuss the approach that we use in integrating functional programming into C++ and the design, specification and development of the program that parses and translates λ-terms into equivalent C++ code.

4.1 Integration of Functional Programming into C++

Even though there are several approaches to integrate functional programming into C++ such as creating a special library for functional programming(FC++) [MS00], our approach has the advantage that it is simple and allows for a correctness proof. We also believe that it is more flexible, since it allows for example λ-terms with side effects. Other approaches will be discussed further in Chapter 7.

The translated code is produced based on the related idea discovered by Kiselyov [Kis98] and Läufer [Lau95] that can be used for functional programming by representing higher order functions using classes. The C++ code that is generated for simply typed λ-terms uses the object-oriented concepts of classes and inheritance. Abstract classes are used in defining the function type of a λ-term with a virtual operator that is overloaded in the definition of the λ-term. The type itself is the type of pointers to an object of this abstract class. The concept of inheritance is involved in the definition of a λ-term where the function type abstract class will be the base class for the λ-term. More details will be discussed in the next chapter.

In its most pure form, functional programs contain no side effect at all [Hug89], (Note that many functional programming languages such as ML allow side effects). Programs with no side effect will lessen the burden of debugging and maintaining the program and also hinder any accidental side effects that might occur during development. Our translated λ-term follow this: the translated code has no assignment statement. The evaluation of the translated λ-term corresponds to call-by-value evaluation. Call-by-value evaluation has been discussed earlier in the Chapter 3.

In C++ there are two ways of passing arguments in a function i.e. through call-by-value and call-by-reference [Eti94]. An argument passed to function using call-by-value will not be changed by the function (eventhough changes to the argument are made in the function) because a copy of the value is made and passed to it. Any change to the copy does not affect the value of the original argument. In case of call-by-reference arguments, the caller gives the called function the ability to access the caller's data directly, and if any changes or modification to the data will affect it directly. Arguments or parameters that are passed by reference in C++ make use of the symbol '&' as a flag for reference. For example, the declaration of the function header `f` with reference parameters: `f(int &x)` where `x` is a reference to an `int`. A reference argument must be an *lvalue*, not a constant or expression that returns an *rvalue*. For example, the call `f(t)` is only allowed if `t` is a variable and `x` is a reference to `t`. Whatever happens to `t` happens to `x` as well. Evaluation for call-by-reference is not a problem because a variable is already evaluated (a variable contains a value).

Generally, for reasons of clarity and performance, many C++ programmers prefer that modifiable arguments be passed to functions by using pointers, small nonmodifiable arguments be passed call-by-value and large nonmodifiable arguments be passed to functions by using references to constants. The reference parameters can be used with a `const` to prevent their values being modified. The `const` keyword can be used in several ways to prevent values of arguments being changed. For example, the previous example is changed using the `const` as follows:

```
void f(const int &x){
x = 1;
}
```

This code will not compile since we cannot change a `const` variable. The use of `const` with reference parameters will cause the parameters be passed without copying (in case of large data will waste too much memory or take too long) but stop it from being altered or changed. Passing large objects such as structures using pointers to constant data, or references to constant data will obtain the performance benefits of call-by-reference and the security of call-by-value.

In call-by-name evaluation, the arguments to a function are not evaluated at all, but are substituted directly into the function body using capture-avoiding substitution. If the argument is not used in the evaluation of the function, it is never evaluated but if it is used several times, it is reevaluated each time. For example, a function :

```
int f(int x){
return x + x;
}
```

and a call `f(t)` will have the effect of computing `t` to a value `n1` and computing `t` to a value `n2`, then `n1 + n2` is computed. This involves in computing `t` twice since its value is needed twice. Note that C++ has no call-by-name evaluation, and the example given is a code in C++ syntax but for a language which has a call-by-name evaluation. Call-by-name evaluation is rarely implemented directly, but frequently used in considering theoretical properties of programs and programming languages. Thus, real-world languages with call-by-name semantics tend to be implemented using call-by-need.

Call-by-need is a memoized version of call-by-name where, if the function argument is evaluated, that value is stored for subsequent uses. For example, the function given previously (for call by name), if the call `f(t)` is call-by-need, `t` is evaluated once since the evaluated version is used for the second use of `t`. In a "pure" (effect-free) setting, this produces the same result as call-by-name. But when the function argument is used two or more times, call-by-need is always faster. Sometimes evaluation of expressions may happen arbitrarily far into computation and due to this languages using call-by-need generally do not support computational effects (such as mutation) except through the use of monads. This eliminates any unexpected behaviour from variables whose values change prior to their delayed evaluation. Lazy evaluation or delayed evaluation is the technique of delaying a computation until such time as the result of the computation is known to be needed. Lazy evaluation also means evaluation is done only once. Most realistic lazy languages such as Haskell use call by need for performance reason.

By embedding λ-calculus into C++, the task of creating a function especially, a mathematical function, will be simpler. As we know in C++, in order to create a function, we must name the function, declare and define it before using it (calling it). But by using the syntax that has been determined to embed λ-calculus, we can have a nameless function and we can omit the extra work of declaring and defining it. Thus, we can have an option of creating a function on the fly even though a named function is encouraged for documentation purposes. We use the variant of the Church style typed λ-calculus discussed at the end of the previous chapter, except that we have constants for arithmetic functions, rather than constants for object of arbitrary types. The λ-calculus was introduced at page 35 and the typing rules for a λ-term is listed in the Chapter 6.

A λ-term $\lambda x^{\text{int}}.t$ where t is of the type `int` will be written in our syntax as `\int x.int t` where the function type is int→int. The reason why we type `int` to t will be explained in the Section 4.4.1. More details on how the function type is determined are discussed in Chapter 5. We have said previously that, creating a function using the concept of λ calculus will rid the task of defining and declaring the function. For example, for creating a function that squares any integer number in C++ , we need to give the function a name, declare and define it as follows:

```
\\declare the square function by giving its prototype
int square(int);

\\define the function
int square(int no){
    return no*no;}

\\calling the function
int n = 50;
int square_number = square(n);
```

But by implementing the λ-calculus, we can create the function above and apply it to the variable n in one expression which is shown as follows:

```
int n;
int square_number =(\int no.int no*no)^^n;
```

We introduce the symbol `"^^"` for the term application. In the expression above, the λ-abstraction is applied to variable n. In this expression the λ-abstraction is reduced to `n*n` and if `n = 2`, the variable `square_number` will have the value 4. This is called β-reduction. The λ-calculus has only functions with one argument. Functions with more than one arguments can be expressed with a function whose result is another function and this kind of function is known as a *curried function*. Curried functions are functions that are represented using nested lambdas. This technique has its name from Haskell B. Curry. An example of curried function written in our syntax is as follows :

```
\int x.\int y.\int z.int x+y+z;
```

When the λ-abstraction above is applied to 3, 4, and 5, it will perform three β-reductions resulting in the value 12. The result is obtained by replacing x with 3, y with 4 and z with 5. The application mentioned can be written as follows:

```
(((\int x.\int y.\int z.int x+y+z)^^3)^^4)^^5
```

This term which is written in our syntax, will then be translated into its equivalent C++ code. In the following sections we will discuss the design, specification and the development of the Parser-Translator program. We will also show some examples of λ-terms that we have tested using our program.

4.2 Overview of the Parser-Translator Program

The Parser-Translator program or PTP was written in C++ using Spirit to generate a parser that parses λ-term based on the grammar that has been determined. The PTP was compiled and executed using the C++ compiler with Boost libraries. The Boost libraries work only with modern C++ compilers which support modern C++ features such as templates and the C++ Standard Library.

Spirit is part of the C++ Boost libraries [Bo02]. It is an object-oriented recursive-descent parser generator framework which was implemented using template meta-programming techniques where expression templates enable us to approximate the syntax of Extended Backus-Normal Form (EBNF) completely in C++. It enables a target grammar to be written exclusively in C++ where it can mix freely with other C++ code and the grammar is immediately executable i.e. the inline EBNF grammar specifications do not need to undergo the step of translation from the source EBNF code to C++ code making Spirit the best choice to be used for developing the PTP.

There are several files that are involved in the PTP. The overview of the files and the flow of data is shown in the Figure 4.1.

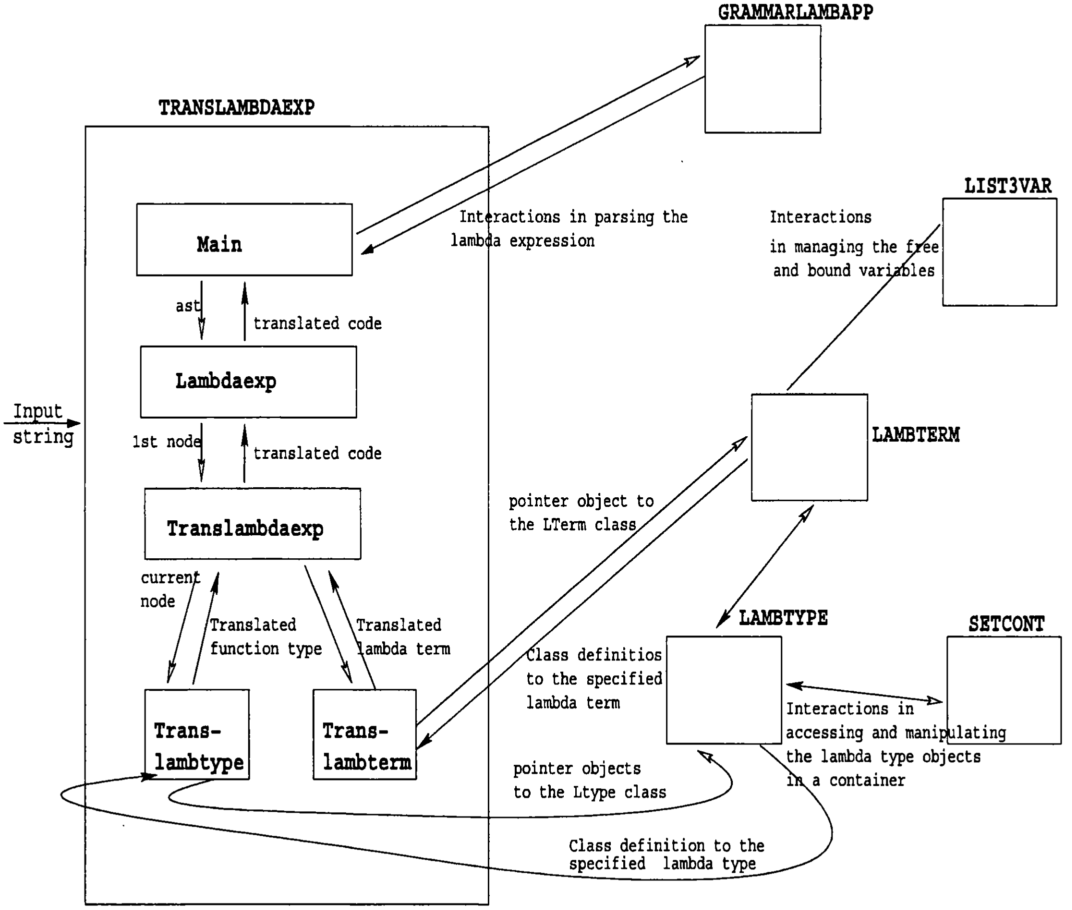
GRAMMARLAMBAPP
TRANSLAMBDAEXP
LIST3VAR
Main
Interactions in parsing the lambda expression
Interactions in managing the free and bound variables
ast
translated code
Lambdaexp
Input string
1st node
translated code
LAMBTERM
pointer object to the LTerm class
Translambdaexp
current node
Translated function type
Translated lambda term
LAMBTYPE
SETCONT
Class definitios to the specified lambda term
Interactions in accessing and manipulating the lambda type objects in a container
Trans-lambtype
Trans-lambterm
pointer objects to the Ltype class
Class definition to the specified lambda type

Figure 4.1: Overview of the the files involved in the PTP

The files in the PTP can be divided into two parts i.e, the parsing and the translating part. These files contain modules that execute certain tasks. `Translammbdaexp` is the main file which contains the main module for the PTP where control of the program is executed. The input string of the λ-expression is entered to the main module where the input will undergo the parsing phase which involves the `grammarlambdaexp` file. If the parsing succeeds, the input will pass the translation part/phase. In this part, the modules in the files `listvar`, `Setcont`, `Lambtype`, and `Lambterm` will go into action and the translated λ-term will be the output.

The grammar rules and the constructor of the classes in the PTP is based on the concept of typed and untyped λ-term as explained in the previous chapter. This concept is depicted in the Figure 4.2 by giving an example of a λ expression.

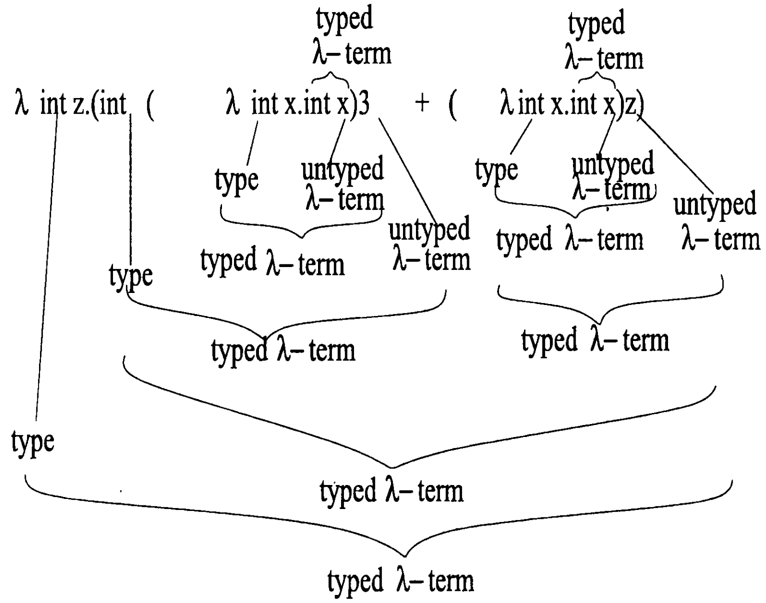

Figure 4.2: Depiction of the concept of the typed and untyped λ-term

4.3 Description of the Modules in the Parsing Phase

The input string of λ-expression is entered through the main module in the `Translambdaexp` file, where the string is parsed based on the grammar rules in the file `Grammarlambdaexp`. The parsing is done here to ensure that the input is written according to the syntax that has been determined. If the parsing succeeds i.e. the input matches with the λ-term grammar rules, an abstract syntax tree (ast) is generated. We will not discuss the parsing process here because details of it are discussed in the next chapter. Here we will discuss the formation of the grammar rules of the λ-term.

EBNF for the production rules in Figure 4.3, and Figure 4.4:

```
<lambstmt>        -> (<lambtype>|<nativetype>) ' ' <identifier>
                     ' ' '=' <lambexp> ;
<lambexp>         -> (<lambdaterm>|<untypedlamterm>)
<lambdaterm>      -> (<lambabstract>|<lambapp>)
<lambabstract>    -> \ (<lambtype>|<nativetype>)  ' ' <identifier>
                     '.' <lambabstract>|<lambtype>|<nativetype>)
                     ' ' <untypedlamterm>
<lambapp>         -> '(' <lambabstract> ')' '^^'
                     (<lambapp>|<digit>|<identifier>)
<untypedlamterm> -> (<digit>|<identifier>|<lambdaterm>)
                     *{(<infixoperator>|'^^')
```

```
                      (<untypedlamterm>|<lambdaterm>)}
                      | <identifier> '('
      (<untypedlamterm>|<lambdaterm>)
                        *{ ',' (<untypedlamterm>|<lambdaterm>)}
                        ')'

<infixoperator>     -> '='|'-'|'/'|'*'
<digit>             -> +{{0|1|2|3|4|5|6|7|8|9}}
<btype>             -> (nativetype|lambtype
                        |'(' <btype> ')')
<nativetype>        -> ('int'|'char'|'string'|'double'
                        |'float'|'long'|'short'
                        |'bool'|'signed'|'unsigned')
<nondigit>          -> ('-'|'a'|'b'|'c'|'d'|'e'|..|'z'
                        |'A'|'B'|'C'|'D'|'E'|'F'..|'Z')
<lambtype>          -> *{<btype> '->' } <btype>
<identifier>        -> <nondigit> *{(<nondigit>|<digit>)}
```

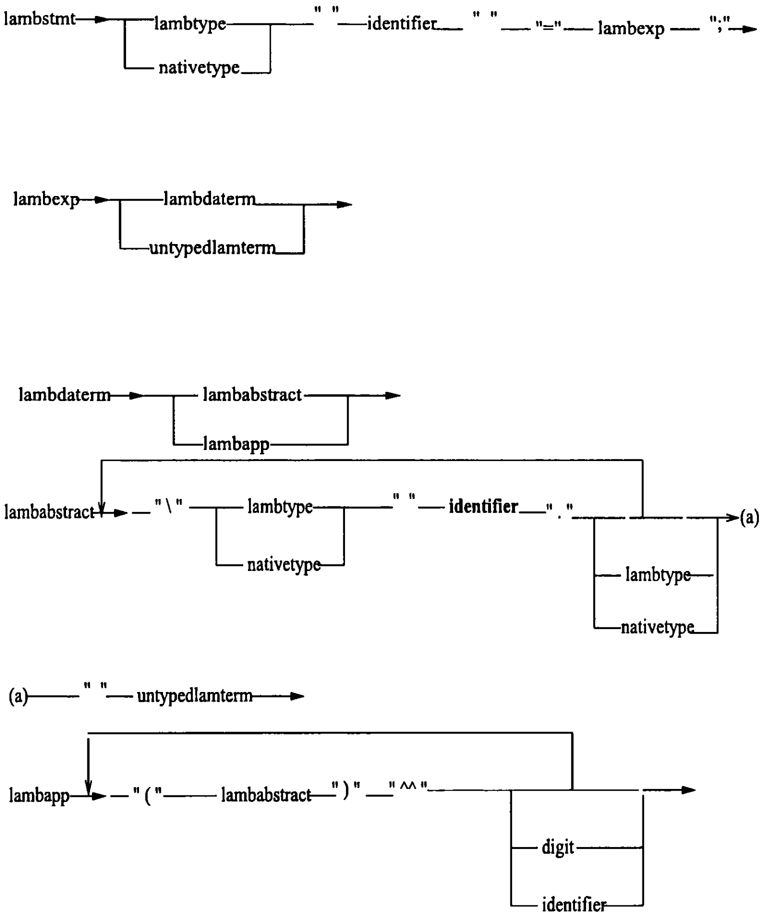

Figure 4.3: Syntax diagram of the λ-term grammar

From the syntax diagram shown in Figure 4.3, we can see that the λ-expression can be a typed λ term or an untyped one. The grammar for the λ-term is divided into λ-abstraction and λ-application. The grammar rule for the lambdaterm is for the building of λ-abstraction and application while the rule for untypedlamterm (Figure 4.4) is for the body of the λ-term or a standalone untyped λ-term. Other variables that made up the grammar of a λ-term are shown in the Figure 4.4. Note that in the syntax diagram, the symbol '*' and '+' is the indication that whatever is associated with them will be repeated zero or many times ('*') and one or more times ('+'). To assist in understanding the grammar given in the syntax diagram, we give the Extended Backus Naur Form (EBNF) for the grammar. We will not go through the precise syntax of the λ-term because we can see it clearly from the syntax diagram. The complete grammar of the λ-term written in Spirit are enclosed in the appendix. We only would like to mention some of the directives and predefined parser in Spirit that we used in the grammar rules such as leaf_node_d, root_node_d, digit_p, alpha_p and etc. In the Figure 4.4, the directives digit_p and alpha_b is used instead in the production rule for digit and nondigit. digit_p recognizes the digits from 1 to 0 and alpha_b recognizes all the characters in the alphabet whether in lower or upper case. Since we are building an abstract syntax tree, directives like leaf_node_d, root_node_d, no_node_d and inner_node_d are beneficial in simplifying the structure of the tree thus making the traversing and processing of the abstract syntax tree formed more easier. Usually every character in a string will be taken as a node in a tree, but leaf_node_d will take all the characters it is formed from as one node – this construct will for instance be used for identifiers. Other directives will be discussed and examples for their usage in building the abstract syntax tree will be given in the next chapter. Every token in the grammar rule is given an identification (id) which is an integer value. This id is used to identify each node in the abstract syntax tree.

4.4 Description of Modules in the Translation Phase

The abstract syntax tree generated will be passed to the module lambdaexp where this module will pass the beginning node of the tree to the module translambaterm to be processed. Here every node including the children node will be processed until the end node of the tree. The tree will be traversed from the beginning node to the end node using the tree iterator which is a special facility from Spirit. Every node and its children are tested for their token id in the grammar rule. When the node is identified, specific module in the file Translambdaexp is called which in turn will call the module in the file Lambterm or Lambtype to translate it to its equivalent C++ code.

The file Lambtype consists of lambda type class (LType) with constructors for function type and native type, and methods for generating the C++ code for the function type and native type. For example if the node is a lambtype or a nativetype, the module translambdatype is called which in turn will instantiate the constructor for the function type or native type in the file Lambtype and execute the appropriate method to translate it to its equivalent C++ code.

The file Setcont is associated with files Lambtype, Listvar and Lambterm. The involvement of these files are discussed in the coming sections.

4.4.1 Description of the Lambterm File and its Associated Files

The file `Lambterm` contains the declaration and the definition of the typed (`LTerm`) and untyped λ-term (`UntypedLTerm`) objects along with the definition of their methods. As mentioned PTP is developed using the object-oriented approach that involves inheritance where the typed λ-term form a subclass of the untyped λ-term. Constructors are built based on all possible terms that can occur for the class and subclass and a method is defined to create a pointer to each constructor. There is a special constructor in `LTerm` class that gives a type to an untyped λ-term where a method called `Lift` creates a pointer to this constructor. Thus any untyped λ-term will become typed using this method.

The `Listvar` file manages the bound and free variables of the λ-term. It consists of a class `Listvariable` with constructors for empty list of variables and adding list of variables. The class `Listvariable` has methods that are responsible in displaying arguments for the λ-term and also a method that will not allow the same variable name to be listed as arguments for the λ-term which indirectly minimizes or disallow aliasing.

The methods of the class of typed λ-term are for the purpose of translating the typed λ-term to its equivalent C++ code. The same thing applies to the methods of the class of untyped λ-term. The translated code produced uses the object-oriented programming technology. A class is created for each λ-term and the translated λ-term is by inheritance an element of the translated function type. Here the function type is an abstract class which is the base class for the class of λ-terms of this type.

The type checking in the translated code is done by the type system of C++. The type system of C++ has decidable type checking not type inference like in Haskell. In C++ we check whether a term t has a certain type but not type inferencing because a term t can have multiple types. For example a term λx^{int}. 3.14 can be of type int→float or int→double and by inheritance a term can be an element of many types. In the translation we added the type to the body of the term because we need to know the type of the body of the λ-term. But if we apply a λ-term t to $s(s\ t)$ and we know the type of s; $t : \sigma \rightarrow \tau$ then $s : \sigma$, we do not demand to assign a type to t and in some examples we can even omit the type.

Why do we need to use inheritance in the translation? To explain this, consider the terms given as follows:

$$g := \lambda x.x \quad : \text{int} \rightarrow \text{int}$$

$$g' := \lambda x.x + x \ : \text{int} \rightarrow \text{int}$$

$$h : (\text{int} \rightarrow \text{int}) \rightarrow \text{int}$$

$$h(f) = f(5)$$

Given the expression: $h(g) + h(g')$, without using inheritance, the left h needs to use the class defining g and the right h needs to use the class defining g'. Assuming g is defined by class `lambda1` and g' is defined by class `lambda2` and h has methods as follows:

```
int operator() (lambda1 g){
return g(5);
}
int operator() (lambda2 g) {
```

```
return g(5);
}
```

In general, it is not possible to predict all possible applications of h since arguments might be defined dynamically. One might suggest that h should have templated method of the form:

```
template operator() <A> (A g){
return g(5);
}
```

But this can only work if we know at compile time the g's to which h is applied. Thus the use of inheritance seems to be the only type method which works in general.

The translation of a λ-term will create various classes. The general form of the translated code is as follows:

```
[ Classes of the function types are defined here
                 :  :
                 :  :
]
{Classes of the lambda-term are defined here
                 :  :
}

[Lambda expression is written here ]
```

If the λ-term is a simple λ-abstraction, the class defined is just a single class that has a function type as the base class. As an example we give the class definitions for the λ-term $\lambda x^{\text{int}}.x + 5$. The function type is int→int which is translated as follows:

```
class Cint_intD_aux
{ public  : virtual int operator() (int x) = 0;};
typedef Cint_intD_aux* Cint_intD;
```

The class definition for the λ-term is as follows:

```
class lambda0 : public Cint_intD_aux{
 public:
 lambda0() {};
 virtual int operator () {int x}
  { return x + 5;};
};
```

The λ-term itself is translated into `new lambda0()`. We are aware that the use of new is expensive. In many simple examples, one could avoid the use of new by replacing pointers to objects by objects. For instance, we could replace a pointer to an object of the class representing a λ-term by the object of this class itself, and then would not need to generate the object dynamically. However we do not know how to deal with the general situation. In general, it seems that we need inheritance. For instance, the C++ class representing

`(\(int->int) f.int f)^^0` could without inheritance only applied to a λ-term of type (int→int) which is translated into one particular object, and therefore not be applied to an arbitrary element of type (int→int). One could create several instances of this method to cater for different objects representing different λ-terms which are all of type (int→int), but only if those objects are known at compile time. If we generate those λ-terms dynamically at run time, then this is no longer possible. So, in the general situation we require the use of inheritance, although many special cases would be optimized.

If the term is a curried function or a nested λ-term, a series of class definition and function type will be generated. The name of the class is automatically generated: it starts with `lambda` followed by an integer that corresponds to the sequence of classes generated. If a curried function or a nested λ-abstraction that involves three arguments like the example given previously is translated, three classes will be created and the name of the class will be `lambda0`, `lambda1` and `lambda2`. Details of the translation is discussed in the next chapter. Here we only give some examples.

The statements declaring λ-terms that will be accepted by our parser have the following form:

```
(nativetype|lambdatype) identifier "=" (lambdaterm
                                        | untyped lambdaterm);
```

`nativetype` refers to tha native or basic type in C++ such as `int`, `char` and `double`, while `lambdatype` refers to the function type A→B where A is the input type and B is the result type of the function. The general processing of the abstract syntax tree is shown in the pseudocode below:

```
1. Begin with the 1st node
2. Execute the module for translation of the type whether it's
   a native type or a lambda type to the children of the node
3. string1 = Translation of type
4. Next node (identifier)
5. string2 = identifier
6. Skip node (for '=')
7. Test the next node
   a. if = lambda term
            Execute the module for translation of the lambda
   term to the children of the node
   string3 = class definition of the lambda term and its
   function type
   string4 = expression of the lambda term
   b. if = untyped lambda term
            Execute the module for translation of the untyped
   lambda term to the children of the node
   string3 = translation of the untyped lambda term
   string4 = expression of the untyped lambda term
8. Output string3 + string1  + string2 + '=' + string4
9. End
```

In step 2, the children of the node is passed to get the object of the type pointer which points to the appropriate constructor of the class λ-type (`LType`). This object pointer then invoke the method that creates the classes of the function type. The class `LType` and its methods are in the file `Lambtype`. An associative container is applied to manage the sequence of λ-type objects. We make use of it because we can order the λ-type objects in a container following a sorting criterion that is predefined in the program and also due to its iterator that offer a common interface for any arbitrary container type. The iterator makes it possible for us to avoid any duplication of the λ-type objects. The container mentioned is defined in the file `Setcont`. Once the object pointer of `LType` is determined, it is checked in the container using the iterator whether it exists or not. If it exists it will be discarded, otherwise it will be added to the container. Then the class definition of the function type will be created where the sequence of class definition of λ-types are based on the λ-type objects in the container.

In step 7, the children of the node is passed to the end to get the object of type pointer that points to the constructor of the class `LTerm` where this object pointer will invoke the appropriate methods to generate the class definitions and expression of the λ-term. Similarly for the untyped λ-term, the same process will be executed to get the class definition and expression for the untyped λ-term. The modules responsible for these tasks reside in the file `Lambterm` and these modules will invoke the modules in the file `Listvar` to manage the bound and free variables of the λ-term. The modules in the file `Lambtype` are also invoked to get the class definition of the function type. The string of class definition of function type and class definition of a term as well as the expression of the λ-term are each assigned to a string variable which are concatenated to produce the whole completed translation.

4.5 Examples of the translation of λ-term expressions

The Parser-Translator program was tested with several forms of λ-term and the translated code was compiled and run. The result then was compared with the result that we got manually. Here we will give some examples of λ-terms that were tested and their translated code.

1) input :

```
int->int f =\int y.int (\int x.int x*x)^^3
             + (\int x.int x*x)^^3 +y;
```

and the translated code is:

```
class Cint_intD_aux
{
 public : virtual int operator() (int x) = 0; };

typedef Cint_intD_aux*  Cint_intD;

class lambda1 : public Cint_intD_aux{
 public :
 lambda1( ) { };
 virtual int operator () (int x)
 { return x*x; };
};

class lambda0 : public Cint_intD_aux{
 public :
 lambda0( ) { };
 virtual int operator () (int y)
 { return (*( new lambda1( )))(3) +
           (*( new lambda1( )))(3)+y; };
};
Cint_intD f = new lambda0( );
```

We can apply this term to an integer value which is shown as follows :

```
int g = (*f)(4);
```

and the result is the value 22.

2) Input:

```
int g =(\int->int f.int (*(f))((*(f))(2)))
                         ^^(\int x.int 2+x);
```

The translated code is as follows:

```
class Cint_intD_aux
{
 public : virtual int operator() (int x) = 0; };

typedef Cint_intD_aux*  Cint_intD;

//Definition of type : ((int->int)->int)
class CCint_intD_intD_aux
{
 public : virtual int operator() (Cint_intD x) = 0; };

typedef CCint_intD_intD_aux*  CCint_intD_intD;

class lambda0 : public CCint_intD_intD_aux{
 public :
 lambda0( ) { };
 virtual int operator () (Cint_intD f)
 { return (*(f))((*(f))(2)); };
};

class lambda1 : public Cint_intD_aux{
 public :
 lambda1( ) { };
 virtual int operator () (int x)
 { return 2 + x; };
};
int y = (*( new lambda0( )))( new lambda1( ));
```

The λ-expression in the second example can be written in λ-notation as :

$$\lambda f.f(f\ 2)(\lambda x.2 + x)$$

and it can be evaluated as:

$$(\lambda x.2 + x)((\lambda x.2 + x)2) = (\lambda x.2 + x)4$$

resulting in the value 6.

3) Input:

```
int k =(((\((int->int)->(int->int)) g.\int->int f.
               int->int g^^g^^f)
          ^^(\int->int f.\int x.int f^^f^^x))
          ^^(\int x.int 2+x))^^3;
```

and the translated code is:

```
class Cint_intD_aux
{
 public : virtual int operator() (int x) = 0; };

typedef Cint_intD_aux*  Cint_intD;

//Definition of type : ((int->int)->(int->int))
class CCint_intD_Cint_intDD_aux
{
 public : virtual Cint_intD operator() (Cint_intD x) = 0; };

typedef CCint_intD_Cint_intDD_aux*  CCint_intD_Cint_intDD;

//Definition of type : (((int->int)->(int->int))
//                             ->((int->int)->(int->int)))
class CCCint_intD_Cint_intDD_CCint_intD_Cint_intDDD_aux
{
 public : virtual CCint_intD_Cint_intDD operator()
                  (CCint_intD_Cint_intDD x) = 0; };

typedef CCCint_intD_Cint_intDD_CCint_intD_Cint_intDDD_aux*
          CCCint_intD_Cint_intDD_CCint_intD_Cint_intDDD;

class lambda1 : public CCint_intD_Cint_intDD_aux{
 public :CCint_intD_Cint_intDD g;
 lambda1( CCint_intD_Cint_intDD g)  {   this-> g = g;};
 virtual Cint_intD operator () (Cint_intD f)
 { return (*(g))((*(g))(f)); };
};

class lambda0 : public
                CCCint_intD_Cint_intDD_CCint_intD
```

```
        _Cint_intDDD_aux{
 public :
 lambda0( ) { };
 virtual CCint_intD_Cint_intDD operator ()
                               (CCint_intD_Cint_intDD g)
 { return new  lambda1( g); }
};

class lambda3 : public Cint_intD_aux{
 public :Cint_intD f;
 lambda3( Cint_intD f)  {   this-> f = f;};
 virtual int operator () (int x)
 { return (*(f))((*(f))(x)); };
};

class lambda2 : public CCint_intD_Cint_intDD_aux{
 public :
 lambda2( ) { };
 virtual Cint_intD operator () (Cint_intD f)
 { return new  lambda3( f); }
};

class lambda4 : public Cint_intD_aux{
 public :
 lambda4( ) { };
 virtual int operator () (int x)
 { return 2 + x; };
};
int k = (*((*((*( new lambda0( )))( new lambda2( ))))
     ( new lambda4( ))))(3);
```

In the above example, the function type for λ-abstraction represented by `lambda0` is (((int→int)→(int→int))→((int→int)→(int→int)) where the class definition for this function type is built from series of function types. The function type will not be duplicated even though we have other λ-abstractions in the expression of the same function type in the series of function types. This is the advantage of using a container for λ-type objects. We have tested this program with many more λ-expressions. Only a few of them are shown here. The implementation of the Parser-Translator Program in integrating functional programming is discussed in greater detail in the next chapter.

We have mentioned previously that lazy evaluation is one of the characteristics of a functional program. After introducing the extended syntax in defining a λ-term in C++, we can represent lazy evaluation in C++ by using the extended syntax. Further details will be discussed in the next section.

4.6 Lazy Evaluation in C++

We represent lazy evaluation in C++ by translating Haskell code using infinite list such as for computing Fibonaci numbers. This example [ABS06a], [ABS06b] requires that we have infinite streams of natural numbers and rely heavily on lazy evaluation. The standard technique for replacing call-by-value by call-by-name is to delay evaluation. The code in Haskell that will be translated into efficient C++ code is shown as follows:

```
fib = 1:1:(zipWith (+) fib (tail fib))
```

In order to delay evaluation, we replace types A by () → A where () is the empty type (i.e. `void`). Lazy evaluation not only delay evaluation, but it evaluates a term only once. So, to obtain this, we define a new type `Lazy(A)` which delays evaluation of an element of type A in such a way that evaluation will be carried out when needed, and it is done only once. Once the value is computed, the result is stored in a variable for later reuse. The definition for the class `lazy` is a general definition which is not restricted to lazy streams. We use the extended C++ syntax for λ-terms, λ-types and especially `r^^t` for application, \ for λ and `->` for → which has been discussed in Chapter 5 in the translated code. The definition of the class `lazy` is as follows:

```
template<typename X> class lazy{
  bool is_evaluated;
  union {X      result;
         () -> compute_function;};
 public:
   lazy(() -> X compute_function){
      is_evaluated = false;
      this->compute_function = compute_function;};
   X eval() {
     if (not is_evaluated){
         result = comput_function ^^ ();
 is_evaluated = true;};
 return result;};};
#define Lazy(X) lazy<X>*
```

The definition given would be much longer and considerably complicated without support from the extended syntax. Using the class `lazy` we can easily define lazy streams of natural numbers. Possibly terminating streams such as lazy list can be defined similarly but require the usual technique based on the composite design pattern for formalising algebraic data types as classes by introducing a main class for the main type which has subclasses for each constructor, each of which stores the arguments of the constructor.

```
template<typename X> class lazy_stream{
  public: Lazy(X) head;
          Lazy(lazy_stream<X>*) tail;
  ... Constructor as usual... }
#define Lazy_Stream(X) lazy_stream<X>*
```

An operation which takes a function of type () → X and returns the corresponding element of type Lazy(X) is defined as :

```
template<typename X> Lazy(X)
                     create_lazy(()-> X compute_function)
{    return new lazy<X>(compute_function);};
```

We need to define operators that will be used in the above definition of fib which is listed as follows:

- lazy_cons_lazy<X> computes the cons-operation on streams and returns lazily a lazy stream:

```
template<typename X>Lazy(Lazy_Stream(X))
           lazy_cons_lazy(Lazy(X) head,
              Lazy(Lazy_Stream(X)) tail){
                return create_Lazy
     (\ () X.new lazy_stream<X>(head,tail))};}
```

- lazy_tail<X> computes tail of a stream lazily where only its type is defined here:

```
Lazy(Lazy_Stream(X)) lazy_tail<X>(
                      Lazy(Lazy(Lazy_Stream(X)) s)
```

- lazy_zip_with<X> computes the usual zip_with function (i.e. zip_with(f, $[a, b, \ldots], [c, d, \ldots]) = [f\ a\ c,\ f\ b\ d, \ldots]$; we define only its type:

```
Lazy(Lazy_Stream(X)) lazy_zip_with<X>
     (X -> X -> X f,
      Lazy(Lazy_Stream(X)) s0,
      Lazy(Lazy_Stream(X)) s1)
```

The definition of lazy_tail and lazy_zip_with is straightforward, once one has introduced a few combinators which deals with Lazy(X). After introducing the operators that are involved in defining Fibonacci numbers, the stream of Fibonacci numbers is defined as follows:

```
()-><Lazy_Stream(int)> fib_aux =
 \() x.Lazy_Stream(int)
       eval(
       lazy_cons_lazy(
        one_lazy,
        lazy_cons_lazy(
         one_lazy,
         lazy_zip_with(
          plus,
           create_lazy(this),
           lazy_tail(create_lazy(this))))));
```

```
  Lazy_Stream(int) fib = eval(create_lazy(fib_aux));
```

`plus` is $\lambda x, y.x+y$, `one_lazy` is the numeral 1 converted into an element of `Lazy(int)`, `create_lazy` transforms element of type `() -> A` into `Lazy(A)`, and `eval` evaluates an element of type `Lazy(A)` to an element of type `A`. The keyword `this` is used in the definition of `fib_aux`. If we use `fib_aux`, C++ will first instantiate `fib_aux` as an empty class and use this value when evaluating the right hand side. We can only obtained a truely recursive definition using `this`. When evaluated, one sees that the nth element of `fib` computes to $fib(n)$ and this computation is the efficient one in which previous calls of $fib(k)$ are memoized. Replacing `Lazy(X)` by `() -> X`, results in an implementation of the fibonacci numbers which is still correct, but requires exponential space since memoization is lost.

Lazy evaluation in C++ has been studied extensively in the literature (eg. [Sch00], [MS00], [Kel97]) where all implementations are restricted to lazy lists. We introduce a general type of lazy elements of an arbitrary type, which not only corresponds to call-by-name (usually achieved by replacing a type A by $() \to A$), but also guarantees that elements are evaluated once, as required by true lazy evaluation. There is no need to a new delay construct to C++ since our implementation of laziness makes use of the existing language of C++ only.

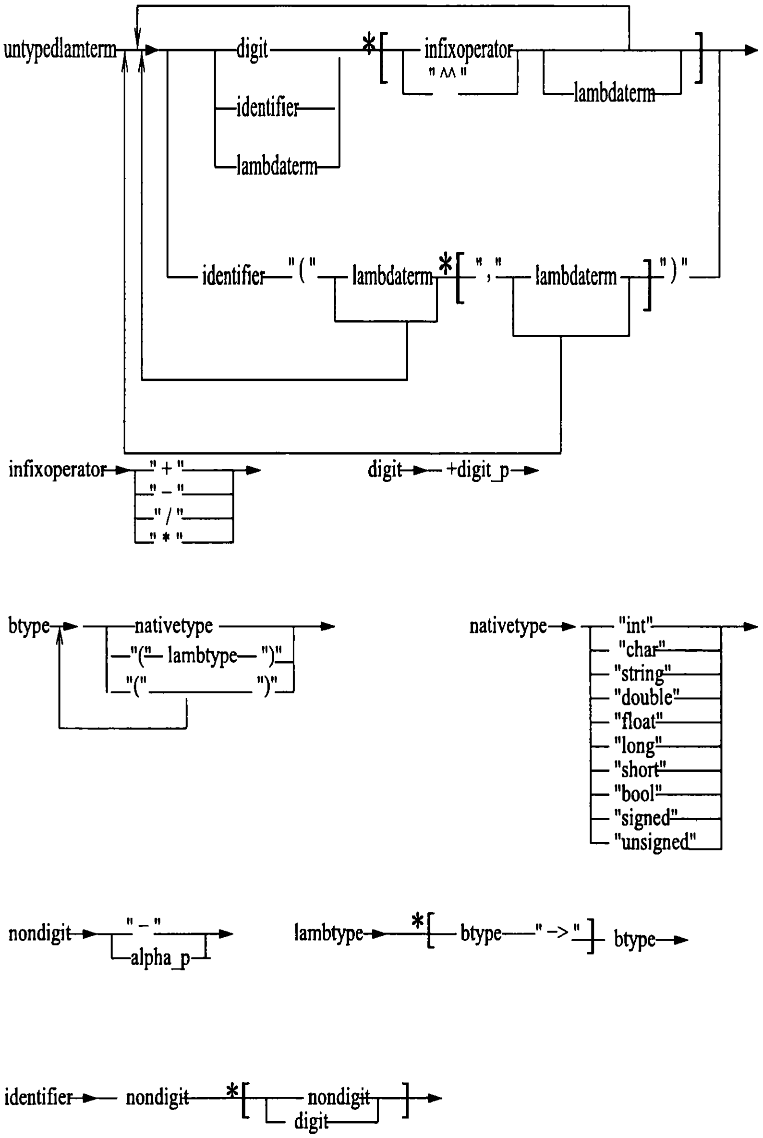

Figure 4.4: Continuation of syntax diagram of the λ-term grammar

Chapter 5

Implementation of The Parser-Translator Program

In the previous chapter, we have discussed the design of the Parser-Translator program. Now we describe the implementation. We write in the following PTP as reference to our Parser-Translator program. The PTP does two jobs which are parsing and translating. When an expression representing a simply typed λ-term is input to the PTP, it parses the input and translates it to a sequence of C++ statements. An overview of how the PTP works are shown in the Figure 5.1. More details of the parsing and translation done by the PTP will be discussed in the coming sections. We will also give an example of a simply typed λ-term input to the PTP, and discuss the translation and execution of the translated code along with the representation of the memory allocation. This information is important in proving the correctness of the translated code because we build the mathematical model from the formal semantics.

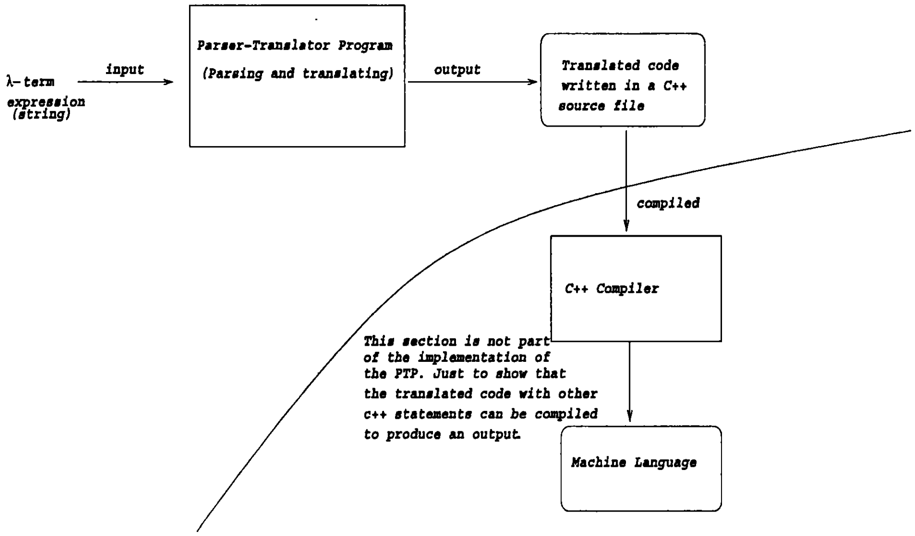

Figure 5.1: Overview of the Implementation of the PTP

5.1 Parsing Phase

PTP creates a parser that parse an input of λ-expression following a specific syntax. The parser calls the scanner to obtain the tokens of the input string and assembles the tokens into a parse tree. The tree is then passed to the translation phase where a sequence of C++ statements, equivalent to the input λ-expression, will be generated. Before explaining in detail the parsing phase, it is important to introduce some of the concepts needed in the discussion of this section. A precise definition of what it means for a sequence of C++ statements to be equivalent to a λ-term as well as a rigorous proof that equivalence holds for the code generated by the PTP will be given in the next chapter.

5.1.1 General Concepts in Scanning and Parsing

The purpose of scanning and parsing is to recognize the structure of the code disregarding the meaning of it. **Scanning** which is also known as lexical analysis simplifies the task of the parser by reducing the size of the input. It reads the input as single characters and groups them into tokens (the smallest meaningful units in a program). **Tokens** are the basic building blocks of a program such as identifiers, digits, keywords and other symbols. We use the notation of regular expression to specify tokens. A **regular expression** generates a regular set where regular sets are sets of strings that can be defined using three operations: concatenation, alternation and Kleene star. Concatenation is used when a regular expression generates two regular expressions next to each other where one string is followed by (concatenate with) another string. Alternation provides choice from a finite set of alternatives for the regular expression usually using the symbol (|). Kleene star is used for arbitrary (possibly zero) many repetitions of a regular expression. For example in C++, a digit_sequence can be generated by the following regular expression:

```
         digit --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
digit_sequence --> digit  digit*
```

Notice that in the above regular expression, the three rules are applied. `digit` is defined as being a digit, and the `digit_sequence` makes use of concatenation and Kleene star to generate integers like 11, 12, 19 and so on. To generate a valid string, the regular expression is scanned from left to right choosing alternatives and repetitions.

Regular expressions are suitable for defining tokens but are not able to specify nested constructs which is important in programming languages. Tokens are translated by the **parser** into a parse tree. This **parse tree** represents higher-level constructs in terms of their constituents which are combined based on a set of potentially recursive rules known as a **context-free grammar**. Every rule in a context free grammar is known as a **production**. The symbol on the left hand side of a production is known as a variable or **non terminal**. **Terminals** are the symbol that make up a string derived from a grammar and they cannot appear on the left hand side of a production. The **Start symbol** names the construct defined by the overall grammar and it is usually the non terminal on the first production.

Context free grammars use notation called **Backus-Naur Form** or BNF in honour of John Backus and Peter Naur . BNF when augmented with extra operators such as concatenation

(|), Kleene star(*), Kleene plus (+) and meta-level parenthesis of regular expressions is called **extended BNF** (EBNF) [ISO96]. For example, a C++ identifier can be generated by the following production rules:

```
identifier  --> nondigit | identifier nondigit | identifier digit
   nondigit --> _ | a | b | c | d | ... | z
                | A | B | C | D | ...| Z
      digit --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Using the production rules above, we can generate identifiers such as `name`, `first_name`, `room1` and so on. The non terminal identifier is the start symbol. The production rule for identifier make use of the recursive construct and alternation to define it. The parser will organize the tokens such as identifier, digit and nondigit into a parse tree based on the grammar above. For example the parse tree for identifier **room1** is shown below:

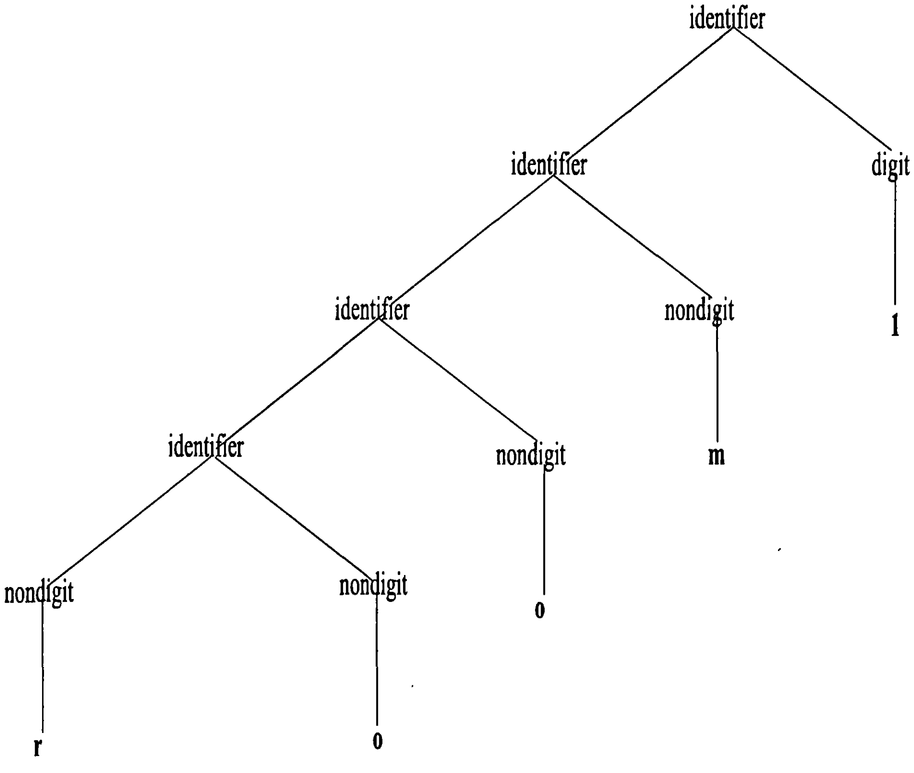

Figure 5.2: Parse tree for identifier room1

The grammar is parsed using the LL parsing technique. LL parser parses input from Left to right and constructs a Lefmost derivation of the expression. An LL parser is called an LL(k) parser if it uses k tokens of look-ahead when parsing a statement. Among the LL(k) grammars, LL(1) grammar is very popular because the corresponding parser need only to look at the next token to make their parsing decision. As mentioned in the previous chapter, PTP is developed using the Spirit. The Spirit parser framework is an object oriented recursive descent parser generator framework where the parser objects are composed through operator overloading and the result is a backtracking LL(∞) that is capable of parsing rather ambiguous grammars.

5.1.2 Parsing a statement

Spirit allows us to approximate the syntax of EBNF completely in C++. So, the grammar given above can be written as:

```
identifier = nondigit >> *(nondigit | digit);
  nondigit = ch_p('_') | alpha_p;
     digit = +digit_p;
```

Notice that from the grammar written above, left recursion is avoided in the rule for identifiers by making use of the sequence operator (>>) and the Kleene star instead. Grammars using Spirit should eliminate direct and indirect left recursion to avoid the parser entering an infinite loop. To simplify the digit and nondigit rule we make use of the predefined parser in Spirit such as digit_p, alpha_p and ch_p. digit_p parses digit, alpha_p parses alphabetical characters and ch_p parses any single character. The Kleene plus(+) in the digit rule means that the digit can appear one or more times. An overview of how a statement is parsed is shown in the Figure 5.3

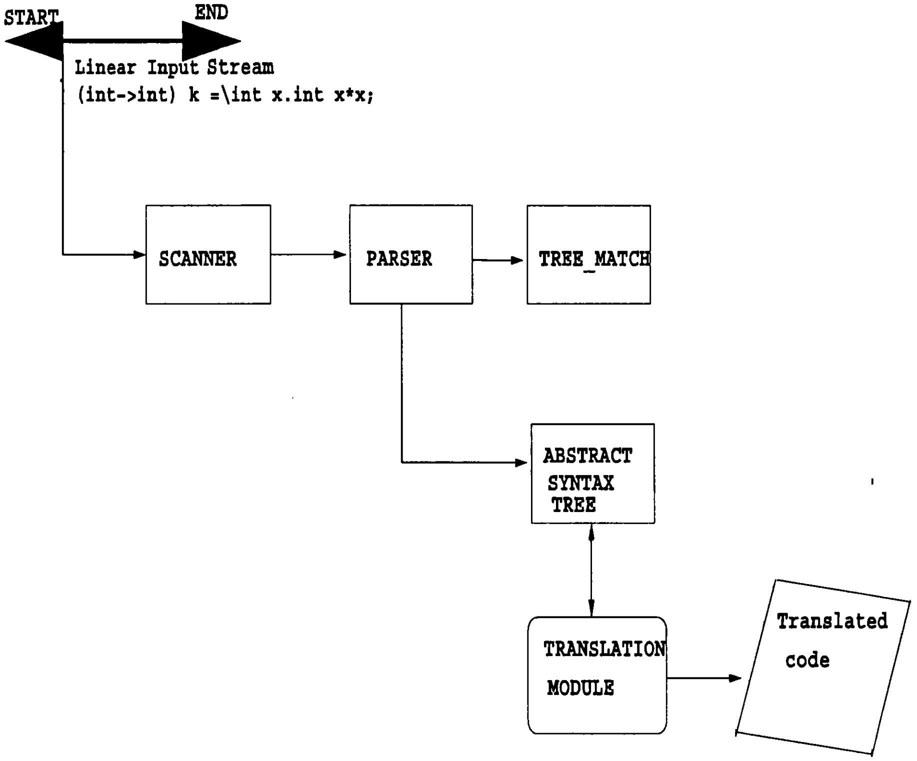

Figure 5.3: Overview of how parsing is executed

In the Figure 5.3, the linear input stream of data is read sequentially by the scanner from the start to the end. The parser does the work of recognizing the input read by the scanner by attempting to match the input with the grammar rules. The parser reports the success or failure of the match through a tree_match object, which we use in order to generate a parse tree. More precisely, in the PTP, we generate an abstract syntax tree (ast)

[Bo02], which is similar to a parse tree. The only difference is that it has the advantage of having more directives which can reduce your code in processing the abstract syntax tree. When the match is successful, an abstract syntax tree is generated where the translation module will traverse or parses the tree to get the translated code. The input stream `(int->int) k=\int x.int x*x;` when parsed will generate the abstract syntax tree shown in Figure 5.4.

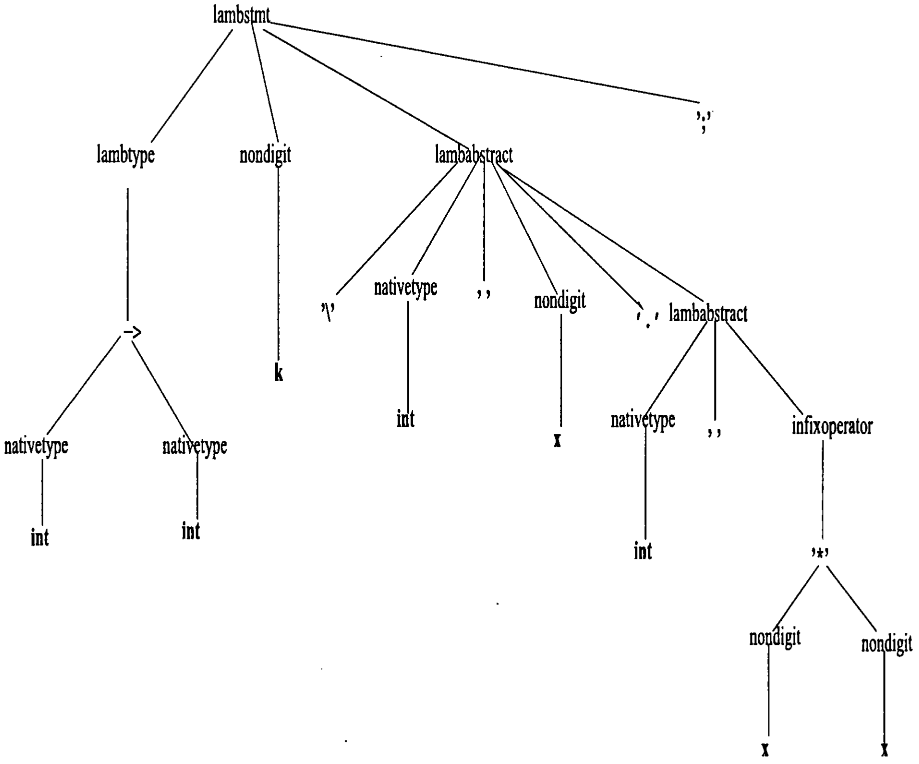

Figure 5.4: An abstract syntax tree for the input statement in fig 5.3

Notice that in the Figure 5.4, the arrow (→) symbol in the `lambtype` and the `infixoperator` is considered the root. The arrow symbol is the root for `lambtype` (int→int) and the `infixoperator(*)` is the root for its operands. This is due to the directive `root_node_d` used to enforce the symbols mentioned as the root node. Also in the abstract syntax tree in Figure 5.4, brackets for the function type (int→int) are not considered as a node in the tree. This is because the directive `inner_node_d` directs the parser to just take the expression in the brackets as the node in the tree. The `root_node_d` and `inner_node_d` directives are directives that only effect the abstract syntax tree. If we do not use the `root_node_d` and `inner_node_d` directives in the grammar, the structure of the abstract syntax tree will be different where brackets in (int→int) will be taken as nodes in the tree and the `lambtype` will have three children of the same level. Similarly, the expression (x*x) would have as syntax tree the variables and the symbol * at the same level. The structure of the the abstract syntax tree without using directives `inner_node_d` and `root_node_d` can be seen in Figure 5.5.

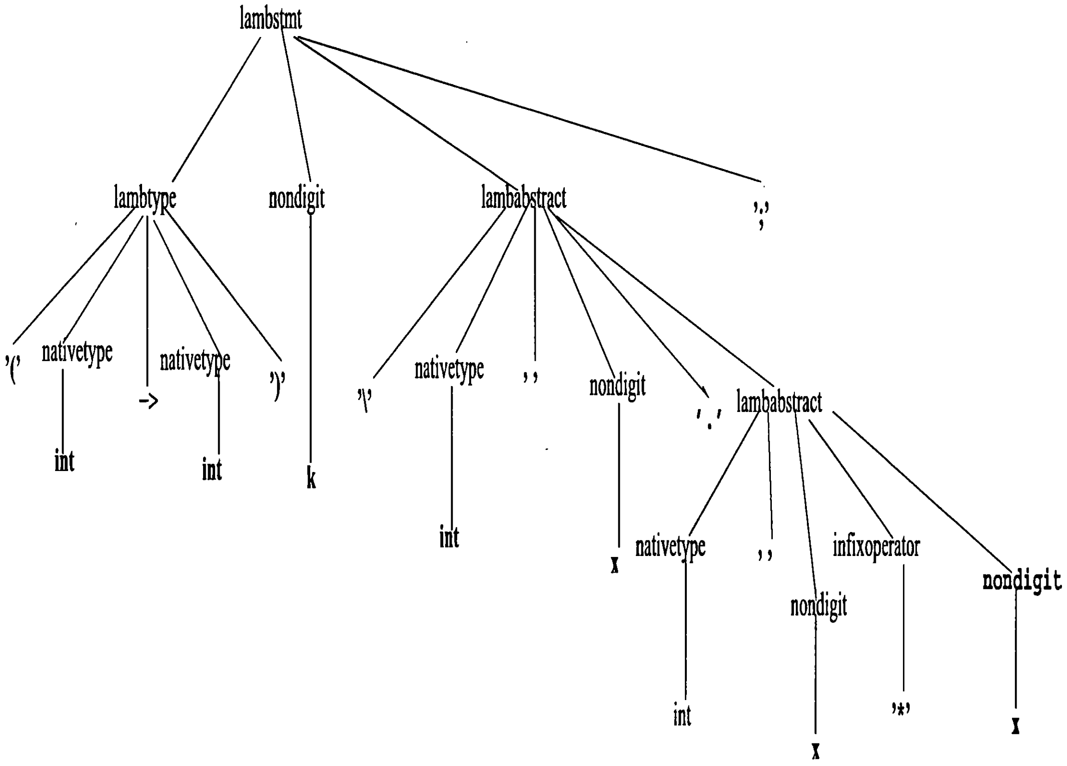

Figure 5.5: An abstract syntax tree without the directives inner_node and root_node

These directives are useful in simplifying the structure of the tree so as to ease the process of traversing and transforming the tree to get the translated code. The complete grammar for the λ-term coded in Spirit is shown in Appendix A.

The `tree_match` class has an operator `bool()` that we can test for a successful match. When a full match meaning the parser has successfully parsed all the input, the translation module for processing the tree is executed to get the translated code. This phase is called translation phase.

5.2 Translation phase

The translation follows the object-oriented method of programming where classes and the concept of inheritance are involved in producing the translated code. We also use pointers and dynamic allocation of the classes in the memory.

Translation is executed by the PTP in three stages :

1. Translation of the function type.
2. Translation of the λ-term.
3. Translation of the expression.

When the expression representing a λ-term is input to the PTP following the syntax discussed earlier, it undergoes several stages of translation to produce the translated code. First, the function type is determined and the abstract class for the function type is defined. The λ-term is then defined as a derived class for the function type abstract class where the virtual operator() is overloaded in the λ-term class. Finally, the λ-expression is translated as an expression that involves instantiating the λ-term class.

As mentioned in the previous chapter, there are two categories of λ-terms i.e. the typed and untyped λ-term. There are two tasks that need to be done for the translation i.e. class definition and λ-expression generation. Thus there are two kinds of method for the typed and untyped λ-term that correspond to the two tasks. We call these modules class_def() and term_exp(), which correspond to class definition and λ-expression respectively. Shown in Figure 5.6 and Figure 5.7 is the general idea of how several forms of the typed and untyped λ-term are translated.

Typed λ-term	class definition	term expression
abstraction $(\lambda x^{A} \cdot r)$	r.class_def()	r.term_exp()
application $(r \;\; s)$	r.class_def()	'(' + '*' + '(' + r.term_exp() + ')' + ')'
	+s.class_def()	+ '(' + s.term_exp() + ')'
application$(r \;\; u)$	r.class_def()	'(' + '*' + '(' + r.term_exp() + ')' '+' ')'
	+u.class_def()	+ '(' + u.term_exp() + ')'

Figure 5.6: Translation of the typed λ-term

Untyped λ-term	class definition	term expression
number	–	number
identifier	–	identifier
r infixoperator s	r.class_def()	r.term_exp() + infixoperator
	+ s.class_def()	+ s.term_exp()
r infixoperator u	r.class_def()	r.term_exp() + infixoperator
	+ u.class_def()	+ u.term_exp()
t infixoperator u	t.class_def()	t.term_exp() + infixoperator
	+ u.class_def()	+ u.term_exp()
t infixoperator s	t.class_def()	t.term_exp() + infixoperator
	+ s.class_def()	+ s.term_exp()
functionsymbol$(t_1, t_2, \ldots, t_n)$	t_1.class_def() +t_2.class_def()	functionsymbol + '(' +
	+ ... + t_n.class_def()	t_1.term_exp()+t_2.term_exp()
		... + t_n.term_exp() + ')'

Figure 5.7: Translation of the untyped λ-term

The method term_exp() and class_def() is not the actual method used in the PTP to execute the translation. They are just as representatives of the methods that are involved in the mentioned tasks. In the Figure 5.6 and Figure 5.7, variables r, s, t and u represents typed and untyped λ-terms. Typed λ-terms are represented by variables r and s, whereas t and u represent untyped λ-terms. The symbol '+' means concatenation. r.class_def() + u.class_def means the string of class definitions for r produced by the mentioned method is concatenated with the string of class definitions for u. Characters that are enclosed in a single quotation are just strings such as '*' and '('.

5.2.1 Translation of the Function Type

For each λ-term input to the PTP, the type is defined first. In PTP there are two kinds of type that is basic C++ type (eg. `int, char`) and the arrow type or function type (eg. (int→int)). The type of the term or function is determined based on the left type and the right type of the term. How the type of the term is determined in the PTP can be shown in the Figure 5.8.

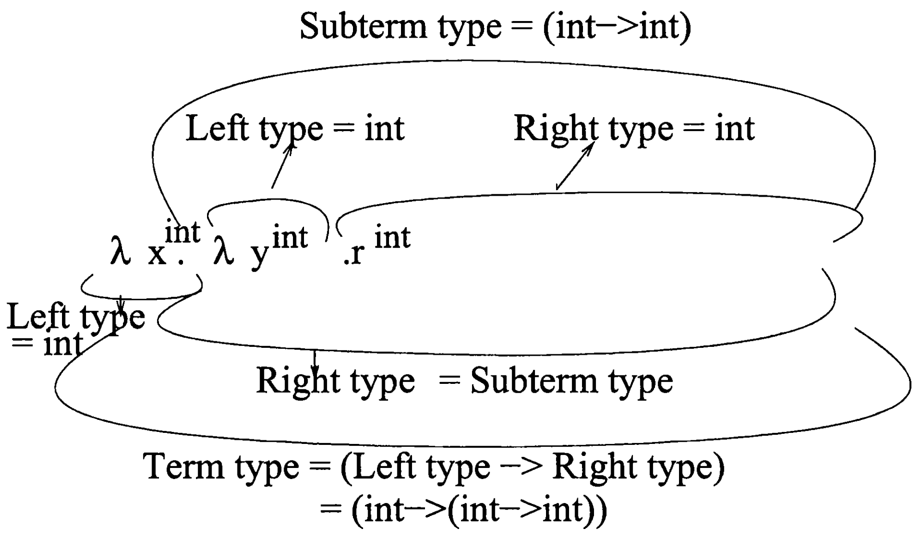

Figure 5.8: How the type of a two argument term are determined by the PTP

In the figure 5.8 we have a two argument λ-abstraction $\lambda x^{\text{int}}, y^{\text{int}}.r$ where the term r is of the type int. The type int for the variable x is the left type for the term and the type for the subterm $\lambda y^{\text{int}} r$ is the right type for the term. The left type is the input type for the term and the right type (subterm type) is the output type for the term. The type for the subterm is determined similarly giving an arrow type (int→int) where the type int for y is the left type and the type int for the term r is the right type. Finally the type of the term is determined as (int→(int→int)) where the left type is int and the right type is (int→int). For each function type an abstract class is defined with a virtual operator that will be overloaded in the definition of the λ-term and the type itself is the type pointers to an object of this abstract class. In general, the abstract class of the function type is defined as follows :

```
class type_classname_aux
{ public : virtual output_type  operator()
                   (input_type  x) = 0;};

\\ =0 means it is a pure virtual function

typedef type_classname_aux*
```

```
type_classname;
```

For the *type_classname* we make use of letters C and D to represent open and close brackets respectively, and an underscore for an arrow. For example, Cint_Cint_intDD means (int→(int→int)). The *input_type* and the *output_type* make up the *type_class_name* of a function type which can be a basic type or an arrow type. For example, the *type_class_name* Cint_intD represents the function type where the *input_type* is int and the *output_type* is int. Similarly for Cint_Cint_intDD, the *input_type* is int and the *ouput_type* is Cint_intD. The definition of the function type is represented in stages. If the function type (int→(int→int)) is to be defined, the definition will be as follows :

```
\\This is the definition of (int->int)
class Cint_intD_aux
{ public : virtual int operator() {int x} = 0;};

typedef Cint_intD_aux* Cint_intD;

\\This  is the definition of (int->(int->int))
class Cint_Cint_intDD_aux
{public : virtual Cint_intD operator() {int x} = 0};

typedef Cint_Cint_intDD_aux* Cint_Cint_intDD;
```

We could use C++ templates in the definition of the function type so as to make the classes generic. It would be easier defining λ-type by hand using a general C++ template for the class corresponding. to the arrow type. But we are not using them in the generated code (translated code) in order to obtain a much faster compilation for the code and also making the task of correctness proof less complicated.

5.2.2 Translation of the λ-term

The concept of inheritance is involved in the definition of the λ-term where the function type abstract class will be the base class for the λ-term class. A general definition of the λ-term class is as follows:

```
class
term_classname:publictype_classname_aux{
  public : [declaration of free variable in the
term];
  constructor with/without arguments;
  virtual output_type operator() (input_type
         bound_variable)
  { return body of term;};
};
```

The members of the class are free variables of the term (if there is any) and the overloaded virtual operator. The virtual operator in the function type class is overloaded here with the bound variable as argument and the return statement here depends on the body of the term. If the body of the term is a subterm, then the return statement is an instantiation of the subterm class. Otherwise, the return statement is just returning the application of the body of the term. The instantiation of the subterm class is done by using the operator new followed by the constructor of the subterm class with *bound_variable* as argument.

For each λ-term or subterm, a class will be defined as an instance of the function type class and it will be translated in stages. A λ-term can be translated into one or more classes depending on the arguments of the term such as for a two argument λ-abstraction, two classes will be defined, one for the term and one for the subterm. The translation of a two argument λ-abstraction $\lambda x^{\text{int}}, y^{\text{int}}.x * y$ can be pictured as in Figure 5.9.

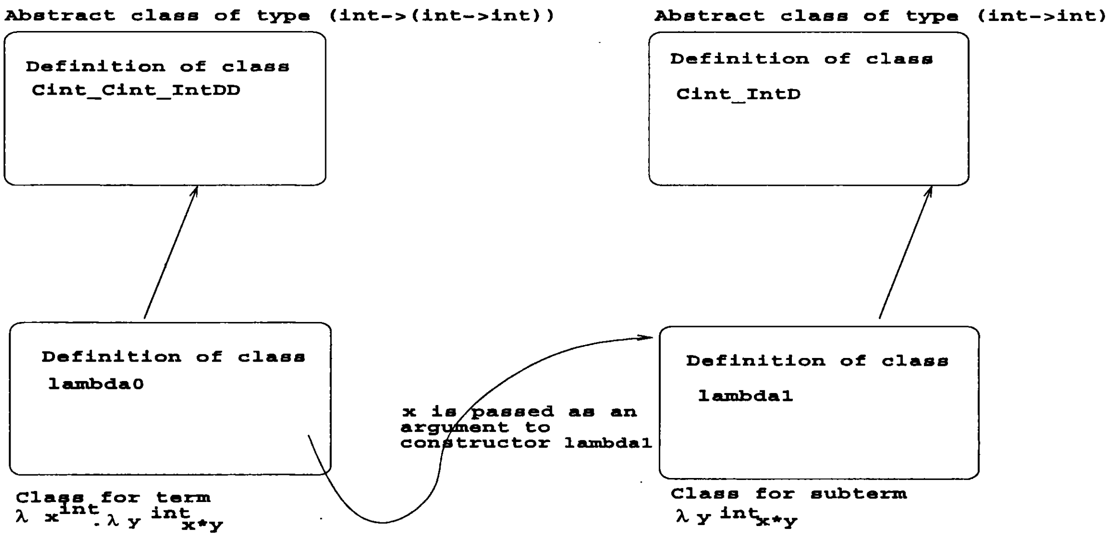
Abstract class of type (int->(int->int))
Abstract class of type (int->int)
Definition of class Cint_Cint_IntDD
Definition of class Cint_IntD
Definition of class lambda0
x is passed as an argument to constructor lambda1
Definition of class lambda1
Class for term λ x^int . λ y int x*y
Class for subterm λ y int x*y

Figure 5.9: The translation of a two argument λ-abstraction

In the Figure 5.9, the λ-abstraction is translated by defining two classes that is lambda0 and lambda1 where lambda0 is the class for the term and lambda1 is the class for the subterm. Each class for the term has a function type abstract class as a base class and the virtual operator is overloaded in the term class. For the term $\lambda x^{\text{int}}, y^{\text{int}}.x * y$, there is no free variable and its bound variable is x which means x is bound in the entire abstraction. This term or function accepts an int type argument and returns a function of the type (int→int). So the definition of the term $\lambda x^{\text{int}}, y^{\text{int}}.x * y$ is:

```
class lambda0:public Cint_Cint_intDD_aux{
  public:
  lambda0( ) { };
  virtual Cint_intD operator  ( ) (int x)
  {return new lambda1(x);};
};
```

The returned function is the internal abstraction or subterm $\lambda y^{\text{int}}.x * y$. In the subterm, y is bound and x is free. In the subterm class the virtual operator is overloaded by accepting an argument y of type int and return the expression $x * y$ where it is also of the type int. The definition of the subterm class is:

```
class lambda1 : public Cint_intD_aux{
  public : int x;
  lambda1(int x) { this->x = x;};
  virtual int operator ( ) (int y)
  {return x * y;};
};
```

5.2.3 Translation of the expression

Finally, at this stage the expression input to the PTP is translated into a C++ expression which involves instantiation of λ-term classes and pointers. In the case of the λ-term as an application term, the λ-term involved in the application is translated in a similar way as described above, and it is instantiated by dynamically allocating the memory for the λ-term class using operator `new` (eg. `new lambda0()`). This means that the λ-term class is allocated an address in memory and to reference it we make use of a pointer. We use the dereference operator `(*)` to the constructor of the λ-term class(eg. `*(new lambda0())`) to access the function of the class. For example, the λ-abstraction above when applied to 3 and 2 written in our syntax as `((\int x.\int y.int x*y)^^3)^^2;`, which is equivalent to $((\lambda x^{\text{int}}, y^{\text{int}}.x * y)3)2$ will be translated as :

```
(*((*(new lambda0()))(3)))(2)
```

Here are a few examples of the translation of the λ-expression :

1) `(int->int) k =\int x.int x*x;`
   translated to: `Cint_intD  k = new lambda0();`
2) `int l =(\int x.int x*x)^^3;`
   translated to: `int l = (*(new lambda0()))(3);`
3) `(int->int) m =(\(int->int) f.\int x.int f^^x)^^(\int y.int y+y);`
   translated to: `Cint_intD m = (*(new lambda0()))(new lambda2());`

5.3 The Execution of the Translated Code

There are several areas of the memory that are used during the evaluation of a λ-expression. Local variables and function parameters are stored on the stack, while instruction code in the code space and global variables are in the global space. Registers are used as internal housekeeping functions such as keeping track of the top of the stack and the instruction pointer. Almost all of the remaining memory is given to the heap. The heap [GJ98] is a dynamic memory area allocated by the command `new` and freed by `delete`. When using `new`, memory for the data the pointer is pointing to is allocated on the heap, and the pointer is assigned the address of the location on the heap. The main property of the heap is that the memory that is reserved is still available until it is explicitly freed. In the translated code, the instantiation of the λ-term class is by dynamically allocating them on the heap.

We use an example in order to explain the execution of the translated code. We choose a more complex term as an example so as the discussion covers more aspects in explaining the execution of the translated code.

The statement to be executed is written as follows in our syntax:

```
int k =((\(int->int) f.\int x.int f^^(f^^x))
        ^^(\int x.int x+2))^^3
```

This corresponds to the λ-term: :

$$\text{int } k = (\lambda f^{(\text{int}\to\text{int})}, x^{\text{int}}.f(f\,x))(\lambda x^{\text{int}}.x + 2)3$$

The function types determined for the λ-terms involved in the expression are defined as follows:

```
class Cint_intD_aux
{ public : virtual int operator() (int x) = 0; };

typedef Cint_intD_aux*  Cint_intD;

//Definition of type : ((int-> int) (int-> int))
class CCint_intD_Cint_intDD_aux
{  public : virtual Cint_intD operator() (Cint_intD x) = 0; };

typedef CCint_intD_Cint_intDD_aux*  CCint_intD_Cint_intDD;
```

The classes defined when translating the λ-term :

```
(\int->int)f.\int x.int f^^(f^^x)
```

in the statement above are as follows:

```
 class lambda1 : public Cint_intD_aux{
   public :Cint_intD f;
   lambda1( Cint_intD f)  {   this-> f = f;};
   virtual int operator () (int x)
   { return (*(f))((*(f))(x)); };
 };
 class lambda0 : public CCint_intD_Cint_intDD_aux{
   public :
   lambda0( ) { };
   virtual Cint_intD operator () (Cint_intD f)
   { return new  lambda1( f); }
 };
```

In the translation above the λ-term is translated as the definition of the classes `lambda0` (for the term `\(int->int) f.\int x.int f^^(f^^x))` and `lambda1` (for the sub-term `\int x.int f^^(f^^x)`). The expression `f^^(f^^x)` in the body of the sub-term is also translated as `(*(f))((*(f))(x))` using the dereference operator(*) where the function `f` is applied twice.

The λ-term `\int x.int x+2` is translated as follows :

```
 class lambda2 : public Cint_intD_aux{
   public :
   lambda2( ) { };
   virtual int operator () (int x)
   { return x + 2; };
```

The λ statement above will be finally translated as an expression given below :

```
int k = (*((*( new lambda0( )))( new lambda2( ))))(3);
```

The statement `int k = (*((*( new lambda0( )))( new lambda2( ))))(3);` is equivalent to sequence of statements shown below:

```
CCint_intD_Cint_intDD k1 = new lambda0();
            Cint_intD k2 = new lambda2();
            Cint_intD k3 = (*(k1))(k2);
                  int k4 = (*(k3))(3);
```

For a better explaination of how the expression k is evaluated, we based our explaination on the sequence of statements above so as to show the stages of evaluation. The execution of the statement can be pictured in the figure 5.10:

The parts of the memory such as the stack for variables and parameters, code space and the heap is pictured separately from each other in Figure 5.10, even though we know they are in the same part of the memory. The reason for this is to show a clear view of the execution of the translated code.

First, the classes of the λ-term are dynamically allocated on the heap by the expression k1 and k2 where the operator method of lambda0 creates an instance of lambda1. In the expression k3, the application of the two λ-terms invokes the operator() method that is overloaded in lambda0 with the instance of variable f set to the instance of lambda2. The result of the application of lambda0 to lambda2 is the instance of lambda1 with f bound to k2. Expression 3 is evaluated to 3. Then the evaluation comes to the stage where the expression of the body of lambda1 i.e. `(*(f))((*(f))(x))` is evaluated.

This evaluation can be shown clearly if we break down the body expression as:

```
  int y1 = ((*(f))(x);
  int y2 = (*(f))(y1);
```

In the expression y1, the operator() method of lambda1 is called. This will make a call to the operator() method of f which is bound to the instance of lambda2 and apply it to 3(where x takes the value 3). This will evaluate to 5. Then the expression y2, will make the operator method of f to be called again, which is still bound to the instance of lambda2 and apply it to y1 which evaluates to 5 giving the result 7. The evaluation of the expression discussed follows the call-by-value evaluation strategy. This evaluation strategy has been discussed in the Chapter 3.

We did not include memory management in the translated code. This is due to the difficulty of doing memory management when using nested λ-terms because we cannot really predict

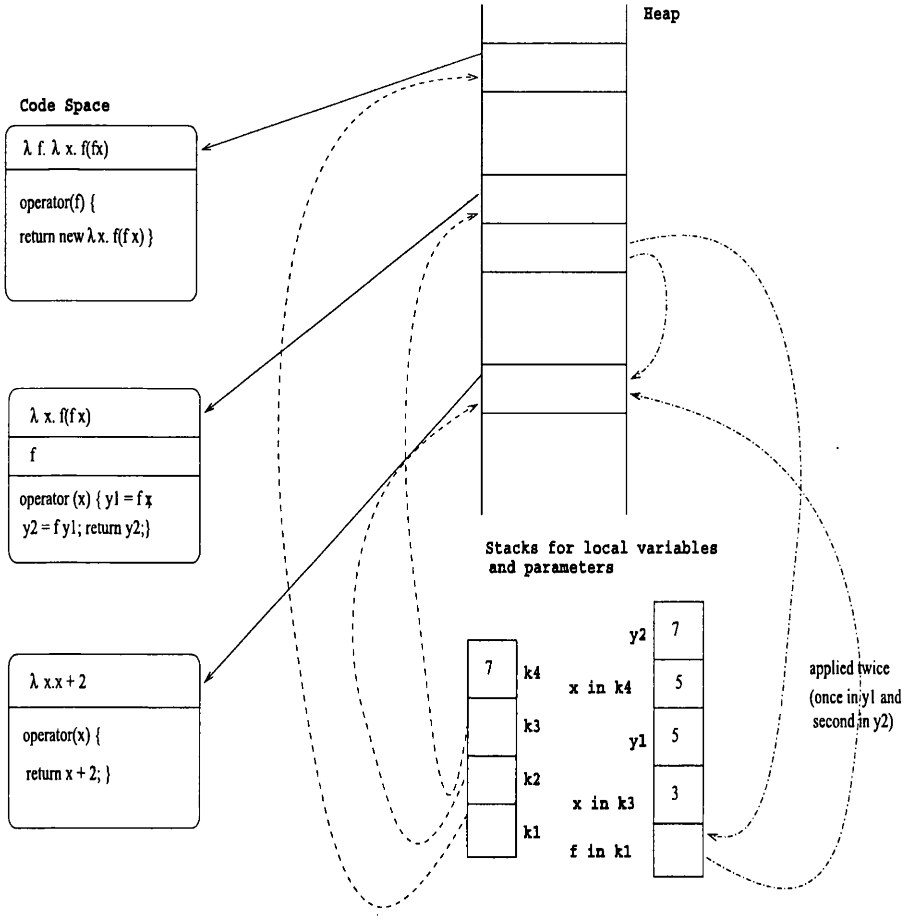

Figure 5.10: Memory representation of the execution of the translated code

when the memory are in use or free. But we did mention in our papers that we wanted to rely on a garbage collected version of C++.

5.4 Testing of the Translated Code

How do we know that what has been translated is correct and it follows the functional method of programming? We answer this question in two ways: in Chapter 6 we give a formal correctness for the translation program. However, this proof is carried out with respect to a mathematical model of a fragment of C++ and there is no formal proof that this model actually reflects the behaviour of C++ correctly (although it is fairly obvious that it does). Therefore there is is still a demand for testing the program correctness. In addition, testing allows us to assess the efficiency of the program.

Well, first the PTP that has been developed was tested on several types of λ-terms from simple to complex ones. This was discussed in the previous chapter and it was found that the result given by the program are correct when compared with the manual evaluation of the terms. Before discussing further, we need to define Church numerals as they are part of the testing samples.

Church numerals are the representations of natural numbers under Church encoding. Church numerals 0, 1, 2, .., n, are defined as follows in the λ-calculus:

$$\begin{array}{lll} 0 & \equiv & \lambda\, f.\lambda\, x.\, x \\ 1 & \equiv & \lambda\, f.\lambda\, x.\, f\ x \\ 2 & \equiv & \lambda\, f.\lambda\, x.\, f\ (f\ x) \\ 3 & \equiv & \lambda\, f.\lambda\, x.\, f\ (f\ (f\ x)) \\ \ldots & \ldots & \\ \mathbf{n} & \equiv & \lambda\, f.\lambda\, x.\, f^n\ x \end{array}$$

The natural number n is represented by the church numeral **n**, which has property that for any λ-terms F and x,

$$\mathbf{n}\, F\ x =_\beta\ F^n\ x$$

In the following we describe the testing of the translated code with more advanced and harder examples. The testing examples are chosen to test the following correctness and performance aspects of the programs:

- Correctness:
  - Bound renaming

    All examples require α-conversion, that is renaming of bound variables, when computed via term rewriting. Our program doesn't do α-conversion explicitly, but only implicitly through the implementation of classes. The tests show that this implicit α-conversion is done correctly.

  - Higher-types

    In order to test that higher types are implemented correctly, all examples involve higher-types, that is variables of a function type. The highest types occur when a Church numeral n is applied to a Church numeral m: $n\,m$. In that case $n = \lambda\, f\ \lambda x.f^n\, x$ where f is of type (Int→Int)→(Int→Int) and x is of type Int→Int. Consequently the term n has type ((Int→Int)→(Int→Int))→((Int→Int)→(Int→Int)). Even higher types are needed to type $k\,n\,m\,m$ etc..

- Efficiency
  - Large results

    Evaluating, for example the term $n\,m$ succ 0 (where $n\,m$ are Church numerals) yields the number m^n. In this way one easily obtains results that go to the limit of the range of floating point number. More dramatically $k\,n\,m$ succ 0 evaluates to m^{n^k}.

  - Long computations

    If in the example above one replaces the successor function succ by the identity function $(\lambda\, x.x)$, then the results will always 0, yet the computation takes as long as with the successor function. In this way pure performance is tested without limitations given by the size of the output.

These tests are based on the execution time of the translated code of the λ-terms shown in the Figures 5.11, 5.12, and 5.13. We used $t_2, t_3, \ldots, t_n$ to represent the church numeral 2, 3 ,. . ., n and succ to represent a function successor. The Church numerals 2, 3, 4, 5 and 6 are applied to a successor function ($\lambda\ x.\ x + 1$) and applied again to 0. The term:

$$s_n^k := \underbrace{t_n(\ldots(t_n}_{k} \text{ succ}\ldots)\,0 \equiv n^k$$

If succ function is replaced by the identity function, the term is defined as follows:

$$i_n^k := \underbrace{t_n(\ldots(t_n}_{k} \text{ identity}\ldots)\,0 \equiv 0$$

Other computations of the Church numeral are also tested and the term is defined as follows:

$$s_{n_1 n_2}^{k_1 k_2} := \underbrace{t_{n_1}(\ldots(t_{n_1}}_{k_1} \text{ succ }\ldots)(\underbrace{t_{n_2}(\ldots(t_{n_2}}_{k_2} \text{ succ})\ldots)\,0) \equiv n_1^{k_1} + {n_2}^{k_2}$$

When a Church numeral (eg. t_2) applied to a function square ($\lambda x.x * x$) and applied to 2 will give 2^4 which is represented in the Figure 5.13 as sq_2^4. If the term sq_2^4 is applied twice will give $(2^4)^4$.

Based on the execution time of the λ-terms, the computation of the Church numerals is limited up to a certain exponent given as follows:

Church numeral	Exponent
2	30
3	19
4	15
5	14
6	11

We say this is because the value of the computation for the term s_2^{31}, s_3^{20}, s_4^{16}, s_5^{15}, and s_6^{12} 'does not make sense' (0 or negative value) and the execution time is quite long (sometimes ∞). This is due to the complexity increasing as the exponent increases. The range of an integer value for the compiler can also be the cause of the limitation of the computation of the term.

If one looks at the translated code naively, it seems quite inefficient to introduce a new element for each λ-term arising. But if one looks at what is really going on, one sees that not the λ-term is stored , but only the free variables. If we look at $\lambda\ x.\ x + y$, the code for the class $\lambda\ x.\ x + y$ is stored as part of the source code and what is stored on the heap is the information that we are referring to the class referring to $\lambda\ x.\ x + y$.

A more reliable way of verifying that the translation code is correct, is by modeling a simplified C++ compiler that executes the translated code. By this modeling we can prove that the translation code is correct for all kinds of λ-terms. The next chapter will give a proof of the correctness of the translation.

No.	λ-terms	Output	Execution time(sec)
1.	s_2^{12}	4096	0
2.	s_2^{13}	8192	0
3.	s_2^{14}	16384	0
4.	s_2^{15}	32768	0
5.	s_2^{16}	65536	0
6.	s_2^{17}	131072	0
7.	s_2^{18}	262144	0
8.	s_2^{19}	524288	0
9.	s_2^{20}	1048576	0.046
10.	s_2^{21}	209152	0.093
11.	s_2^{22}	4194304	0.187
12.	s_2^{23}	8388608	0.39
13.	s_2^{24}	16771216	0.765
14.	s_2^{25}	33554432	1.531
15.	s_2^{26}	67108864	3.031
16.	s_2^{27}	134217728	6.078
17.	s_2^{28}	268435456	12.14
18.	s_2^{29}	536870912	24.375
19.	s_2^{30}	1073741824	48.562
20.	s_2^{31}	-2147483648	97.109
21.	s_2^{32}	0	194.218
22.	s_3^{8}	6561	0
23.	s_3^{9}	19683	0
24.	s_3^{10}	59049	0
25.	s_3^{11}	177147	0.015
26.	s_3^{12}	531441	0.031
27.	s_3^{13}	1594323	0.062
28.	s_3^{14}	4782969	0.156
29.	s_3^{15}	14348907	0.484
30.	s_3^{16}	43046721	1.421
31.	s_3^{17}	129140163	4.265
32.	s_3^{18}	387420481	12.812
33.	s_3^{19}	1162261467	38.453
34.	s_3^{20}	-808182895	115.281
35.	s_4^{5}	1024	0

Figure 5.11: Table of Execution Time for the Samples of λ-terms

No.	λ-terms	Output	Execution time(sec)
36.	s_4^6	4096	0
37.	s_4^7	16384	0
38.	s_4^8	65536	0
39.	s_4^9	262144	0015
40.	s_4^{10}	1048576	0.031
41.	s_4^{11}	4194304	0.109
42.	s_4^{12}	16777216	0.453
43.	s_4^{13}	67108864	1.859
44.	s_4^{14}	268435456	7.422
45.	s_4^{15}	1073741824	29.672
46.	s_4^{16}	0	118.687
47.	s_4^{17}	0	∞
48.	s_5^4	625	0
49.	s_5^5	3125	0
50.	s_5^6	15625	0
51.	s_5^7	78125	0
52.	s_5^8	390625	0.015
53.	s_5^9	1953125	0.046
54.	s_5^{10}	9765625	0.25
55.	s_5^{11}	48828125	1.218
56.	s_5^{12}	244140625	6.14
57.	s_5^{13}	1220703125	30.671
58.	s_5^{14}	1808548329	153.406
59.	s_5^{15}	0	∞
60.	s_6^4	1296	0
61.	s_6^5	7776	0
62.	s_6^6	46656	0
63.	s_6^7	279936	0.015
64.	s_6^8	1679616	0.031
65.	s_6^9	10071696	0.234
66.	s_6^{10}	60466176	1.437
67.	s_6^{11}	362797056	8.609
68.	s_6^{12}	-2118184960	51.609
69.	s_3^9	20195	0
70.	s_3^{10}	59561	0

Figure 5.12: Continuation of the Table of Execution Time for the Samples of λ-terms

No.	λ-terms	Output	Execution time(sec)
71.	$s_{3_2}^{11^9}$	177659	0
72.	$s_{3_2}^{12^9}$	531953	0.31
73.	$s_{3_2}^{13^9}$	1594835	0.46
74.	$s_{3_2}^{14^9}$	4783481	0.171
75.	$s_{3_2}^{15^9}$	14349419	0.484
76.	$s_{3_2}^{16^9}$	43047233	1.468
77.	$s_{3_2}^{17^9}$	129140675	4.375
78.	$s_{3_2}^{18^9}$	387421001	13.125
79.	$s_{3_2}^{19^9}$	1162261979	39.359
80.	$s_{3_2}^{20^9}$	0	∞
81.	$s_{3_2}^{18^{11}}$	1162263515	38.796
82.	$s_{3_2}^{18^{12}}$	1162265563	38.812
83.	$s_{3_2}^{18^{15}}$	1162310619	39.218
84.	$s_{3_2}^{18^{19}}$	1166242779	39.39
85.	$s_{5_4}^{4^6}$	4721	0
86.	$s_{5_4}^{7^7}$	94509	0.015
87.	$s_{5_4}^{8^8}$	456161	0.015
88.	$s_{5_4}^{9^9}$	2215269	0.062
89.	$s_{5_4}^{10^{10}}$	10814201	0.281
90.	$s_{5_4}^{11^{11}}$	33022429	1.343
91.	$s_{5_4}^{12^{12}}$	260917841	6.609
92.	$s_{5_4}^{13^{13}}$	1287811989	32.546
93.	$s_{5_4}^{14^{13}}$	1875657193	155.281
94.	sq_2^4	65536	0
95.	$(sq^4)^4$	0	0
96.	$i_3^{18}\ i_2^{11}$	0	38.802
97.	$i_3^{18}\ i_2^{12}$	0	38.913
98.	$i_3^{18}\ i_2^{15}$	0	39.39
99.	$i_5^7\ i_4^7$	0	0.014
100.	$i_5^8\ i_4^8$	0	0.015
101.	$i_5^9\ i_4^9$	0	0.0612

Figure 5.13: Continuation of the Table of Execution Time for the Samples of λ-terms

Chapter 6

Correctness Proof

In order to prove the correctness of the translation we give a formal semantics of the translated code by building a mathematical model of it. The mathematical model is based on the execution of the translated code. First we give a denotational semantics of the typed λ-calculus. Then the correctness of the implementation of the typed λ-calculus by C++ classes is proved with respect to the denotational semantics. The correctness proof of the translated code is based on a Kripke-style logical relation between values (the results of evaluating expressions) and denotations (elements of the model).

The approach of using denotational semantics and logical relation in proving program correctness has been used before by researchers such as Plotkin [Plo77], and many others. The method of logical relation can be traced back at least to Tait [Tai67] and has been used for a large variety of purposes (eg. Jung and Tiuryn [JT93], Statman [Sta85], and Plotkin [Plo80]).

Before we start building a mathematical model of the translated code, we list some of the mathematical preliminaries that will be frequently used in this chapter. The presentation of the proof follows the style of Winskel [Win93].

6.1 Mathematical preliminaries

Mappings

i) If X, Y are sets, then a list $m = (x_1 : y_1), \ldots, (x_n : y_n) \in$ list$(X \times Y)$ is considered as a finite map m from X to Y which is defined as follows: If $x \in X$, $x = x_i$ and $x \neq x_j$ for $j > i$, then $m(x) := y_i$. If $x \neq x_i$ for all $i = 1, \ldots, n$, then $m(x)$ is undefined.

ii) We define dom(m) as the domain of m which is, if m is as above, $\{x_1, \ldots, x_n\}$.

iii) If $x \in X$, $y \in Y$, then $m[x \mapsto y] := m, (x, y)$, i.e.the extension of the list m by (x, y). Since in the way we have defined lists to denote finite functions, $m,\ (x, y)$ will denote the function which maps x to y and all other variables to what they are mapped to by m. Note that $\mathrm{dom}(m[x \mapsto y]) = \mathrm{dom}(m) \cup \{x\}$ and

$$m[x \mapsto y](x') = \begin{cases} y, & \text{if } x' = x, \\ m(x'), & \text{otherwise} \end{cases}$$

6.2 Definition of the Typed λ-calculus

We briefly recall the syntax of the simply typed λ-calculus which was discussed in detail in Chapter 3. The syntax is similar to the one given in Section 3.2.8 (church style), but differs slightly because we have a single base type of integers, and terms include the construct of applying function names to argument terms.

6.2.1 Types

The set **Typ** of types is inductively given by :

i) Int $\in$ **Typ**.

ii) if $A, B \in$ **Typ**, then $A \rightarrow B \in$ **Typ**.

An alternative way of defining the set **Typ** is by means of a recursive domain equation :

$$\mathbf{Typ}{=}\{\mathbf{Int}\}{+}\mathbf{Typ}{\times}\mathbf{Typ}$$

Remark: Note that in the clause ii) of the set **Typ** of types, the "→" is a syntactic symbol of the object language in the domain equation given above. The "+" and "×" are symbols of the metalanguage denoting the set theoretic operations of disjoint sum and cartesian product respectively. The definition of types above is essentially the same as that given in Section 3.2.8. There the λ-calculus was based on base types, of which we use in this chapter only the type Int of integers.

6.2.2 Terms

The **Terms** of the λ-calculus are defined as follows:

i) n is a term ($n \in \mathbf{N}$).

ii) x is a term ($x \in$ **Var**, where **Var** = String).

iii) $r\ s$ is a term, if r, s are terms. (Term r is applied to term s).

iv) $\lambda x : A.r$ is a term, if $x \in$ **Var**, $A \in$ **Typ**, r is a term. (λ-abstraction).

v) $f[r_1 \dots r_n]$ is a term (abbreviated as $f[\vec{r}]$), if $f \in \mathcal{F}$ and r_i are terms. Here $\mathcal{F}$ is a set of names for computable functions on $\mathbf{N}$). The function denoted by f is written as $[\![f]\!]$.

The above can be written as a domain equation as follows:

$$\mathbf{Term{=}N{+}Var{+}Term{\times}Term{+}Var{\times}Typ{\times}Term} + \mathcal{F}{\times}\mathbf{List(Term)}$$

This definition of terms corresponds to the definition of terms given in the page 36. In addition is the fifth term which is a function applied to a list of terms.

6.2.3 Typing

A **Context Γ** is a map from variables to types i.e. a list of variables and their type :

Context=list(Var×Typ)

Contexts will be denoted as $\Gamma = x_1 : A_1, \ldots, x_n : A_n$

The typing rules below correspond to the third style of typing described in Section 3.2.8. The **Typing** rules of the simply typed λ-calculus are :

i)

$$\frac{}{\Gamma, x : A \vdash x : A}$$

ii)

$$\frac{}{\Gamma \vdash n : \mathrm{Int}}$$

iii)

$$\frac{\Gamma, x : A \vdash r : B}{\Gamma \vdash \lambda x : A.r : A \to B}$$

iv)

$$\frac{\Gamma \vdash r : A \to B \qquad \Gamma \vdash s : A}{\Gamma \vdash rs : B}$$

v)

$$\frac{f : \mathrm{Int} \times \ldots \times \mathrm{Int} \to \mathrm{Int} \qquad \Gamma \vdash r_1 : \mathrm{Int} \ldots \Gamma \vdash r_n : \mathrm{Int}}{\Gamma \vdash f[r_1, \ldots, r_n] : \mathrm{Int}}$$

The first rule says that in the context A, the variable x can be given a type A, provided it is assigned this type in the context. The constant n has preassigned type Int. The third rule is for an abstraction where we follow the variant of church-style typing discussed on the page 45. If in context Γ, extended by $x : A$ we have $r : B$, then $\lambda x : A.r$ has type $A \to B$ (in context Γ). For an application term $r\,s$ we have the following rule: if r in context Γ is of type $A \to B$ and s in the same context is of the type A, then the term $r\ s$ has type B in the context. The fifth rule is an additional rule to the typability rules on the page 44. The rule involves a function with a list of arguments or terms, where we assume the list of arguments or terms is of type Int. Then the type of the resulting term is Int.

6.2.4 Denotational Semantics

The sets of **functionals** of type A denoted as $\mathrm{D}(A)$ are defined as follows :

i) $\mathrm{D}(\mathrm{Int}) = \mathbf{N}$

ii) $\mathrm{D}(A \to B) = \{f | f : \mathrm{D}(A) \to \mathrm{D}(B)\}$

iii) $\mathrm{D} := \biguplus_{A \in \mathbf{Typ}} \mathrm{D}(A)$ where $\biguplus$ denotes disjoint union.

A **Functional Environment** is a finite mapping

$$\xi : \mathbf{Var} \to \mathrm{D}$$

We define FEnv := **Var** $\to_{\mathrm{fin}}$ D to be the set of all functional environment. If Γ is a context, then $\xi : \Gamma$ means $\forall x \in \mathrm{dom}(\Gamma).\xi(x) \in \mathrm{D}(\Gamma(x))$.

For every typed λ-term $\Gamma \vdash r : A$ and every functional environment $\xi : \Gamma$ the denotational value $[\![r]\!]\xi \in \mathrm{D}(A)$ is defined as follows:

(i) $[\![n]\!]\xi = n$
(ii) $[\![x]\!]\xi = \xi(x)$
(iii) $[\![r\ s]\!]\xi = [\![r]\!]\xi([\![s]\!]\xi)$
(iv) $[\![\lambda x : A.r]\!]\xi(a) = [\![r]\!]\xi[x \mapsto a]$
(v) $[\![f[\vec{r}]]\!] = [\![f]\!]([\![\vec{r}]\!]\xi)$

An **Implementation** of the typed λ-calculus is an (implementation of an) algorithm computing for every closed term $r : Int$ the value $[\![r]\!] \in \mathbf{N}$.

6.3 Implementation by C++ Classes

As mentioned in the previous chapter, the λ-term that was input to the parser, will be translated to C++ statements which involves the creation of C++ classes for the λ-term. The created classes depend on the λ-term. The more complex the term is the more classes will be created. When the class is instantiated, an address of the class will be stored on the heap. Further instantiations of other classes will create further objects on the heap. Variables will be assigned addresses of the objects created on the heap.

Every class is instantiated by calling the constructor of the object i.e. the name of the class with or without any arguments. The body of the λ-term is associated with the application in the syntactic sets of this translated code. Based on the syntax of this translated code we distinguish each entity of the syntax by grouping them into syntactic sets.

The list of syntactic sets associated with C++ classes is as follows:

- **Addr = Int**
  These are addresses (**Addr**) of classes or variables on the heap.
- **Constr = String**
  An element of (**Constr**) is a constructor or name of a class.

- **Val = Int + Addr**
  A value (**Val**) is either an integer or an address on the heap.

- **App = Int + Var +** $\mathcal{F}\times$ **list(App) + App** $\times$ **App + Constr** $\times$ **list(App)**
  This is the same as the definition of **Term** above.
  An application (**App**) can be any of the following:

  - **Int**
  - **Var**
  - $\mathcal{F}$ **x list(App)** e.g. $f(x, y, z)$
  - **App** $\times$ **App** e.g. $r\ s$.
  - **Constr x list(App)** e.g. new lambda1(x) or new lambda0()

- **Abst** $=$ **Var**$\times$**Typ** $\times$ **Context**$\times$**App**
  An abstraction (**Abst**) corresponds to the body of a class. Classes occurring in this setting consist of instance variables and one method `operator()(A x){ ... }`. They are therefore given by the variable bound by the operator method, the type of that variable, an application term which is the body of the operator method, and a context which describes the instance variables of the class.
  Types such as Int $\to$ Int, which is an arrow type, will be represented in C++ as strings such as `Cint_intD`.

- **Env = list(Var** $\times$ **Val)**
  An environment (**Env**) is a list of variables and their values, or a finite map from variables to values.

- **Heap = list(Addr** $\times$ **Constr**$\times$ **list(Val))**
  Heap consists of a list of addresses of class names (constructors) and a list of values of the instance variables of that class. It is therefore a finite map from addresses to pairs consisting of constructors and a list of values of the instance variables.

- **Class = list(Constr** $\times$ **Abst)**
  The set **Class** of classes consists of list of names of classes (constructors) and the abstraction term describing the body of that class. It is therefore a finite map from constructors to **Abst**.

We assume that every $f \in \mathcal{F}$ is given by a side effect free C++ function.

6.3.1 The Evaluation of the λ-terms in C++

When a λ-term r is evaluated in an environment **Env**, then for all λ-terms s which are λ-abstractions involved in the evaluation of r, elements on the heap (**Heap**) will be created. They will contain the constructor of the class representing the translated λ-term, and values for all the free variables of s. Therefore the evaluation will take an element of **Env**, and an application term (element of **App**), and compute a value (element of **Val**) and an extended heap.

If a value is an address, the meaning of that address will be looked up in the given class environment C:Class.

Thus the functionality of the evaluation function (**eval**) is as follows:

$$\textbf{eval : Class} \rightarrow \textbf{Heap} \rightarrow \textbf{Env} \rightarrow \textbf{App} \rightarrow \textbf{Val} \times \textbf{Heap}$$

In case of function application, where one value is applied to another, again, during the computation, the heap will be extended. So the application function takes a heap, two values, and returns a value and an extended heap. Thus the functionality of the application function (**apply**) is as follows:

$$\textbf{apply : Class} \rightarrow \textbf{Heap} \rightarrow \textbf{Val} \rightarrow \textbf{Val} \rightarrow \textbf{Val} \times \textbf{Heap}$$

Note that the function **eval** and **apply** depend on the class environment, but they do **not** change it. Moreover in the recursive definition of eval and apply, the argument C:Class is not changed in the recursive calls. Therefore we drop the class argument in order to simplify the notation. We write

$$\mathrm{eval}\, \eta\, a \text{ instead of } \mathrm{eval}\, C\, \eta\, a$$

and

$$\mathrm{apply}\, H\, \eta\, a \text{ instead of } \mathrm{apply}\, C\, H\, v\, w$$

.

The reason why the class environment C does not change is that classes are built during the parsing phase only (see Section 6.3.3). In the evaluation phase they are only looked up but not modified.

In presenting the evaluation rules we will follow the convention that

- n range over numbers **N**
- x range over variables **Var**
- a , b range over application **App**
- v , w range over values **Val**
- k range over address **Addr**
- H ranges over **Heap**
- c range over constructor **Constr**
- C range over **Class**
- A, B range over **Type**
- η range over **Env**

The meta variables we use to range over the syntactic categories can be primed or subscripted. For example, H, H', H'', H_k stand for heaps, C, C', C'', C_1 stand for classes and v_1, v' stand for values.

The recursive rules for the evaluation of λ-terms are as follows:

Evaluation of an applicative term which is a number

$$\text{eval}\, H\, \eta\, n = (n, H) \tag{6.1}$$

Thus any number is evaluated to itself without any change to the heap.

Evaluation of an applicative term which is a variable

$$\text{eval}\, H\, \eta\, x = (\eta(x), H) \tag{6.2}$$

Thus a variable evaluates to its content in an environment η without any change to the heap.

Evaluation of an applicative term which is a function with a list of arguments

$$\text{eval}\, H\, \eta\, f[\vec{a}] = ([\![f]\!](\vec{n}), H') \tag{6.3}$$

where

$$(\vec{n}, H') = \text{eval*}\, H\, \eta\, \vec{a}$$

The auxiliary function eval* is defined by

$$\text{eval*}\ H\ \ \eta\ \ (a_1, a_2, \dots a_k) := ((n_1, n_2, \dots, n_k), H')$$

where

$$\begin{aligned} \text{eval}\, H\, \eta\, a_1 &= (n_1, H_1) \\ \text{eval}\, H_1\, \eta\, a_2 &= (n_2, H_2) \\ &\vdots \\ &\vdots \\ \text{eval}\, H_{k-1}\, \eta\, a_k &= (n_k, H') \end{aligned}$$

A function f with a list of arguments evaluates to $[\![f]\!]$ applied to the result of evaluating the arguments. The arguments need to evaluate to numbers, and the evaluation will result in an extended heap H'. H' is unchanged because $f \in \mathcal{F}$ has no side effect.

Evaluation of an applicative term which is the application of one term to the other

$$\text{eval}\, H\, \eta\, (a\, b) = \text{apply}\, H''\, v\, w = (v', H''') \tag{6.4}$$

where

$$\text{eval}\, H \;\; \eta \;\; a = (v, H')$$
$$\text{eval}\, H' \;\; \eta \;\; b = (w, H'')$$

Thus an application of one term a to another term b is evaluated by first evaluating the application term a giving a value v on the extended heap H'. Then the second application term b is evaluated with the value on the heap H' giving a value w on the extended heap H''. Then the function **apply** is used which computes the result of applying v to w.

The definition of **apply** in detail is shown as follows :

$$\text{apply}\, H \;\; k \;\; v = \text{eval}\, H \;\; \eta \;\; a \tag{6.5}$$

where $\quad H(k) = (c, \vec{w}),$

$$C(c) = (x : A; \vec{y} : \vec{B}; a) \qquad (\text{assuming } \; c \in \text{dom}(C))$$

$$\text{dom}\, \eta\, a = \{x, \vec{y}\}$$
$$\eta(x) = v$$
$$\eta(y_i) = w_i$$

So **apply** is only defined if the first value is an address k on the heap. Assume it is, and its class is c, and the values of the instance variables are $\vec{w}$. Assume the class name c is in the domain of the class environment C, and the corresponding class denotes the abstraction $(x \; : \; A; \vec{y} \; : \; \vec{B}; a)$. Then the `operator()` method of this class is to be applied to the second value v (argument of **apply**). This is done by evaluating a in the environment where the variable x is mapped to v, and the instance variables are mapped to the values given on the heap for address k. This will result in an extension of the heap.

Evaluation of λ-term where the applicative term is a constructor with a list of arguments

$$\text{eval}\, H\, \eta\, c[\vec{a}] = (k, H'[k \mapsto c[\vec{v}]) \qquad (k \in \mathbf{Addr}, v \in \mathbf{Val}) \tag{6.6}$$

where $\quad \text{eval*}\; H\, \eta\, \vec{a} = (\vec{v}, H')$

and $\quad k = \text{new}(H') \qquad (\text{new}(H') \text{ is an address not in dom}(H'))$

Thus the evaluation of a constructor c with its arguments will result in first evaluating the arguments of the constructor in sequence. Then a new element is created on the heap with constructor name set to c and the instance variables set to the evaluated arguments of the constructor. The value returned is the new address created on the heap.

In all other cases for the application, it is termed invalid and an error will be returned.

Lemma 1:

i) eval $H\,\eta\,a = (v, H') \Longrightarrow H \subseteq H'$
ii) apply $H\,v\,w = (v', H') \Longrightarrow H \subseteq H'$
iii) eval* $H\,\eta\,\vec{a} = (\vec{n}, H') \Longrightarrow H \subseteq H'$

Proof of i) ii) iii) by simultaneous induction on the definition of **eval** and **apply**.

i) Proof by induction on the definition of **eval**:

Cases :

1) $a = n$ (when the applicative term is a number):
eval $H\,\eta\,n = (n, H)$
2) $a = x$ (when the applicative term is a variable):
eval $H\,\eta\,x = (\eta(x), H)$
3) $a = f[\vec{a}]$ (when the applicative term is a function with a list of arguments):
eval $H\,\eta\,f[\vec{a}] = ([\![f]\!](\vec{n}), H')$
where
eval* $H\,\eta\,\vec{a} = (\vec{n}, H')$
By induction hypothesis (iii), $H \subseteq H'$.
4) $a = a_1, a_2$ (when one applicative term is applied to another applicative term):
eval $H\,\eta\,(a_1\,a_2) = (v', H''')$
where
eval $H\,\eta\,a_1 = (v, H')$,
eval $H'\,\eta\,a_2 = (w, H'')$,
apply $H''\,\eta\,v\,w = (v', H''')$
By induction hypothesis (i), $H \subseteq H' \subseteq H''$ and by induction hypothesis (ii), $H'' \subseteq H'''$
5) $a = c[\vec{a}]$ (when the applicative term is a constructor with a list of arguments):
eval $H\ \ \eta\ \ c[\vec{a}] = (k, H'[k \mapsto c[\vec{n}]])$
where
eval* $H\,\eta\,\vec{a} = (\vec{n}, H')$
$k = \text{new}(H')$
By induction hypothesis (iii), $H \subseteq H' \subseteq H'[k \mapsto c[\vec{n}]]$

ii) Proof of **apply** by giving the detailed definition of apply :

apply $H\,k\,v = (v', H')$

where $k \in \text{dom}(H)$,

eval $H\ (x, \vec{y} \mapsto v, \vec{w})a = (v', H')$

We know : $H(k) = (c, \vec{w})$ and

$C(c) = (x : A; \vec{y} : \vec{B}; a)$

By induction hypothesis (i), $H \subseteq H'$

iii) Proof by induction on the definition of **eval***.

Cases :

1) $\vec{a} = []$ (when the list of arguments is an empty set, i.e, no arguments):
$\quad$ eval* $H\,\eta\,[] = ([], H)$, therefore$H = H' \subseteq H'$

2) $\vec{a} = a : \vec{b}$:
$\quad$ eval* $H\,\eta\,[a : \vec{b}] = ([n : \vec{n}], H'')$
where
$\quad$ eval* $H\,\eta\,a = (n, H')$
$\quad$ eval* $H'\,\eta\,\vec{b} = (\vec{n}, H'')$

By induction hypothesis (i), $H \subseteq H'$and by induction hypothesis (iii, $H' \subseteq H''$.

Thus we conclude Lemma 1.

Recall that the true signatures of eval and apply are as follows :

eval : Class →Heap→Env→App→Val×Heap

apply : Class →Heap→Val→Val→Val×Heap

We write eval$_C$ $H\,\eta\,a$ and apply$_C$ $H\,v\,w$ if the argument C:Class is to be made explicit.

6.3.2 Modelling the Parser-Translator Program

The parser-translator program (PTP) described in Chapter 5 takes as input a string representing a typed λ-term and outputs corresponding C++ class definitions. In order to simplify things and to concentrate on the most important aspects of the problem we assume that the input is given as an abstract term rather than a string. The parsing from a string to a term is a traditional parsing problem which is of no interest here. What is interesting is the process of creating a system of C++ classes that represents a λ-term.

In order to give a recursive description of this process, we must assume that the term in question is not the first term being parsed, but other terms (or subterms) have been parsed before having created a system of classes. Furthermore, if the term has free variables, then the types of these variables must be fixed by an appropriate context. Therefore, the parser **P** (corresponding to the parser-translator program) has the following functionality:

P : Class→Context→Term → App×Class

In the recursion definition of P$_C\,\Gamma\,t$ above, we do a case analysis on the possible forms of the term t:

Parsing when the term is a number:

$$\mathrm{P}_C\,\Gamma\,n = (n, C) \tag{6.7}$$

Thus the parsing of a number will give the value of the number and an unchanged class C.

Parsing when the term is a variable

$$\mathrm{P}_C\, \Gamma\, x = (x, C) \tag{6.8}$$

Thus the parsing of a variable will give the variable and unchanged class C.

Parsing when the term is a function with a list of arguments:

$$\mathrm{P}_C\, \Gamma\, f[\vec{r}] = (f[\vec{a}], C') \tag{6.9}$$

where

$$\mathrm{P}*_C\, \Gamma\, \vec{r} = (\vec{a}, C')$$

Thus the parsing of a term that has a function with a list of arguments will result in the extended class of the function with the list of applications where the list of applications will be parsed recursively first.

The recursive definition of P* is:

$$\mathrm{P}*_C\, \Gamma\, (r_1, r_2, \ldots, r_k) = ([a_1, a_2, \ldots, a_k], C')$$

where

$$\begin{aligned} \mathrm{P}_C\, \Gamma\, r_1 &= (a_1, C_1) \\ \mathrm{P}_{C_1}\, \Gamma\, r_2 &= (a_2, C_2) \\ &\vdots \\ &\vdots \\ \mathrm{P}_{C_{k-1}}\, \Gamma\, r_k &= (a_k, C') \end{aligned}$$

Parsing of an application:

$$\mathrm{P}_C\, \Gamma\, (r\ s) = (a\, b, C'') \tag{6.10}$$

where

$$\begin{aligned} \mathrm{P}_C\, \Gamma\, r &= (a, C') \\ \mathrm{P}'_C\, \Gamma\, s &= (b, C'') \end{aligned}$$

Thus in the case of parsing a λ-term which is an application, the first term will be parsed first giving a resulting term a and an extended class C'. Then the second term with extended class C' (from the parsing of the former) will be parsed giving a resulting term b with extended class C''. The resulting term will be $a\ b$ and the class will be C''.

Parsing of a λ-abstraction:

$$\mathrm{P}_C\, \Gamma\, (\lambda x : A.r) = (c[\vec{y}], C'[c \mapsto (x : A; \Gamma; a)]) \tag{6.11}$$

where $\vec{y} = \mathrm{dom}(\Gamma)$, $\mathrm{P}_C\, \Gamma\, [x \mapsto A]\, r = (a, C')$, and $c =$ new C' meaning that c is a name of a class that is "new" i.e. has not been used before.

Remark : we only generate $c[\vec{x}] \in$ App with $\vec{x} \in$ list(Var) and not $c[\vec{a}]$ with arbitrary $\vec{a} \in$ list(App)

Lemma 2:

i) $\mathrm{P}_C\,\Gamma\, r = (a, C') \quad \Longrightarrow C \subseteq C'$
ii) $\mathrm{P^*}_C\,\Gamma\, \vec{r} = (\vec{a}, C') \quad \Longrightarrow C \subseteq C'$

Proof of Lemma 2 by induction on r respectively $\vec{r}$.

i) Proof by induction on λ-term r.

Cases on the term r:

1) $r = n$ (when the term is a number):
$\mathrm{P}_C\,\Gamma\, n = (n, C)$
2) $r = x$ (when the term is a variable):
$\mathrm{P}_C\,\Gamma\, r = (x, C)$
3) $r = f[\vec{r}]$ (when the term is a function with a list of arguments):
$\mathrm{P}_C\,\Gamma\, f[\vec{r}] = (f[\vec{a}], C')$
where
$\mathrm{P^*}_C\,\Gamma\, \vec{r} = (\vec{a}, C')$
By induction hypothesis (ii), $C \subseteq C'$
4) $r = r_1, r_2$ (when the term is an application):
$\mathrm{P}_C\,\Gamma\, (r_1\, r_2) = (a\, b, C')$
where
$\mathrm{P}_C\,\Gamma\, r_1 = (a, C')$
$\mathrm{P}_{C'}\,\Gamma\, r_2 = (b, C'')$
By induction hypothesis (i) applied twice, $C \subseteq C'$ and $C' \subseteq C''$
5) $r = \lambda x : \mathrm{A}.r$ (when the term is an abstraction):
$\mathrm{P}_C\,\Gamma\, (\lambda x : A.r) = (c[\vec{y}], C'[c \mapsto (x : A, \Gamma; a)])$
where
$\vec{y} = \mathrm{dom}(\Gamma)$, $\mathrm{P}_C\,(\Gamma\,[x \mapsto A])\, r = (a, C')$ and $c =$ new C'
By induction hypothesis (i), $C \subseteq C'$ and because $c \notin \mathrm{dom}(\Gamma), C' \subseteq C'[c \mapsto (x : A, \Gamma; a)]$

ii) Proof by structural induction on $\vec{r}$

Cases on $\vec{r}$:

1) $\vec{r} = []$ (when $\vec{r}$ is an empty list:)
$\mathrm{P^*}_C\,\Gamma\, [] = ([], C)$
2) $r : \vec{r}$:
$\mathrm{P^*}_C\,\Gamma\, f[r : \vec{r}] = ([a : \vec{a}], C'')$
where
$\mathrm{P}_C\,\Gamma\, r = (a, C')$
$\mathrm{P^*}_C\,\Gamma\, [\vec{r}] = ([\vec{a}], C'')$
Thus by induction hypothesis (i), $C \subseteq C \subseteq C'$ and by induction hypothesis (ii), $C' \subseteq C''$

Therefore Lemma 2 is proven.

6.3.3 The Correctness of The Translated Code

The correctness proof of the translated code is based on a Kripke-style logical relation between the C++ representation of the term ($\in$ Val $\times$ Heap) and its denotational value ($\in$ D(A)). The relation is indexed by the class environment C and the type A of the term. Since in the case of an arrow type, $A \to B$, extensions of H and C have to be taken into account, this definition has some similarity with Kripke models. The relation

$$\sim_A^C \subseteq (\text{Val} \times \text{Heap}) \times \text{D}(A) \text{ where } A \in \text{ Typ }, C \in \text{Class}$$

is defined by recursion on A as follows:

$$\begin{aligned} (v,H) \sim_{\text{Int}}^C n : &\iff v = n \\ (v,H) \sim_{A\to B}^C f : &\iff \forall C \subseteq C', \forall H \subseteq H', \forall (w,d) \in \text{Val} \times \text{D}(A) : \\ &\quad\;\; (w,H') \sim_A^{C'} d \Longrightarrow \text{apply}_{C'} H' v w \sim_B^{C'} f(d) \end{aligned}$$

We also set $(\eta, H) \sim_\Gamma^C \xi := \forall x \in \text{ dom}\,\Gamma(\eta(x),H) \sim_{\Gamma(x)}^C \xi(x) \in \text{D}(\Gamma(x))$

Lemma 3:

$$(v,H) \sim_A^C d, C \subseteq C', H \subseteq H' \Longrightarrow (v,H') \sim_A^{C'} d$$

Proof of Lemma 3 by induction on **A**.

Cases:

1) A = Int:
By definition

$$\begin{aligned} (v,H) \sim_{\text{Int}}^C n : &\Longrightarrow v = n \\ &\Longrightarrow (v,H') \sim_{\text{Int}}^{C'} n \end{aligned}$$

2) $A \to B$
Assume

$$\begin{gathered} (v,H) \sim_{A\to B}^C f, \\ C' \subseteq C'', H' \subseteq H'', \end{gathered} \tag{6.12}$$

and let

$$(w,d) \in \text{Val} \times \text{D}(A) : (w,H'') \sim_A^{C''} d$$

We have to show : $\text{apply}_{C''} H'' v w \sim_B^{C''} f(d)$

Since $C \subseteq C' \subseteq C''$ and $H \subseteq H' \subseteq H''$, this holds by the assumption (6.12).

Therefore the Lemma 3 is proven.

Our main theorem, which corresponds to the usual "Fundamental Lemma" or "Adequacy Theorem" for logical relations, reads as follows:

Adequacy Theorem: If η : Env, ξ : FEnv, $\Gamma \vdash r : A, \xi : \Gamma, \text{P}_C \Gamma r = (a, C'), C' \subseteq C'', (\eta, H) \sim_\Gamma^{C''} \xi$, and $H \subseteq H'$, then $\text{eval}_{C''} H' \eta a \sim_A^{C''} [\![r]\!]\xi$

Proof: Let us assume :

$$\eta : \text{Env},\ \xi : \text{FEnv},\ \Gamma \vdash r : A,\ \xi : \Gamma,\ \text{P}_C\Gamma r = (a, C'),$$
$$C' \subseteq C'',\ (\eta, H) \sim_{\Gamma}^{C''} \xi,\ H \subseteq H'$$

We have to show :

$$\text{eval}_{C''}\ H'\ \eta\ a\ \sim_{A}^{C''}\ [\![r]\!]\xi$$

We prove this by induction on the typing judgement $\Gamma \vdash r : A$

Cases :

1) $\Gamma, x : A \vdash x : A$
Since, by definition,
$$\text{P}_C\ (\Gamma, x : A)\ x = (x, C),$$
we know $a = x$ and $C' = C$
Furthermore, by the definition of $(\eta, H) \sim_{(\Gamma,x:A)}^{C''} \xi$,
we know :
$$(\eta(x), H) \sim_{A}^{C''} \xi(x) \tag{6.13}$$
Since $\text{eval}_{C''}\ H'\ \eta\ x = (\eta(x), H')$ and $[\![x]\!]\xi = \xi(x)$,
we have to show :
$$(\eta(x), H') \sim_{A}^{C''} \xi(x)$$
But this follows from (6.13) using Lemma 3 and the fact that $C = C' \subseteq C''$ and $H \subseteq H'$

2) $\Gamma \vdash n : \text{Int}$
Since by the definition of the parser,
$$\text{P}_C\ \Gamma\ n = (n, C),$$
we know $a = n$ and $C = C'$
Since, $\text{eval}_{C''}\ H'\ \eta\ n = (n, H')$
and, by definition $[\![n]\!]\xi = n$
we have to show : $(n, H') \sim_{\text{Int}}^{C''} n$
This hold by definition of $\sim_{\text{Int}}$

3)
$$\frac{\Gamma, x : A \vdash r : B}{\Gamma \vdash \lambda x : A.r : A \to B}$$
Since, by definition,
$$\text{P}_C\ \Gamma\ (\lambda x : A.r) = (c[\text{dom}(\Gamma)], \widetilde{C}[c \mapsto (x : A; \Gamma; a')])$$
where $\text{P}_C\ \Gamma\ [x \mapsto A]\ r = (a', \widetilde{C})$, and
$c = \text{new}\ \widetilde{C}$
We know $a = c[\text{dom}(\Gamma)]$ and $C \subseteq C' \subseteq C''$

where $C' = \tilde{C}[c \mapsto (x : A; \Gamma; a')]$
We have to show :

$$\mathrm{eval}_{C''}\ H'\ \eta\ a\ \sim^{C''}_{A\to B}\ [\![\lambda x.r]\!]\xi$$

Assume

$$\begin{aligned} \mathrm{eval}_{C''}\ H'\ \eta\ a &= (v, \tilde{H}) \\ &= (k, H''[k \mapsto \mathrm{c}[\vec{v}]]) \end{aligned}$$

Hence,

$$\begin{aligned} v &= k \\ \tilde{H} &= H''[k \mapsto \mathrm{c}[\vec{v}]] \end{aligned}$$

where

$$\begin{aligned} \mathrm{eval*}_{C''}\ H'\ \eta\ \mathrm{dom}(\Gamma) &= (\vec{v}, \mathrm{H}'') \\ k &= \mathrm{new}\ H'' \end{aligned}$$

Assume $C'' \subseteq \tilde{\tilde{C}}$, $\tilde{H} \subseteq \tilde{\tilde{H}}$ and $(w, \tilde{\tilde{H}})\ \sim^{\tilde{\tilde{C}}}_{A}\ d$
We have to show that :

$$\mathrm{apply}_{\tilde{\tilde{C}}}\ \tilde{\tilde{H}}\ v\ w\ \sim^{\tilde{\tilde{C}}}_{B}\ f(d)$$

We know

$$f = [\![\lambda x.r]\!]\xi \quad \text{and} \quad d \in \mathrm{D}(A)$$

$$f(d) = [\![\lambda x : A.r]\!]\xi(d) = [\![r]\!]\xi[x \mapsto d]$$

Since $\tilde{\tilde{H}}(k) = \tilde{H}(k) = c[\vec{v}]$
and $\tilde{\tilde{C}}(c) = C'(c) = (x : A; \Gamma; a')$
We have :

$$\mathrm{apply}_{\tilde{\tilde{C}}}\ \tilde{\tilde{H}}\ k\ w = \mathrm{eval}_{\tilde{\tilde{C}}}\ \tilde{\tilde{H}}(x; \mathrm{dom}(\Gamma\mapsto w; \vec{v})\ a'$$

We have to prove that :

$$\mathrm{eval}_{\tilde{\tilde{C}}}\ \tilde{\tilde{H}}(x; \mathrm{dom}(\Gamma) \mapsto w; \vec{v})\ a'\ \sim^{\tilde{\tilde{C}}}_{B}\ f(d) \tag{6.14}$$

Using the induction hypothesis for $\Gamma,\ x : A \vdash r : B,\ \tilde{\tilde{H}}$,

$$\begin{aligned} \eta' &= \eta[x \mapsto w], \\ \xi' &= \xi[x \mapsto d], \\ \Gamma' &= \Gamma[x \mapsto A], \qquad (\text{ where } \mathrm{dom}(\Gamma') = \mathrm{dom}(\Gamma) \cup \{x\}) \\ \mathrm{P}_C\,\Gamma'\,r &= (a', \tilde{C}). \end{aligned}$$

$\tilde{C} \subseteq \tilde{\tilde{C}}$ holds because $\tilde{C} \subseteq C' \subseteq C'' \subseteq \tilde{\tilde{C}}$

We have to show :

$$(\eta', \tilde{\tilde{H}})\ \sim^{\tilde{\tilde{C}}}_{\Gamma'}\ \xi'$$

i) Let $y \in \mathrm{dom}(\Gamma)$, we have to show :

$$(\eta(y), \tilde{\tilde{H}}) \ \sim^{\tilde{\tilde{C}}}_{\Gamma(y)} \ \xi(y)$$

This follows from assumption :

$$(\eta, H) \ \sim^{C''}_{\Gamma} \ \xi$$

and Lemma 3 provided we can show $H \subseteq \tilde{\tilde{H}}$ (since $C'' \subseteq \tilde{\tilde{C}}$ holds by assumption)
Proof of $H \subseteq \tilde{\tilde{H}}$:
We have $H \subseteq H'$, by assumption.
Since $\quad \mathrm{eval}_{C''}\ H'\,\eta\ a = (v, \tilde{H})$,
we have $\quad H' \subseteq \tilde{H}$ by Lemma 1.
Furthermore $\tilde{H} \subseteq \tilde{\tilde{H}}$ by assumption. Hence $H \subseteq \tilde{\tilde{H}}$.

ii) $(w, \tilde{\tilde{H}}) \ \sim^{\tilde{\tilde{C}}}_{A} \ d$ holds by assumption.

All conditions for applying the induction hypothesis are satisfied and we conclude (6.14).

4)

$$\frac{\Gamma \vdash r : A \to B \qquad \Gamma \vdash s : A}{\Gamma \vdash r\ s : B}$$

By definition,

$$\mathrm{P}_C\,\Gamma(r\ s) = (e\,b, \tilde{\tilde{C}})$$

where

$$\mathrm{P}_C\,\Gamma r = (e, \tilde{C})$$

$$\mathrm{P}_{\tilde{C}}\,\Gamma s = (b, \tilde{\tilde{C}})$$

we know that $a = e\,b$ and $C' = \tilde{\tilde{C}}$
Hence, $C \subseteq \tilde{\tilde{C}} \subseteq C''$
Assume

$$(\eta, H) \ \sim^{C''}_{\Gamma} \ \xi$$

We have to show

$$\mathrm{eval}_{C''}\ H'\,\eta(e\,b) \ \sim^{C''}_{B} \ [\![r]\!]\xi([\![s]\!]\xi) \tag{6.15}$$

Since, by definition ,

$$\mathrm{eval}_{C''}\ H'\,\eta(e\,b) = \mathrm{apply}_{C''}\ \tilde{\tilde{H}}\ v\ w$$

where

$$\mathrm{eval}_{C''}\ H'\,\eta\ e = (v, \tilde{H})$$

$$\mathrm{eval}_{C''}\ \tilde{H}\,\eta\ b = (w, \tilde{\tilde{H}})$$

and

$$[\![r\ s]\!]\xi = [\![r]\!]\xi([\![s]\!]\xi)$$

Hence we have to show :

$$\text{apply}_{C''}\ \widetilde{\widetilde{H}}\ v\ w\ \sim_{B}^{C''}\ [\![r]\!]\xi([\![s]\!]\xi) \tag{6.16}$$

We use the induction hypothesis for $\Gamma \vdash\ r : A \to B$ and $\mathrm{P}_C\Gamma r = (e, \widetilde{C})$
$\widetilde{C} \subseteq C''$, (holds because $\widetilde{C} \subseteq \widetilde{\widetilde{C}} \subseteq C''$ by Lemma 2)
$(\eta, H)\ \sim_{\Gamma}^{C''}\ \xi, H \subseteq H'$
thus, $\text{eval}_{C''}\ H'\ \eta\ e\ \sim_{A\to B}^{C''}\ [\![r]\!]\xi$
We got :

$$(v, \widetilde{H})\ \sim_{A\to B}^{C''}\ [\![r]\!]\xi$$

We use the definition of $\sim_{A\to B}^{C''}\ [\![r]\!]\xi$,
we know $f = [\![r]\!]\xi$,
$C' \subseteq C'', \widetilde{H} \subseteq \widetilde{\widetilde{H}}$ hold by Lemma 1, and $(w, \widetilde{\widetilde{H}})\ \sim_{A}^{C''}\ [\![s]\!]\xi$ is proved by induction hypothesis for $\Gamma \vdash s : A$
Hence, we conclude that (6.16) holds.

5)

$$\frac{f : \text{Int} \times \ldots \times\ \text{Int} \to \text{Int} \qquad \Gamma \vdash r_1 : \text{Int}, \ldots, \Gamma \vdash r_n : \text{Int}}{\Gamma \vdash f[r_1 \ldots r_n] : \text{Int}}$$

By definition

$$\mathrm{P}_C\, \Gamma\ f[\vec{r}] = (f[\vec{e}], \widetilde{C}) \tag{6.17}$$

where

$$\mathrm{P*}_C\, \Gamma\, \vec{r} = (\vec{e}, \widetilde{C})$$

The detail definition of $\mathrm{P*}_\mathrm{C}\, \Gamma\, \vec{r} = (\vec{e}, \widetilde{\mathrm{C}})$ are :

$$\mathrm{P}_C\, \Gamma\, r_1 = (e_1, C_1)$$

$$\mathrm{P}_{C_1}\, \Gamma\, r_2 = (e_2, C_2)$$

$$\vdots$$

$$\vdots$$

$$\mathrm{P}_{C_{k-1}}\, \Gamma\, r_k = (e_k, C_k)$$

We know that $C_k = \widetilde{C}$, hence $C_i \subseteq \widetilde{C} \subseteq C''$
We know from (6.17), $a = f[\vec{e}], C' = \widetilde{C}$, hence $C \subseteq \widetilde{C} \subseteq C''$
Assume

$$(\eta, H)\ \sim_{\Gamma}^{C''}\ \xi$$

we have to show:

$$\text{eval}_{C''}\ H'\ \eta f[\vec{e}]\ \sim_{\text{Int}}^{C''}\ [\![f]\!]([\![r]\!]\xi) \tag{6.18}$$

Since, by definition

$$\text{eval}_{C''}\ H'\ \eta f[\vec{e}] = ([\![f]\!](\vec{n}), H'')$$

where

$$\text{eval*}_{C''}\ H'\ \eta\vec{e} = (\vec{n}, H'')$$

H'' unchanged because $f \in \mathcal{F}$ has no side effect.

The definition

$$\text{eval*}_{C''}\ H'\ \eta\vec{e} = (\vec{n}, H'')$$

is elaborated as follows :

$$\text{eval}_{C''}\ H'\ \eta e_1 = (n_1, H'_1)$$
$$\text{eval}_{C''}\ H'_1\ \eta e_2 = (n_2, H'_2)$$
$$\vdots$$
$$\vdots$$
$$\text{eval}_{C''}\ H'_{k-1}\ \eta e_k = (n_k, H'_k)$$

where $H'_k = H''$
Hence $\mathrm{H} \subseteq \mathrm{H}' \subseteq \mathrm{H}'_1 \subseteq \mathrm{H}'_2 \subseteq \ldots \subseteq \mathrm{H}'_k$
Therefore, by induction hypothesis ,

$$\text{eval}_{C''}\ H'_i\ \eta e_i\ \sim^{C''}_{\text{Int}}\ [\![r_i]\!]\xi$$

that is

$$(n_i, H_i)\ \sim^{C''}_{\text{Int}}\ [\![r_i]\!]\xi$$
$$\Longrightarrow n_i = [\![r_i]\!]\xi$$

We have to show :

$$\text{eval}_{C''}\ H'\ \eta f[\vec{e}]\ \sim^{C''}_{\text{Int}}\ [\![f]\!]([\![r]\!]\xi)$$

Since, by definition of,

$$([\![f]\!](\vec{n}), H'')\ \sim^{C''}_{\text{Int}}\ [\![f]\!](\vec{n})$$

and

$$[\![f]\!](\vec{n}) = [\![f]\!](\vec{n})$$

Therefore (6.18) holds.

This completes the proof of the Adequacy Theorem.

Corollary (Correctness of the implementation):

If $\vdash r\ :\ \text{Int}, \mathrm{P}_C\, r = (a, C'), C' \subseteq C$", then for any heap H, $\text{eval}_{C''}\ H\ \eta\ a = ([\![r]\!], H')$ for some $H' \supseteq H$

Proof: This is a special case of the Adequacy Theorem with $\Gamma = \emptyset$. Note that $(\eta, H)\ \sim^{C''}_{\emptyset}\ \xi$ holds trivially.

Chapter 7

Related Work

It has been discovered by several researches[Kis98], [Lau95] that C++ can be used as functional programming by representing higher-order functions using classes. Our representation in the translated code is based on similar ideas. There are other approaches that have made C++ a language that can be used for functional programming such as the FC++ library [MS00], FACT! [SS00], [FA00], Lambda Library [JP00], Funk library [Hal02] and creating macros that allow creation of single-macro closure [Kis98]. We will discussed briefly these approaches below.

There are other fragments of object-oriented languages in the literature which are used to prove the correctness of programs such as the well known Featherweight Java [AIW99]. The model of this language avoids the use of a heap, since methods do not modify instance variables. However our model of C++ does make use of a heap which is closer to the actual implementation of C++.

7.1 FC++ Library

FC++ is a library for doing functional programing in C++. The library comprises of a general framework or functoids and about 100 common/useful functions. FC++ is claimed to be different from other libraries which provide either syntax support (such as "lambda" operator for anonymous functions) or a framework for expressing higher-order function-type [MS03] due to its powerful type system. FC++ offers complete support for manipulating polymorphic functions where passing them as arguments to other functions and returning them as results. For example FC++ supports higher order polymorphic operators such as `compose()` which is a function that takes two arguments(possibly polymorphic) and returns a possibly polymorphic result.

FC++ also can be used to embed a lot of the capabilities of modern functional programming languages (such as Haskell or ML) in C++. It also comes with a lot of useful predefined function which is a large part of the Haskell Standard Prelude and supports lazy evaluation. It has lazy list data structure and several functions that operate on this lazy list. It has a number of support functions for transforming FC++ data structure into data structures of the

```
#include<assert.h>
#include<string>
#include "prelude.h"

int main(){
  int x=1, y=2, z=3;
  std:string s="foo", t= "bar", u="qux";

  List<int> li = cons(x, cons(y,cons(z,NIL)));
  List<std:string> ls = cons(s,cons(t,cons(u,NIL)));

  assert( head(li) == 1);
  //list_with makes short_list
  assert( tail(li) == list_with(2,3));

  ls = compose(tail,tail)(ls);
  assert( head(ls) == "qux");
  assert( tail(ls) == NIL);
  }
```

Figure 7.1: List and compose

C++ Standard Template Library and vise versa. Also, it has operators for promoting normal functions into FC++ functoids and supplies indirect functoids i.e. runtime variables that can refer to any functoid with a given monomorphic type signature.

FC++ implementation relies heavily on C++ templates and the C++ type system. It does not focus on improving the syntax using either the preprocessor (eg., `#define`) or overloading techniques (eg., expression templates). Its value lies on its type system for polymorphic function providing a nicer syntactic front-end for defining functions.

An example [MS01] of manipulations of list written in C++ using the FC++ library is shown in Figure 7.1

The example given in Figure 7.1 demonstrates the capabilities of FC++ manipulating polymorphic functions. The List is parametrized by the type of its elements where in the Figure 7.1, we see both the list of integers and strings. The `tail()` function takes a "list of T" and returns a "list of T" where T can be of any type. `compose(f,g)` yields a new function `h` such that `h(x)` is the same as `f(g(x))`. The `compose` operator composes two unary functions where it can take polymorphic functions as parameters and return a polymorphic function as a result. As a result, `compose(tail,tail)` is a polymorphic function with the same signature as `tail`. FC++ lists are lazy: elements of a list is evaluated only when they are requested. Operations can be performed lazily on the list such as using the function `filter()` defined in the library. For example:

```
List<int> evens = filter(even, integers);
```

creates a list of even integers. `even` is another function defined in FC++.

FC++ functoids supports currying. For example plus is curryable i.e. plus(2) yields a new function f(x), where f(x) = 2 + x. Currying is supported by the FC++ operators that are themselves (higher-order polymorphic) functoids. Using the operator (eg. ptr_to_fun()) can transform regular C++ functions or methods into functoids so that they can be used with the predefined functionality, including higher-order operators like currying and compose.

The functoids that we have seen are direct functoids because call to them are statically bound. FC++ also supports indirect functoids through the FunN classes. These functoids are dynamically bound and thus can change their "function values" by assignmnent. Indirect functoids are described by their monomorphic type signature and variables of type FunN can be bound to any function with the right signature. For example, Fun1<int,bool> describes a one argument function that takes an int and returns a bool, whereas Fun2<int, int,string> describes a two-argument function which takes two ints and return a string. The function makeFunN() converts a direct functoid into an indirect one. More examples of the use of FC++ library in [MS01] and [MS00].

FC++ allows higher-order polymorphic function types to be expressed and used; type signature are explicitly declared unlike Haskell and ML where types can be inferred. The type language (building blocks for Sig template classes) is awkward eventhough it will not be a problem in learning to use it. There is a bound in the number of arguments that the functoids can support but it can be remedied by adding templates with more parameters in the framework. The naming of the base classes in FC++ like Fun1 and FunImpl, as well as operators makeFun1 and Fun1Impl encode in their names the number of arguments of the functions they manipulate.

Compiler error messages can be verbose when a user of FC++ makes a type error where the compiler typically reports the full template instantiation stack, resulting in many lines of error messages. Another limitations to FC++ is that it cannot fully prevent side effects in user code. Nevertheless, by declaring a method to be const can prevent it from modifying the state of the enclosing object. This is what FC++ try to enforce in order to have "side-effect freedom". Even though the indirect functoids are side effect free because any class inheriting from FunNImpl classes have to have a const operator(), but users could decide to add methods which is not side effect free to the subclass of FunNImpl.

7.2 FACT!

FACT! (Functional Addition to C++ through Templates and Classes) is a C++ library that offers several aspects of functional programming to C++ programmers. It provides methods to get curried representations of C++ functions/class member functions, functional composition, λ-expression, and has basic support for lazy evaluation. Through currying FACT! allows for partial application of C++ functions making it possible to pass less than n arguments to a n-ary function giving a valid result.

The currying approach of FACT! offers a more consistent and flexible way to bind arguments of a function to some specific values. Template libraries such as STL contain several generic algorithms that expect functions as arguments (higher-order functions), resulting in a frequent use of function objects. User-defined functions are awkward because they need to be declared as a class in namespace scope before being used. The point of use and the point of definition may get more and more dispersed making code harder to read and understand. Using FACT! λ-notation point of use and point of definition can be kept close together. Thus functions can be define on the fly which is common in functional programming languages.

The `lambda` function takes a list of variables which is called the λ-list, an expression (called λ-expression) that can contain any of this list of variables and returns a function which usually has the same number of arguments as the elements in the λ list. For example:

```
lambda(x, y, x + y)
```

where `x`, `y` formed the λ list, and `x + y` is the λ-expression. A binary function is returned from the `lambda` function since the λ list has two members. Functions returned by `lambda` are polymorphic, thus `x` and `y` may be bound to values of type `int`, `complex`, `string` or any other type that is compatible with the λ-expression. λ-expressions may contain calls to other functions, for example:

```
lambda(x, y, z, sqrt(sqr(x) + sqr(y) + sqr(z)))
lambda(x, y, sin(x)/cos(y))
```

λ-variables may be bound to functions and `lambda` functions may return a function, which in turn will return a function as well, which are shown as follows:

```
lambda(f, x, y, f(x, y))// f is a placeholder for a function
lambda(x, lambda(y, x + y))
```

Functions returned by `lambda` are presented in curried form, making them capable of taking arguments one at a time and thereby offers the opportunity of partial application. Expression templates [Vel95] are a way to handle λ-expression. Expression templates are nested template structures, used to represent the parse tree of an expression. They are built during compile time through overloaded arithmetic operators which instead of immediately applying an operation, it returns objects that incrementally build up the parse tree. The parse tree is represented as a type tree (expression template tree) and as a tree of objects (the expression object which is an instance of expression template tree).

λ-variables become part of the the expression template tree by using the expression template technique. The expression template tree emphasizes types, so different λ-variables must be of different types enabling template meta programs to do the substitution during compile time. Thus λ-variables need to be of unlimited types where ARG is a suitable representation because it can be used to form `numeric_limits<int>::max()` of different types. The structure of ARG is as follows:

```
template <int>
struct ARG {};
```

FACT! has a large number of predefined λ-variables which are defined in the scope of namespace LAMBDA. A user just writes using LAMBDA::x to make the λ-variable x visible in the current scope. Expressions templates can be formed out of expressions containing instances of ARG by using PETE (Portable Expression Templates Engine). PETE allows for the easy integration of expression template functionality to user defined classes. By building λ-expressions on top of PETE, the user can use his expression template functionality aware classes within a λ-expression by still taking benefit of all the related optimization. More on building λ-expression with PETE can be seen in [SS00]. Even though the Curry function is claimed to be powerful as its functional counterpart, there are still limitations with λ-expression and lazy evaluation.

7.3 Lambda Library (LL)

The Lambda Library (LL) is a C++ template library implementing a form of λ-abstraction for C++. It is designed to work with the Standard Template Library(STL) which is now a part of the C++ Standard Library. Therefore the library does no language extensions or preprocessing. The LL consists of rich set of tools for defining unnamed functions which works with the STL algorithms. It offers significant improvements in terms of generality and ease of use compared to the binders and functors in the C++ standard library. We will show some examples of the use of LL taken from [JP00].

- Initialize the elements of a container to the value 1:

  ```
  list<int> v(10);
  for_each(v.begin(), v.end(), _1 = 1);
  ```

  The example above _1 = 1 creates a λ-function which assigns the value 1 to every element in v. The variable _1 is a placeholder with an empty slot which will be filled with a value at each iteration. We call _1 = 1 a λ-expression and a function object created by a λ-expression is a λ-functor.

- Create a container of pointers and make them point to the elements in the container v.

  ```
  list<int*>  vp(10);
  transform(v.begin(), v.end(), v.begin(), &_1);
  ```

  The address of each element in v(with &_1) are assigned to the corresponding element in vp.

- For each element in v, a function foo is called, passing the original value of each element as an argument to foo.

  ```
  int foo(int);
  for_each(v.begin), v.end(), _1 = bind(foo, _1));
  ```

- The elements of vp are sorted and output:

  ```
  sort(vp.begin(), vp.end(), *_1 > *_2);
  for_each(vp.begin(), vp.end(), cout<< *_1 <<endl);
  ```

> The call to sort, sorts the elements by their contents in descending order. The λ-expression `*_1 > *_2` contains two different placeholder `_1` and `_2` creating binary λ-functor. When this functor is called the first argument is substituted for `_1` and the second argument for `_2`. Finally the sorted content of vp is output.

In λ-calculus and in functional programming languages, the formal parameters are commonly named within the λ-expression such as:

$$\lambda x, y.x + y$$

But the LL counterpart of the above expression is written as `_1 + _2` where the placeholder variables have predefined names. The use of a placeholder variable in an expression implicitly turns the exppression into λ-expression. There is no explicit syntactic construct for λ-expression. The LL supports the placeholders `_1, _2` and `_3` which means λ-functors can take not more than three arguments passed in by STL algorithm and zero parameter is possible too. The third placeholder is a necessity in order to implement all the features of the current library.

The LL provides typedefs for the placeholder types, making it easy to define the placeholder names to your liking. A placeholder leaves the argument totally open, including the type, meaning that the lambda functor can be called with arguments with any type for which the underlying function makes sense. Since the type of the placeholder remains open, the return type of the λ-functor is not known either. The LL has a type deduction system that figures out the return type when the λ-functor is called where it covers operators of built-in types and operators of user-defined types.

For an ordinary function call, an explicit syntactic construct is needed. In this case the `bind` function template serves the purpose. The syntax of λ-expression created with the `bind` function is :

```
bind(target-function, bind-argument-list)
```

In a bind expression, the `bind-argument-list` must be a valid argument list for `target-function`, except that any argument can be replaced with a placeholder, or, generally, with a λ-expression. When a placeholder is used in place of an actual argument, the argument is said to be unbound. The `target-function` can be a pointer to function, a reference to a function or a function object. Examples of bind expression [JP00] is shown as follows. Suppose `A, B, C` and `X` are some types:

```
X foo(A, B, C); A a; B b; C c;
....
bind(foo, _1, _2, c);
bind(&foo, _1, _2, c);
bind(foo, _1, _1, _1);
bind(_1, a, b, c);
```

The first and second bind expression returns a binary λ-functor but the second bind expression uses a function pointer instead of a references. For the third bind expression, the argument will be duplicated in each place the placeholder is used, and for the expression to make sense and to compile, the argument to the resulting unary λ-functor must be implicitly convertible to `A, B` and `C`. The fourth bind expression shows the case where the target function is left unbound where the resulting λ-functor takes one parameter, the function to be called with arguments `a, b` and `c`. More examples of bind expression with member functions as targets and other uses of the overloaded operators in LL are shown in [JP00].

Even though LL overloaded almost every operator for λ-expressions based on the basic rule that any operand of any operator can be replaced with a placeholder or with a λ-expression, there are some special case and restrictions; the return types cannot be chosen freely while overloading operators `->, new, delete, new[]` and `delete[]`, thus these cannot be overloaded directly for λ-expressions; it is not possible to overload the `., *`, and `?:` operators in C++; the assignment and subscript operators must be defined as member functions which creates some asymmetry to λ-expressions (eg. `int i; _1 = i;` is valid λ-expression but not `i = _1`); the return type deduction system may not handle all user-defined operators.

The Lambda Library (LL) allows generic function objects to be defined on the fly. This library does not focus on functional programming style, rather it emphasizes on imperative programming allowing multiple assignments, while loops, and several imperative constructs within an expression that defines a function object. The LL does not have support for n-arity functions, because it is meant to be used with STL algorithms which do not accept ternary functions. It only supports for the generation of nullary, unary, binary and ternary function objects. However the LL provides good means to define even very complex function objects through expressions.

7.4 Kiselyov's Functional Style in C++

The definition of a local class, within a function, method or block is permitted in C++ where this feature makes nested functions and closures possible. Nested functions and nested methods are actually compiled inline unless they are virtual. A local class follows regular lexical scoping rules. For example, a variable of an outer block can be declared visible or modifiable within the inner scope. Also, to some extent, a local scope can be captured and a closure is return as the value of a function. Returning an object as the result of a function involves deep copying of the object to and from temporaries which can be costly for big objects such as matrices and images. Therefore an alternative to this is the lazy construction where objects themselves are never returned from functions instead yield a "recipe" on how to make an object. The construction of the object will occur later when it is needed. For example [Kis98]:

```
Matrix haar = haar_matrix(5);
```

`harr_matrix` is a class not a simple function. It constructs an object `Lazy_matrix`. A special constructor `Matrix(const Lazymatrix& recipe)` follows the recipe and makes the matrix haar right in place without any intermediary temporaries.

The following code segment [Kis98] is the representation of λ-expression which is C++ standard-compliant.

```
main(void){
MakeTestFunction("cos(x) - x",
                Lambda((const double x), double,
                return cos(x)-x)) fcos;
//function is instantiated
fcos.run(2.0,3.0); //run the function with two values

MakeTestFunction("HUMPS function zerodemo.m",
                  Lambda((const double x), double,
                  return 1/(sqr(x - 0.3) + .01)
                         + 1/(sqr(x - 0.9) + .04)
                         -6))().run(0.4,1.6,1.299954968);
//function is instantiated and run.
}
```

MakeTestFunction is a subclass of ATestFunction which has a method runfor running a test and making sure the result is correct. Both these functions and Lambda are defined in the LinAlg: a Numerical Math Class Library [LA96]. MakeTestFunction has arguments that consists of:

i) the title of the test case
ii) the test's body itself specified as anonymous function Lambda (genuine λ-abstraction)

Lambda consists of three arguments: input argument, return type and the body of the function/abstraction. In the code segment above, examples are given in testing two computations titled "cos(x) -x"' and "HUMPS function zerodemo.m" and these computations are defined by Lambda. The whole MakeTestFunction clause is subsequently instantiated and run.

Kiselyov introduces the features of closures, late binding and λ-abstraction in incorporating the functional style in C++.

7.5 Funk: A Framework for Functional Style in C++

All Funk code is based around evaluating or aggregating other expression templates (ET). ETs are template instantiations that represent recursively constructed expressions. All ETs are formed around atomic ET variables or ETs that contain C++ function pointers. ET variables are defined by instantiations of struct template ETvar which needs two parameters for instantiation: name variables and type variable. Name variables are just a character template parameter which are limited to a single character only. The definition of ETvar is as follows:

```
template<char n, classT>
struct ETvar{ };
```

Variables can be define using the definition of `ETvar`. For example defining variable a of type `int` is:

```
ETvar<'a',int> a;
```

ET can also define algebraic expressions, for example an ET `plus` can be define by combining two other ETs:

```
template<class LHT, class RHT>
struct plus{};
```

where LHT and RHT are expression templates.

An algebraic expression $(a + b) \times (c + d)$ are defined as:

```
times<plus<ETvar,<'a',int>,
           ETvar<,b,,int>>,
   plus<ETvar<'c',int>,ETvar<'d',int>>>
```

where `times` is an ET that is defined similar as `plus`.

Funk implements partial application of functions using the types `lambda` and `apply` and several utility metaprograms. `apply` is used to hold the values of arguments to which functions have been applied, whereas `lambda` is used to specify the need for and type of a function's parameter. The definition of `lambda` and `apply` is:

```
template<class V, class ET> lambda{ };
template<class V, class ET> apply{ };
```

Any Funk λ-expression that has type `lambda<A,ET>` corresponds to the λ-expression $\lambda a.et$ where a has type `A` and et has type `ET`. Expression template can be turned into λ-expressions by embedding them into a series of instantiations of `lambda` template. Thus the following structure represents the λ-expression $\lambda a.(\lambda b.(a + b))$:

```
lambda<ETvar<'a',int>'
    lambda<ETvar<'b',int>,
       plus<ETvar<'a',int>,
         ETvar<'b',int>>>>
```

`lambda`'s first template parameter holds information about the variable it manages and the body of the λ-expression is represented by the second template parameter. When a Funk λ-expression is partially applied to an arguments, the resultant type of the application is `apply` instantiated with the same argument as the former `lambda` is instantiated with. Thus if a λ-expression given above is applied to an argument, its type would become:

```
apply<ETvar<'a',int>, //note the apply instead of the lambda
```

```
  lambda<ETvar<'b',int>,
       plus<ETvar<'a',int>,
         ETvar<'b',int>>>>
```

and after application to another argument, its type becomes:

```
apply<ETvar<'a',int>,
  apply<ETvar<'b',int>,
       plus<ETvar<'a',int>,
         ETvar<'b',int>>>>
```

Since the type of this λ-expression is fully applied, it must be reduced to the λ expression's ET's final type i.e. `int` in order to be usable for computation. There are mechanisms that perform type translations of Funk λ-expressions into their applied state [Hal02]. An example of a λ-expression applied to two values 3 and 4:

$$(\lambda a, b.a + b)3\,4$$

is represented using expression templates is given as follows:

```
apply_lambda_to_arg(
 apply_lambda_to_arg(
  lambda<ETvar<'a',int>,
         lambda<ETvar<'b',int>,
           plus<ETvar<'a',int>,
              ETvar<'b,',int>,3),4)
```

How the application of the λ-expression is evaluated is shown in [Hal02].

Funk has a type resurrection system which is set up by making the superclass actually a structure template with one template parameter. All expression templates inherit from this superclass instantiated with the type of the expression template. When the type of an expression template is sliced[1] upon being passed as an argument to a restrictive function template, it's original type can be resurrected from the template parameter of the sliced object. Thus the necessary parts of the system is redefined to allow type resurrection.The base structure is defined as :

```
template<class>
struct ET{ };
```

`ETvar` and `plus` are redefined so they derive themselves from `ET<T>` [Hal02]. To preserve data information and type information, but still have the class match as a superclass, an argument type of `ET<T>&` must be used for the parameters of function and operator templates that will resurrect type. When an argument is a reference, slicing only affects the object's type and not its data allowing casting the argument back to its original type and still retain its data integrity. The definition of operator +:

```
template<class E1, class E2>
```

[1] slicing occurs when data members exclusive to a subclass get truncated as an object instance is cast as its superclass

```
plus<E1,E2> operator + (ET<E1>& lhs, ET<E2>& rhs){
 return plus<E1,E2(static_cast<E1&>(lhs),
                   static_cast<E2&>(rhs));
```

Thus by the definitions above things like $a + b$ can be written and the result will be an expression template. λ-expressions are created by making comma separated lists of parameters followed by the `--` and `>` operators followed by the expression template representing the body of the Funk λ-expression. The following code is legal C++ code once the Funk libraries are loaded.

```
x-->(x*x);
(a, b, c)-->((a+b)/c);
```

Functions are partially applied arguments through the use of operator `<<`.

For example:

```
((x,y)-->((x+y)/(x*y))) <<3 <<4;
```

Funk currently does not offer support for polymorphic types in expression templates. Eventhough Funk does provide some nice feature to the C++ programmer, it has its limitations. Naming an expression template is not possible without stating its entire type in the declaration. It is possible to state an expression's template type, but it's not worthwhile because typenames for ETs get very complicated very fast.

7.6 Comparisons

What have been discussed above are approaches that are related to our project where advantages and limitations of these approaches are also given. These works are too extensive in comparison with our project since our project is still new and "young". Many things have not been covered such as polymorphic higher-order functions that have polymorphic arguments (in FC++ library) and type inference. But one thing we can say is that the novelty of our project is that it provides a correctness proof that is lacking in all of the approaches. Our project provides a simple way of creating a function on the fly (λ-expression) with a syntax that is easily read and understand which is similar to the usual λ-notation. This λ-expression is translated into C++ statements that can be compiled with any C++ compiler. We uses the usual C++ statements without overloading any operator to define an anonymous functions.

To compare other approaches discussed above with our approach in defining a λ-term, we will compare them based on an example. The example is a simple λ-term:

$$(((\lambda\ x.\lambda\ y.\lambda\ z.x + y * z)3)2)5$$

- FC++ Library:

  ```
  #include "prelude.h"
  #define FCPP_ENABLE_LAMBDA
  //to get the lambda portion of the library
  LambdaVar<1>x; //declare  variable x
  ```

```
LambdaVar<2>y; //declare variable y
LambdaVar<3>z; //declare variable z
cout<<lambda(x,y,z)[multiplies[plus[x,y],z](3,2,5);
```

`LambdaVar` is for declaring variables used in the λ-expression.

`lambda(LambdaVar)[lambdaExp]` creates a lambda on the fly.

- Lambda Library:

  ```
  int x = 3;
  int y = 2;
  int z =5;
  _1 = x;
  _2 = y;
  _3 = z;
  _1 + _2 * _3;
  ```

  It makes use of a placeholder for a variable and the Lambda Library only supports 3 placeholders, meaning that λ-functors cannot take n-arity arguments which is quite difficult if we want to have nested λ-terms. We can see here Lambda Library makes use of the imperative way in defining the λ-terms where assignments are used and it is not side effect free proof.

- FACT!:

  ```
  lambda(x, y, z, x+y*z) (3,2,5);
  ```

  The λ-expression is handled by expression templates which are used to represent the parse tree of the expression. As mentioned previously (section 7.2), PETE is used to form expression templates from expression containing instances of `ARG` and the evaluation of the application is also done by PETE. Users of FACT! are not required to know about PETE.

- Funk

  ```
  ((x,y,z)-->(x+y*z)) <<3 <<2 <<5;
  ```

  One of Funk's goal is to become a self contained sublanguage of C++. Operator > converts the list created by the comma operator and an expression template into an actual Funk λ-expression.

Using our syntax in writing the λ-expression above is given as follows:

```
(\int x.\int y.\int z.int x+y*z)^^3^^2^^5;
```

FACT! and Funk make use of template meta programming for their λ-expression, and the difference between the two; Funk has its own syntax for λ-expression where the operators involved in the expression are overloaded in the Funk library using expression templates. FC++ and Lambda Library make use of C++ templates in their implementation. Comparing other approaches with our approach in writing the λ-term, we can say that our way is much simpler is much closer to the original λ-term. Even though the type of the function is explicitly declared, the type checking is done by the C++ type system. The translation of the λ-term uses classes and inheritance in an essential way which is simpler and easier to understand.

The other approaches deal only with simple λ-terms, but are they much more efficient, since they don't use inheritance, so the λ-terms are not dynamically generated. All these approaches have been optimised for performance rather than for generality. The other advantage of those approaches is that they don't require an extension of C++ but are just a library used in addition to C++. In general it is always desirable to add new feature by using libraries rather than by extending a language, since each new language extension makes the language more complicated, and the language of C++ is already a rather complex language.

Chapter 8

Summary and Outlook

In this chapter we will summarize our project and give some considerations for future work in extending and improving our project.

8.1 Summary

The objective of this project was to extend C++ language in order to enhance its existing support for different paradigm such as object-oriented, procedural and generic programming. The additional support that we implemented is functional programming. We developed a parser-translator program that translates typed λ-terms into C++ classes so as to integrate functional concepts into C++.

We introduced a syntax for representing λ-types and -terms in C++. In that extended language, we write `A -> B` for the function type $A \rightarrow B$, `r^^s` for the application of r to s, and `\A x.B s` for $\lambda x^A.s$ if $s : B$. We use functional style notation rather than overloading existing C++ notation, since we believe that this will improve readability and acceptability by functional programmers. The λ-abstraction is interpreted as a function of its free variables in the form `(new T(`$x_1, \ldots x_n$`))`. Hence, the evaluation of a λ-abstraction in an environment for free variables is similar to a "closure" in implementations of functional programming languages.

The translated code uses the object-oriented approach of programming that involves the creation of classes for the λ-term. By using inheritance, we achieved that the translation of a λ-abstraction is an element of a function type. A λ-abstraction is represented as a new instance of its corresponding class. Even if the classes for two occurrences of the same λ-abstraction coincide, for each occurrence a new instance is created. Therefore, if a variable occurs as the same name, but with different referential meaning in two identical λ-expression, it will not be a problem.These features have been tested on several λ-terms.

The correctness of our implementation is proved with respect to the usual (set-theoretic) denotational semantics of the simply typed λ-calculus and a mathematical model of a sufficiently large fragment of C++. The proof is based on a Kripke-style logical relation between C++ values and denotational values.

We model only the fragment of C++ that is involved in the translation of the simply typed λ-calculus. We assume that classes translated for the λ-term have instance variables, one constructor, and one method which corresponds to the `operator()` method. The method has one argument, and the body consist of an applicative term. Therefore, a class is given by a context representing its instance variables, the abstracted variable of the method and its type, and an applicative term. Applicative terms are numbers, variables, function terms applied to applicative term, the application of one applicative term to another, or a constructor applied to the applicative terms. When a constructor call of a class is evaluated, its arguments are first evaluated. Then the memory is allocated on the heap for the instance variables, where they are set to the evaluated arguments. The address to this memory location is the result returned by evaluating the constructor call. The only possible result of the evaluation of the applicative term is a number, so values are addresses or numbers.

The syntactic sets are groups of each entity of the syntax in the translated code. The syntactic sets described above are defined in the section 6.3. Applicative term which we write as $n, x, f[a_1, \ldots, a_n], (a \ \ b)$ and $c[a_1, \ldots, a_n]$, corresponds to the C++ constructs `n, x, f(a1,...,an), (*(a))(b)` and `new c(a1,...,a2)` The class written in the form $(\Gamma; x : A; b)$ with $\Gamma = x_1 : A_1, \ldots, x_n : A_n$, corresponds to the C++ translated code discussed in Chapter 5.

During the execution of a λ-term, a class address of the application (App) of the λ-term is created on the heap (Heap) and with respect to the environment (Env), a λ-term is evaluated to the value (Val) and extended heap (which contains the address of the value that has been evaluated for the λ-term). For a function application, the heap which contains the class address of the two terms with the values evaluated from each term is evaluated to a value and an extended heap.

The recursive description of the process in creating a system of C++ classes that represents a λ-term is based on the assumption that the λ-term is not the first term being parsed, but other terms (subterm) have been parsed before creating a system of classes, and if the term has free variables, the types of these variables are fixed an appropriate context.

After going through the definitions of the evaluation function *eval*, the implementation of the C++ classes do coincide with the denotational semantics of the simply typed λ-calculus. An integer n is evaluated by itself and a variable is evaluated by returning its value in the current environment η. The application of a native C++ functions to arguments $a_1, a_2, \ldots, a_n$ is carried out by evaluating the arguments in sequence first , and then apply the function f to those evaluated arguments. The application $(a \ \ b)$ corresponds to the construct $(*(a))(b)$ where a and b is evaluated first and due to the type correctness, a must be an element of the type of pointers to a class. Therefore the value of a will be an address on the heap. The information about the class used and the values of the instance variables are stored on the heap. $(*(a))(b)$ is then computed by evaluating the body of the method of the class in the environment where the instance variables have values as stored on the heap, and the abstracted variables has the result of evaluating b. This is what is computed by *eval* which makes use of the auxiliary function *apply*. The expression $c[\vec{a}]$, which stands for the C++ expression `new` $c(a_0, \ldots, a_n)$, is evaluated by first computing $a_0, \ldots, a_n$ in sequence. What the function *eval* carried out is the intended behaviour of C++ which is the information about the class used and the result of evaluating $a_0, \ldots, a_n$ is stored on the heap. Therefore, we can say that our implementation is proven correct.

8.2 Conclusion

Through the discussion of related works (Chapter 7) we could see that there are researchers who were very keen in merging the functional programming paradigm to C++ language with their pros and cons in doing so. The advantages of our approach/solution is that it is simple and it uses classes and inheritance in an essential way . Another advantage that is quite significant is that our approach is really integrated into C++, which avoids having strange error message like the unnecessary error messages of a user making a type error in using FC++. Furthermore in applying functional programming into C++ , one does not have to learn and use new constructs (like in FC++ and FACT!). Most importantly, we have a formal correctness proof and to our knowledge the verification of the implementation of the λ-calculus in C++ (and related object-oriented languages) using logical relations is new. A correctness proof of other implementations (such as Lambda Library and FC++) would have been difficult, since the libraries are very big, and make use of the C++ template mechanism. In our case we had complete control over the code generated, which made it much easier to carry out the proof.

The idea behind my thesis is to make established modelling and proof technology from mathematics and logic available for the analysis of stateful programs. We address the following technology; Denotational semantics for higher types which is first set theoretic then domain-theoretic (the latter is not worked out in the thesis due to lack of time), logical relations which provides powerful means to prove properties of higher type programs and Kripke semantics to deal with states.

The fact that it is possible to have a denotational semantics at a description level where pointers are manipulated explicitly entails that the well known benefits of denotational semantics, extensionality and compositionality, are still available at that level. This has been proven where we were able to give a short and concise proof of our C++ fragments using the denotational semantics instead of a complicated operational argument. More benefits are to be expected when it comes to verifying programs written in this C++ fragment.

Our original goal was to extend at reasonable fragment of C++ by λ-terms. Unfortunately this turned out to be too long, especially because using the Spirit parsing library turned out to be complicated. Spirit is difficult to use since it is a recursive descendent parser, which would have required to substantially modify the grammar of C++, whereas, using Lex or Yacc would have been much easier. Apart from this, Spirit was difficult to use because of the expansion of templates. But the advantage of using Spirit is that the grammar is directly part of the code instead of (when using Lex and Yacc) generating C++ code from the grammar. If we would start the project again, it would be advisable to start with Lex/Yacc, we realised the difficulty unfortunately too late.

We believe that if our approach is extended to cover full C++, we will obtain a language that merges the worlds of functional and object-oriented programming, and we will see many examples, where the combination of both language concepts (eg. the use of λ-terms with side effect) will result in interesting new program techniques. We have introduced a general technique for introducing lazy evaluation into C++ [ABS06b]. It is illustrated by computing the Fibonacci numbers efficiently in C++ (with the extended C++ syntax) using infinite streams and lazy evaluation.

The ideal system for our approach would be an extensions of the full language C++ with λ-terms in addition of other constructs from functional programming such as data types.

8.3 Future Work

We have shown how the functional concepts are introduced into C++ in a provable correct way. This work has lent itself to a number of extensions, such as the integration of recursive higher-order functions, polymorphic and dependent type systems, as well as the combination of larger parts of of C++ with the λ-calculus. The accurate description of these extensions would require more sophisticated, eg. domain theoretic constructions and a more systematic modelling of C++.

The proofs of theorems and how the functions P, $eval$, and $apply$ are defined is rather low level since it mentions and manipulate the class environment and the heap explicitly. It would be desirable to lift the proofs and definitions to the abstract monadic level. A framework for carrying this out might be provided by suitable versions of Moggi's Computational λ-calculus [Mog91], Pitt's evaluation logic [Pit91] and special logical relations for monads [JGLN02].

We intend to upgrade this to an extension of the language by λ-types and λ-terms together with a parser program which translates this extended language into native C++. We would like to extend our implementation to support polymorphism using C++ templates since our implementation does not support polymorphism specifically parametric polymorphism. Our implementation deduces the function type of a λ-term based on the function type of its subterms and it depends on the C++ type system for type checking. Thus, to give great value to our implementation, we would extend it to support type inference. It would be interesting to expand our fragment of C++ to deal with side effects. This would allow for instance in proving that our lazy construct shown in the section 4.6 actually gives rise to an efficient implementation of the Fibonacci function.

We would like to include memory management in our implementation to eliminate runtime crashes and memory leaks. We intend to use garbage collector in C++ i.e. using the `libgc` library. Using `libgc` automatically protects your program against memory leaks, allows writing program without calling `delete` of `free`, allows fixing premature frees in the code and provides a fast non-fragmenting memory allocator.

Appendix A

Grammar of λ-terms Coded in Spirit

```
lambstmt      = (lambtype | nativetype)
                >> no_node_d[ch_p(' ')]
                >> identifier
                >> no_node_d[ch_p(' ')]
>> no_node_d[ch_p('=')]
>> lambexp
  >> ch_p(';');

lambexp       = lambdaterm | untypedlamterm;

lambdaterm    = lambabstract | lambapp;

lambabstract  = chlit<>('\\')
        >> (lambtype | Nativetype)
>> ch_p(' ')
>> identifier
>> ch_p('.')
>> (lambabstract
   | (lambtype | Nativetype)
     >> ch_p(' ')
     >>untypedlamterm);


lambapp       = no_node_d[ch_p('(')]
                >> lambabstract
        >> no_node_d[ch_p(')')]
        >> (root_node_d[str_p("^^")]
>> (lambapp |digit | identifier);

untypedlamterm = longest_d[(digit | identifier | lambdaterm)
 >> *(root_node_d[(infixoperator | "`^^"')]
                 >> (untypedlamterm | lambdaterm)]
         | identifier >> ch_p('(')
   >> (untypedlamterm | lambdaterm)
                   >> *(ch_p(',')
                   >> (untypedlamterm | lambdaterm);

infixoperator   = ch_p('+')
          | ch_p('-')
  | ch_p('*')
  | ch_p('/');

digit           = leaf_node_d[lexeme_d[+digit_p]];

lambtype        = *(btype >> root_node_d[str_p("->")])
          >> btype;

btype           = longest_d[Nativetype
          | inner_node_d[ch_p('(') >>lambtype >> ')']
          |inner_node_d[ '(' >>btype >>')']];

nativetype      = str_p("int")
  | str_p("char")
  | str_p("string")
  | str_p("double")
  | str_p("float")
  | str_p("long")
  | str_p("short")
  | str_p("bool")
  | str_p("signed")
  | str_p("unsigned");

identifier      = leaf_node_d[nondigit
                  >>*(nondigit|digit)];

nondigit        = ch_p('_')
          | alpha_p;
```

Appendix B

Integration of Functional Programming into C++:Implementation and Verification

Integrating Functional Programming Into C++:Implementation and Verification

Rose Hafsah Ab. Rauf [*]

Abstract

We describe a parser-translator program that translates typed λ-terms to C++ classes so as to integrate functional programming. We prove the correctness of the translation with respect to a denotational semantics using Kripke-style logical relations.

1 Introduction

C++ is a general purpose language that supports object oriented programming as well as procedural and generic programming, but unfortunately not functional programming. We have developed a parser-translator program that translates typed λ-term to C++ statements so as to integrate functional progamming. This translated code uses the object oriented approach of programming that involves creation of classes for the λ-term where for a complex term the concept of inheritance is applied. We build a mathematical model from the formal semantics of the translated code to prove its correctness. First, we give the denotational semantics of the typed λ-calculus. Then the correctness of the implementation of the typed λ-calculus by C++ classes is proved with respect to the denotational semantics. The correctness proof of the translated code is based on a Kripke-style of logical relation between the C++ class and the denotational model.

The parser-translator program that has been developed will parse a string represention of typed λ-term and translate it to a sequence of C++ statements. The translation of this λ-term will be discussed in the next section. How the translated code is executed will also be discussed along with the representation of the memory allocation. The mathematical model was based on the execution of the translated code. In building up this mathematical model, we will first give the denotational semantics of the typed λ-calculus. Then we will implement the C++ classes with the denotational semantics. These will be discussed in section 3. Some related works on integrating functional programing into C++ will be discussed at the end of this paper.

The approach of using denotational semantics and logical relation in proving the correctness of programs has been used before by researchers such as Plotkin[6], and many others. The method of logical relation can be traced back at least to Tait[13] and has been used for a large variety of purposes (eg. Jung

[*] This paper is part of my Phd project and I would like to thank my supervisors Dr. Ulrich Berger and Dr. Anton Setzer for their knowledge and guidance making it possible for me to complete it.

and Tiuryn[1], Statman[9] and Plotkin[5]). To our knowledge the verification of the implementation of λ-calculus in C++ using logical relation is new.

2 Translation

For the purpose of explaining how the λ-term is translated to its equivalent C++ statements and execution of the translated code, we will not go through the details of the parser-translator program in action. A λ-term $\lambda\, x^{\alpha}\, .\, t$ is witten in our syntax as $\backslash\lambda.x.\beta.t$ where $t : \beta$. We will give an example of a λ-term input to the parser-translator program and how it is executed. The statement shown below is the string that was entered to the program:

$$int\ \ k = ((\backslash int\ \ \rightarrow int\ \ f \cdot \backslash int\ \ x \cdot int\ \ f\wedge\wedge f\wedge\wedge x)\wedge\wedge(\backslash int\ \ x \cdot int\ \ x + 2))\wedge\wedge 3 \quad (1)$$

and it is equivalent to $k = (\lambda f \cdot \lambda x \cdot f(fx))(\lambda x \cdot x + 2)3$

The λ-term $\backslash int\ \ \rightarrow int\ \ f \cdot \backslash int\ \ x \cdot int\ \ f\wedge\wedge f\wedge\wedge x$ in the statement above is translated as objects which is defined as follows ;

```
class lambda1 : public Cint_intD_aux{
 public :Cint_intD f;
 lambda1( Cint_intD f)  {   this-> f = f;};
 virtual int operator () (int x)
 { return (*(f))((*(f))(x)); };
};
class lambda0 : public CCint_intD_Cint_intDD_aux{
 public :
 lambda0( ) { };
 virtual Cint_intD operator () (Cint_intD f)
 { return new  lambda1( f); }
};
```

and the λ-term $\backslash int\ \ x \cdot int\ \ 2 + x$ is translated as follows :

```
 class lambda2 : public Cint_intD_aux{
 public :
 lambda2( ) { };
 virtual int operator () (int x)
 { return x + 2; };
};
```

The statement (1) will be finally translated as the expression :

$$int\ \ k = (*((*(new\ \ lambda0()))(new\ \ lambda2())))(3);$$

The classes for the λ-terms are instantiated by statements new lambda0() and new lambda2() where pointers will be created that point to the addresses of the classes on the heap. The heap which is also known as free store is a dynamic store in the memory. Classes are created for each λ-term objects and each classes have pointers to its addresses on the heap. The local variables and function parameters are stacked for every execution and these storage allocated for the variables will be deleted after each execution terminates.

3 Proof of correctness

Before we start building a mathematical model of the translated code, we list some of the mathematical preliminaries that will be frequently used in this section. The presentation of the proof follows the style of Winskel[14].

3.1 Mathematical preliminaries

Mappings

1. If X, Y are sets, then a list $m = (x_1 : y_1), \ldots, (x_n : y_n) \in$ list(X × Y) is considered as a finite map from X to Y which is defined as follows : If $x \in$ X, $y \in$ Y, then $m(x) := y$ where $x = x_i, y = y_i$ and $x \neq x_j$ for $j > i$.

2. We usually define dom(m) = the domain of $m = x_1, \ldots, x_n$.

   If $x \in$ X ,$y \in$ Y, then $m[x \mapsto y]$:= m, (x,y), the extension of the list m by (x,y). Note that dom($m[x \mapsto y]$) = *dom*(m) $\cup\{x\}$ and

$$m[x \mapsto y](x') = \begin{cases} y & \text{if } x' = x \\ m(x') & \text{if } x' \in \text{dom}(m) \setminus \{x\} \end{cases} \quad (x' \in \text{X})$$

3.2 Implementation of the typed λ-calculus

a) **Types**

The set **Typ** of types is inductively given by :

i) Int ∈ Typ

ii) if $A, B \in$ Typ, then $A \to B \in$ Typ

b)**Terms**

The **Terms** for the λ-calculus can be any of the following shown below.

i) $n \in$ N (any number)

ii) $x \in$ Var (where Var = String)

iii) $r\,s$ (term r is applied to term s)

iv) $\lambda x : A.r$ (term is an abstraction)

v) $f[r_1 \ldots r_n] = f[\vec{r}]$ ($f \in \mathcal{F}$ is a set of names for computable functions on N).The function denoted by f is written as $[\![f]\!]$

c) **Typing**

A **Context Γ** is a map from variables to types i.e. a list of variables and their type : **Context=list(Var× Typ)**

Context will be denoted as $\Gamma = x_1 : A_1, \ldots, x_n : A_n$

The **Typing** rules of the simply typed λ-calculus are :

i)

$$\frac{}{\Gamma, x : A \vdash x : A}$$

ii)

$$\frac{}{\Gamma \vdash n : \text{Int}}$$

iii)

$$\frac{\Gamma, x : A \vdash r : B}{\Gamma \vdash \lambda x r : A \to B}$$

iv)

$$\frac{\Gamma \vdash r : A \to B \qquad \Gamma \vdash s : A}{\Gamma \vdash rs : B}$$

v)

$$\frac{f : \mathrm{Int} \times \ldots \times \mathrm{Int} \to \mathrm{Int} \qquad \Gamma \vdash r_1 : \mathrm{Int} \ldots \Gamma \vdash r_n : \mathrm{Int}}{\Gamma \vdash f[r_1, \ldots, r_n] : \mathrm{Int}}$$

d) **Denotational semantics**

The sets of **functionals** of type A denoted as $\mathrm{D}(A)$ are defined as follows :

i) $\mathrm{D(Int)} = \mathrm{N}$

ii) $\mathrm{D}(A \to B) = \{f | f : \mathrm{D}(A) \to \mathrm{D}(B)\}$

iii) $\mathrm{D} := \biguplus_{A \in \mathrm{Typ}} \mathrm{D}(A)$ where $\biguplus$ denotes disjoint union

A **Functional Environment** is a mapping of ξ : Var → D. We let FEnv := Var → D be the set of all functional environments. If Γ is a context, then $\xi : \Gamma$ means $\forall x \in \mathrm{dom}(\Gamma).\xi(x) \in \mathrm{D}(\Gamma(x))$.

For every typed λ-term $\Gamma \vdash r : A$ and every functional environment $\xi : \Gamma$ the denotational value $\llbracket r \rrbracket \xi \in \mathrm{D}(A)$ is defined as follows :

i) $\llbracket n \rrbracket \xi = n$

ii) $\llbracket x \rrbracket \xi = \xi(x)$

iii) $\llbracket r \quad s \rrbracket \xi = \llbracket r \rrbracket \xi(\llbracket s \rrbracket \xi)$

iv) $\llbracket \lambda x : A.r \rrbracket \xi(a) = \llbracket r \rrbracket \xi[x \mapsto a]$

v) $\llbracket f[\vec{r}] \rrbracket = \llbracket f \rrbracket(\llbracket \vec{r} \rrbracket \xi)$

By an **implementation** of the typed λ-calculus we mean an (implementation of an) algorithm computing for every closed term r : Int the value $\llbracket r \rrbracket \in$ N.

3.3 Implementation by C++ classes

The classes that will be created depend on the λ-term that is being parsed, the more complex the term is the more level of classes will be created and this involves inheritance. When the class is instantiated, an address of the class will be stored on the heap, and further instantiation of other classes will create a stack of addresses on the heap with addresses of any variables which is bound to the classes (or λ-term).

Every class is instantiated by calling the constructor of the object i.e. the name of the class with or without any arguments.The body of the λ-term is associated with the application in the syntactic sets of this translated code. The list of syntactic sets associated with the C++ classes are as follows:

- **Addr = Int**
  These are addresses (**Addr**) of classes or variables on the Heap.

- **Constr = String**
  Constructor (**Constr**) is the name of the class

- **Val = Int + Addr**
  A value (**Val**) is either an integer or an address of a class or variables

- **App = Int + Var +** $\mathcal{F}\times$ **list(App) + App** $\times$ **App + Constr** $\times$ **list(App)**

- **Abst = Var** $\times$ **Typ** $\times$ **Context** $\times$ **App**
  Abstraction(**Abst**) consist of the variables and the type bound to the abstraction, and the context which is the list of variables and their types and the application. Types like Int→Int will be represented in C++ as strings.

- **Env = list(Var** $\times$ **Val)**
  Environment (**Env**) is the list of variables and their values

- **Heap = list(Addr** $\times$ **Constr** $\times$ **list(Val))**
  Heap consists of list of addreses of constructors and their list of values of the variables

- **Class = list(Constr** $\times$ **Abst)**
  Class consists of list of constructor and their abstraction

We assume that every $f \in \mathcal{F}$ is given by a side effect free C++ function

a) **The evaluation of the λ-terms in C++**

When a λ-term is executed, a class address of the application of the λ-term is created on the heap and with respect to the environment, a λ-term is evaluated to the value and an extended heap. This extended heap contains the address of the value that has been evaluated for the λ-terms. Thus the functionality of the evaluation function (**eval**) is :

$$\textbf{eval} : \textbf{Heap} \rightarrow \textbf{Env} \rightarrow \textbf{App} \rightarrow \textbf{Val} \times \textbf{Heap}$$

For a function application, where a lambda term is applied to another lambda term, the heap which contain the classes address of the two terms with the two values evaluated from the two terms will evaluate to a value and an extended heap. Thus the functionality of the application function (**apply**) is :

$$\textbf{apply} : \textbf{Heap} \rightarrow \textbf{Val} \rightarrow \textbf{Val} \rightarrow \textbf{Val} \times \textbf{Heap}$$

In the definition of the function **eval** and **apply** we fix some C:Class.
In presenting the evaluation rules we will follow the convention that :

- n ranges over numbers **N**

- x ranges over variables **Var**

- a , b ranges over application **App**

- v , w ranges over values **Val**

- k ranges over address **Addr**
- H ranges over **Heap**
- c ranges over constructor **Constr**
- C ranges over **Class**
- A, B ranges over **Typ**
- η ranges over **Env**

The metavariables we use to range over the syntactic categories can be primed or subscripted. For example, H, H', H'', H_k stand for heaps, C, C', C'' stand for classes and v_1, v' stand for values.

The rules for the evaluation of the λ-terms are as follows:

i) **Evaluation of a λ-term where application is a number**

$$\mathrm{eval}\, H\, \eta\, n = (n, H)$$

ii) **Evaluation of a λ-term where application is a variable**

$$\mathrm{eval}\, H\, \eta\, x = (\eta(x), H)$$

iii) **Evaluation of a λ-term where application is a function with a list of arguments**

$$\mathrm{eval}\, H\, \eta\, f[\vec{a}] = ([\![f]\!](\vec{n}), H_k)$$

where $\vec{a} = a_1, \ldots, a_k, \vec{n} = n_1, \ldots, n_k$ and $\mathrm{eval}^*\, H\, \eta\, \vec{a} = (\vec{n}, H_k)$.

Here we define $\mathrm{eval}^*\, H\, \eta\, \vec{a} = (\vec{n}, H_k)$ if eval $H \eta\, a_1 = (n_1, H_1), \ldots,$ eval $H_{k-1}\, \eta\, a_k = (n_k, H_k)$. H_k is not changed by f because $f \in \mathcal{F}$ has no side effect.

iv) **Evaluation of a λ-term where the application is the application of one term to the other**

$$\mathrm{eval}\, H\, \eta\, (a \;\; b) = \mathrm{apply}\, H'' v\, w = (v', H''')$$

where $\mathrm{eval}\, H\, \eta\, a = (v, H'),\;\; \mathrm{eval}\, H'\, \eta\, b = (w, H'')$

The definition of **apply** in detail is shown as follows :

$$\mathrm{apply}\ \mathrm{H}\, k\, v = \mathrm{eval}\, H\, [x, \vec{y} \mapsto v, \vec{w}] a$$

where $H(k) = (c, \vec{w})$, $C(c) = (x : A; \vec{y} : \vec{B}; a)($ assuming $c \in \mathrm{dom}(C))$

v) **Evaluation of a λ-term where the application is a constructor with a list of arguments**

$$\mathrm{eval}\, H\, \eta\, c[\vec{a}] = (k, H'[k \mapsto c[\vec{v}]]) \quad (k \in \mathbf{Addr}, v \in \mathbf{Val})$$

where $\mathrm{eval}^*\;\; H\, \eta\, \vec{a} = (\vec{v}, H')$ and $k = \mathrm{new}\ H'$ (new H' is an address not in $\mathrm{dom}(H')$)

In all other cases for the application, it is termed invalid and an error will be returned.

Lemma 1

1. eval $H\,\eta\,a = (v, H') \Longrightarrow H \subseteq H'$
2. apply $H\,v\,w = (v', H') \Longrightarrow H \subseteq H'$
3. eval* $H\,\eta\,\vec{a} = (\vec{n}, H') \Longrightarrow H \subseteq H'$

The proof for Lemma 1 is by induction on the definition of **eval** and **apply**.

Note that, since **eval** and **apply** depend on C:Class, the true signatures of eval and apply are as follows :

eval : Class→Heap→Env→App→Val×Heap
apply : Class→Heap→Val→Val→Val×Heap

We write $\mathbf{eval}_C\, H\,\eta\,a$ and $\mathbf{apply}_C\, H\,v\,w$ if the argument C:Class is to be made explicit.

b) **The Parsing of the λ Term**

Traditionally, the λ-term that is input is parsed as a long string which will undergo several steps of parsing to get the translated code. The parsing will create classes for the λ-term where in the case of a complex λ-term it will create several levels of classes where the class of an upper level is an extension of the lower level class. In order to simplify things and to concentrate on the most important aspects of the problem we assume that the input is given as an abstract term rather than a string. The parsing from a string to a term is a traditional parsing problem which is of no interest here. What is interesting here is the process of creating a system of C++ classes that represents a λ-term.

In order to give a recursive description of this process, we must assume that the term in question is not the first term being parsed, but other terms (or subterms) have been parsed before having created a system of classes. Furthermore, if the term has free variables, then the types of these variables must be fixed by an appropriate context. Therefore, the parser **P** has the following functionality :

$$\mathbf{P : Class \to Context \to Term \to App \times Class}$$

The rules for the parsing are as follows :

i) **Parsing when the term is a number:** $\mathrm{P}_C\Gamma n = (n, C)$

ii) **Parsing when the term is a variable:** $\mathrm{P}_C\Gamma x = (x, C)$

iii) **Parsing when the term is a function with a list of arguments :**

$$\mathrm{P}_C\Gamma f[\vec{r}] = (f[\vec{a}], C')$$

where $\mathrm{P*}_C\Gamma\vec{r} = (\vec{a}, C')$ and P* is defined in a smilar way as eval*.

iv) **Parsing of an application:** $\mathrm{P}_C\Gamma(r\ \ s) = (a\ \ b, C'')$
where $\mathrm{P}_C\Gamma r = (a, C')$, $\mathrm{P}_{C'}\Gamma s = (b, C'')$

v) **Parsing of a λ abstraction :** $\mathrm{P}_C\Gamma(\lambda x : A.r) = (c[\vec{y}], C'[c \mapsto (x : A; \Gamma; a)])$
where $\vec{y} = \mathrm{dom}(\Gamma), \mathrm{P}_C\Gamma[x \mapsto A]r = (a, C'),$ and $c = \text{new } C'$
meaning that c is a name of a class that is "new" i.e. has not been used before.

Remark: We only generate $c[\vec{x}] \in$ App with $\vec{x} \in$ list(Var) and not $c[\vec{a}]$ with arbitary $\vec{a} \in$ list(App)
Lemma 2 i) $\mathrm{P}_C\Gamma r = (a, C') \Longrightarrow C \subseteq C'$
ii) $\mathrm{P}^*{}_C\Gamma\vec{r} = (\vec{a}, C') \Longrightarrow C \subseteq C'$
The proof for Lemma 2 is by induction on r respectively $\vec{r}$.

3.4 The correctness of the translated code

The correctness proof of the translated code is based on a Kripke-style relation between the C++ representation of the term ($\in$ Val $\times$ Heap) and its denotational value ($\in \mathrm{D}(A)$). The relation is indexed by the class environment C and the type A of the term. Since in the case of an arrow type, $A \to B$, extensions of H and C have to be taken into account, this definition has some similarity with Kripke models. The relation

$$\sim_A^C \subseteq (\text{Val} \times \text{Heap}) \times \mathrm{D}(A) \text{where } A \in \text{ Typ }, C \in \text{Class}$$

is defined by recursion on A as follows:

$$\begin{aligned}
(v, H) \sim_{\text{Int}}^C n : &\iff v = n \\
(v, H) \sim_{A \to B}^C f : &\iff \forall C \subseteq C', \forall H \subseteq H', \forall (w, d) \in \text{Val} \times \mathrm{D}(A) : \\
&\qquad (w, H') \sim_A^{C'} d \Longrightarrow \text{apply}_{C'} H' v w \sim_B^{C'} f(d)
\end{aligned}$$

We also set $(\eta, H) \sim_\Gamma^C \xi := \forall x \in \text{ dom } \Gamma(\eta(x), H) \sim_{\Gamma(x)}^C \xi(x) \in \mathrm{D}(\Gamma(x))$
Lemma 3:

$$(v, H) \sim_A^C d, C \subseteq C', H \subseteq H' \Longrightarrow (v, H') \sim_A^{C'} d$$

The proof for Lemma 3 is by induction on A.

Our main theorem, which corresponds to the usual "Fundamental Lemma" or "Adequacy Theorem" for logical relations, reads as follows:
Theorem: If η : Env, ξ : FEnv, $\Gamma \vdash r : A, \xi : \Gamma, \mathrm{P}_C\Gamma r = (a, C'), C' \subseteq C'', (\eta, H) \sim_\Gamma^{C''} \xi,$ and $H \subseteq H',$ then $\text{eval}_{C''} H' \eta a \sim_A^{C''} [\![r]\!]\xi$

The theorem can be proved by an induction on the typing judgement $\Gamma \vdash r : A$ using the Lemma 1-3 above. Due to limited space we omit details.

For a closed term r , we define $\mathrm{P}r = \mathrm{P}_\emptyset\emptyset r$.

Corollary(Correctness of the implementation):
If $\vdash r :$ Int, $\mathrm{P}r = (a, C), C \subseteq C'$, then for any heap H, $\text{eval}_{C'} H \eta a = ([\![r]\!], H')$ for some $H' \supseteq H$

4 Conclusion

The aim of this paper was to introduce a new approach of integrating functional programming into C++ and to show a method of proving the correctness of the translation code produced by denotational semantics and logical relation. In the past, several researches [2],[3] discovered that C++ can be used for functional programming by representing first class functions and higher order functions using classes, and by this technique we produced the translated code. There are other approaches that have made C++ a language that can be used for functional programming such as FC++ library [4] (a very elaborate approach), FACT! [12] (extensive use of templates and overloading) and [2] (creating macros that allow creation of single macro-closure in C++).The advantages of our solution are that it is very simple, it uses classes and inheritance in an essential way and, most importantly, we have a formal correctness proof.

In addition to the mathematical proof given in this paper, the correctness of the translated code produced by the parser-translator program has been verified by testing it with several types of λ-term from simple to complex ones.

References

[1] Jung A., Tiuryn J.: A New Characterization of Lambda Definability. Typed Lambda Calculus and Applications, 1993.

[2] Kiselyov O.: Functional Style in C++ : Closures, Late Binding, and Lambda Abstraction. Poster presentation, Int. Conf. on Functional Programming, 1998.

[3] Läufer K.: A Framework For Higher Order functions in C++. Proc. Conf. Object Oriented Technologies(COOTS), Monterey, C.A., June 1995.

[4] McNamara B., Smaragdakis Y.: Functional Programming in C++. ICFP '00, Montreal Canada,ACM Press, 2000.

[5] Plotkin G. D.: Lambda Definability in the Full Type Hierarchy. To H.B. Curry; Essays on Combinatoric Logic, Lambda Calculus and Formalism., J.P.Seldin , J.R. Hindley, eds., 363-373, 1980.

[6] Plotkin G. D.: LCF Considered As a Programming Language. Theoretical Computer Science, 5:223-255, 1977.

[7] Polak W.: Program Verification Based On Denotational Semantics. Proceedings of the 8^{th} ACM, SIGPLAN-SIGACT Symposium on Principles Of Programing Analysis. ACM Press, Jan. 1981.

[8] Setzer A.: Java as a Functional Programming Language. Types for Proofs and Programs: International Workshop, Types 2002, Berg en Dal,April 24-28,2002. Selected Papers, Geuver H., Wiedijk F., eds, 279-298, LNCS 2646, 2003.

[9] Statman R.: Logical Relation and the Typed λ Calculus. Information and Control, 65:85-97, 1985.

[10] Stoy, J: Denotational Semantics - The Scot-Strachey Approach To Language Theory. MIT Press, Cambride, 1977.

[11] Scott, D., Strachey, C.: Mathematical Semantics For Computer Language. Tech. Monograph PRG-6, Programming Research Group, University Of Oxford, 1971.

[12] Striegnitz J. : FACT!-The Functional Side of C++. http://www.fz-juelich.de/zam/FACT.

[13] Tait W.: Intentional Intrepretation of Funtional of Finite Type I. Journal Of Symbolic Logic, 32(2):198-212, 1967.

[14] Winskel, G. : The Formal Semantics Of Programming Languages : an Introduction. Massachusetts Institute Of Technology, 1993.

Appendix C

A Provably Correct Translation of the Lambda-Calculus into a Mathematical Model of C++

A Provably Correct Translation of the λ-Calculus into a Mathematical Model of C++

Rose H. Abdul Rauf,[*] Ulrich Berger, Anton Setzer[‡]

Abstract

We introduce a translation of the simply typed λ-calculus into C++, and give a mathematical proof of the correctness of this translation. For this purpose we develop a suitable fragment of C++ together with a denotational semantics. We introduce a formal translation of the λ-calculus into this fragment, and show that this translation is correct with respect to the denotational semantics. We show as well a completeness result, namely that by translating λ-terms we obtain essentially all C++ terms in this fragment. We introduce a mathematical model for the evaluation of programs of this fragment, and show that the evaluation computes the correct result with respect to this semantics.

1 Introduction

C++ is a general purpose language that supports object-oriented programming as well as procedural and generic programming, but unfortunately not directly functional programming. We have developed a parser-translator program that translates typed λ-terms into C++ statements so as to integrate functional concepts. The translated code uses the object-oriented approach of programming that involves the creation of classes for the λ-term. By using inheritance, we achieve that the translation of a λ-abstraction is an element of a function type. As example we show how the λ-term

$$t = (\lambda f^{\mathsf{int}\to\mathsf{int}}\lambda x^{\mathsf{int}}.f\ (f\ x))\ (\lambda x^{\mathsf{int}}.x + 2)\ 3$$

is translated into native C++ code. We begin with the translation of the types involved. Those are int, int → int and (int → int) → (int → int), the latter being the type of the subterm $\lambda f^{\mathsf{int}\to\mathsf{int}}\lambda x^{\mathsf{int}}.f\ (f\ x)$. The type int is translated into the native C++ type `int`. The type int → int translates to

```
class Cint_intD_aux
{ public : virtual int operator() (int x) = 0; };

typedef Cint_intD_aux*  Cint_intD;
```

[*] Faculty of Information System and Quantitative Science, University of Technology MARA, 40450 Shah Alam, Selangor D.E., Malaysia. Email: hafsah@tmsk.uitm.edu.my

[†] Department of Computer Science, University of Wales Swansea, Singleton Park, Swansea SA2 8PP, UK. Email: {u.berger,a.g.setzer}@swansea.ac.uk

[‡] Supported by EPSRC grant GR/S30450/01.

i.e. the type of pointers to elements of the class Cint_intD_aux containing one virtual method. The translation of $(\mathsf{int} \to \mathsf{int}) \to (\mathsf{int} \to \mathsf{int})$ is similar:

```
class CCint_intD_Cint_intDD_aux
{  public : virtual Cint_intD operator()
                              (Cint_intD x) = 0; };

typedef CCint_intD_Cint_intDD_aux*
        CCint_intD_Cint_intDD;
```

As for types, the translation of the term t proceeds in stages. The class definition for the subterm $t_1 := \lambda x^{\mathsf{int}}.f\ (f\ x)$ is

```
class t1 : public Cint_intD_aux{
 public :Cint_intD f;
 t1( Cint_intD f)  {   this-> f = f;};
 virtual int operator () (int x)
 { return (*(f))((*(f))(x)); };
};
```

and for $t_0 := \lambda f^{\mathsf{int} \to \mathsf{int}} \lambda x^{\mathsf{int}}.f(fx)$, $t_2 := \lambda x^{\mathsf{int}}.2 + x$ (using identifiers t0, t2)

```
class t0 : public CCint_intD_Cint_intDD_aux{
 public :
 t0( ) { };
 virtual Cint_intD operator () (Cint_intD f)
 { return new  t1( f); }
};

 class t2 : public Cint_intD_aux{
 public :
 t2( ) { };
 virtual int operator () (int x)
 { return x + 2; };
};
```

Finally, the whole term t translates into the C++ expression

```
(*((*( new t0( )))( new t2( ))))(3);
```

In this article, we do not only present this translation, but give as well a mathematical proof that it is correct. For this purpose we introduce a suitable fragment of C++ with a precise denotational semantics. We give a formal translation of λ-terms into this fragment and show that it preserves this semantics. We will show as well completeness, i.e. essentially all programs in this fragment of C++ can be obtained by translating terms of the λ-calculus. We develop a mathematical model for the evaluation of programs in this model, and show that this evaluation is correct with respect to the denotational semantics. This shows that our translation results in C++ programs which are evaluated correctly in our mathematical model of C++.

We hope that our model of a fragment of C++ which includes a formal model of the heap, will have applications which go beyond the translation of the typed λ-calculus. We expect that extensions of this model can be used to verify formally the correctness of more complex C++ programs, including programs with side effects.

Organisation of the paper. In Sect. 2 we introduce the typed λ-calculus together with a standard denotational semantics. In Sect. 3, we present our parser-translator program, which translates λ-terms into the full language of C++. We discuss how the translated code is executed including a description of the memory allocation. Since it is not feasible (at least for our group) to prove the correctness of our translation with respect to the full operational semantics of the very complex language C++, we develop in Sect. 4 a small fragment of C++ into which we can translate the typed λ-calculus. We introduce as well the evaluation of applicative terms in this language. In Sect. 5 we give a formal translation of the typed λ-calculus into this fragment. We introduce a denotational semantics of the fragment of C++ and show that formal translation respects the denotational semantics (correctness). Furthermore we show that, if one imposes a slight restriction on the applicative terms, then up to renaming of class names all applicative terms can be obtained by translating suitable typed λ-terms (completeness). In Sect. 6 we show that the evaluation of λ-terms is correct with respect to the denotational semantics. This proof makes use of a Kripke-style logical relation. We conclude with an overall result, namely that if we translate a λ-term into the fragment of C++ and evaluate it, we obtain the correct result with respect to the denotational semantics.

Related work. Several researchers [Kis98], [Läu95] have discovered that C++ can be used for functional programming by representing higher order functions using classes. Our representation is based on similar ideas. There are other approaches that have made C++ a language that can be used for functional programming such as the FC++ library [MS00] (a very elaborate approach) as well as FACT! [Str] (extensive use of templates and overloading) and [Kis98] (creating macros that allow creation of single macro-closure in C++). What is different in our paper is that we develop a mathematical model of a fragment of C++, and that we formally prove the correctness of our translation.

The approach of using denotational semantics and logical relations for proving program correctness has been used before by Plotkin [Plo77], Reynolds [Rey83] and many others. The method of logical relations can be traced back at least to Tait [Tai67] and has been used for various purposes (e.g. Jung and Tiuryn [JT93], Statman [Sta85] and Plotkin [Plo80]). To our knowledge the verification of the implementation of the λ-calculus in C++ (and related object-oriented languages) using logical relations is new.

There are other fragments of object-oriented languages in the literature which are used to prove the correctness of programs. A well-known example is Featherweight Java ([IPW99]). The model for this language avoids the use of a heap, since methods do not modify instance variables. In contrast, our model of C++ does make use of a heap and is therefore closer to the actual implementation of C++. Although our fragment of C++ does not allow for methods with side effects, it could easily be extended this way and then used to verify programs in C++ with side effects.

Acknowledgements: We would like to thank the referees of earlier versions of this paper for valuable comments.

1.1 General Notations

Notation 1.1 (Finite maps) *By $X \rightarrow_{\mathsf{fin}} Y$ we denote the set of finite maps from the set X to the set Y, that is, the set of functions $f\colon \mathsf{dom}(f) \rightarrow Y$ where $\mathsf{dom}(f)$ is a finite subset of X. If $f \in X \rightarrow_{\mathsf{fin}} Y$ and $(x,y) \in X \times Y$, then $f[x \mapsto y]$ denotes the finite function with domain $\mathsf{dom}(f) \cup x$ that sends x to y and any other $x' \in \mathsf{dom} f$ to $f(x')$. A list $x_1 : y_1, \ldots, x_n : y_n$, where the x_i are distinct elements of X and the y_i are in Y, denotes an element of $X \rightarrow_{\mathsf{fin}} Y$ in the obvious way. Furthermore, $f, x : y := f[x \mapsto y]$.*

In this article we observe a strict naming convention: Once a group of letters has been used to range over a certain entity (e.g. A, B (but not C) range over types), letters in that group (possibly with sub- or superscripts) will always denote instances of that entity. There will be only two exceptions: x, y, z may denote elements of unspecified sets X, Y, Z as well as variables (Definition 2.2), and f ranges over unspecified functions as well as basic C++ functions (Assumption 2.1).

List of notations. For the reader's convenience there is a complete table of notations in Sect. 8 at the end of this article

2 The Typed λ-Calculus

We introduce a version of the typed λ-calculus based on base types, which are native C++-types, and basic functions, which are native C++-functions.

Assumption 2.1 *(a) We fix a set* basetype *of base types $\rho, \sigma, \ldots$. One specific base type is the type of integers* int.

(b) We fix a set F *of names for* basic functions *$f : (\rho_1, \ldots, \rho_k) \rightarrow \sigma$.*

(c) We view functions of arity 0 as constants and denote them by the letter n.

(d) Let $[\![\rho]\!]$ denote the set of elements of base type ρ. In case of int, *$[\![\mathsf{int}]\!]$ is the set of integers.*

(e) Let $[\![f]\!] : [\![\rho_1]\!] \times \ldots \times [\![\rho_k]\!] \rightarrow [\![\sigma]\!]$ be the function denoted by $f \in \mathsf{F}$. Especially we assume that a constant (0-ary function) n, which stands for the integer n, is interpreted by itself (i.e. by n).

Any native C++ type can be used as a base type, and any native C++ functions without side effects (including constants) can be used as basic functions [1].

Definition 2.2 (Simply typed λ-calculus with basic functions)

(a) We fix a set Var *of variables $x, y, z, \ldots$.*

(b) The sets of types, contexts and λ-terms are defined as follows:

[1] The translation given below makes sense as well for functions with side effects, including those which affect instance variables of the classes used. However, in this case we would go beyond the simply typed λ-calculus, and could not use the simple denotational semantics of the λ-calculus in order to express the correctness of the translation.

$$\begin{array}{llll} \textit{Types:} & \mathsf{Type} \ni A, B, E, F & ::= & \rho \mid A \to B \\ \textit{Contexts:} & \mathsf{Context} \ni \Gamma, \Delta & ::= & \mathsf{Var} \to_{\mathsf{fin}} \mathsf{Type} \\ \lambda\textit{-terms:} & \mathsf{Term} \ni r, s, t & := & x \mid f[r_1, \ldots, r_k] \mid r\ s \mid \lambda x^A.r \end{array}$$

(c) *The relation* $\Gamma \vdash r : A$ *is inductively defined by the following standard* typing rules*:*

$$\Gamma, x : A \vdash x : A \qquad \frac{\Gamma, x : A \vdash r : B}{\Gamma \vdash \lambda x^A.r : A \to B} \qquad \frac{\Gamma \vdash r : A \to B \qquad \Gamma \vdash s : A}{\Gamma \vdash r\ s : B}$$

$$\frac{\Gamma \vdash r_1 : \sigma_1 \ldots \Gamma \vdash r_k : \sigma_k}{\Gamma \vdash f[r_1, \ldots, r_k] : \rho} \quad (f : (\rho_1, \ldots, \rho_k) \to \sigma)$$

Note that we do not have product types and that native C++-functions are not necessarily stand-alone C++ terms and hence not λ-terms either. Therefore the rule for $f[r_1, \ldots, r_k]$ is not subsumed by the rule for $r\ s$.

The simply type λ-calculus has a well-known operational semantics defined by β-reduction, $(\lambda x^A r)s \to_\beta r[s/x]$, and function reduction, $f[n_1, \ldots, n_k] \to_f [\![f]\!](n_1, \ldots, n_k)$. But there is also an equivalent denotational semantics which, for our purposes, will be more convenient to work with. Since our calculus does not allow for recursive definitions, it is possible to interpret types and terms in a naive set-theoretic hierarchy D of functionals of finite types over the base types:

Definition 2.3 *For* $A \in \mathsf{Type}$ *we define the set* $\mathsf{D}(A)$ *of functionals of finite types over* A *by induction on* A*:*

$$\begin{array}{rcl} \mathsf{D}(\rho) & := & [\![\rho]\!] \\ \mathsf{D}(A \to B) & := & \textit{the set of functions from } \mathsf{D}(A) \textit{ to } \mathsf{D}(B) \\ \mathsf{D} & := & \bigcup\limits_{A \in \mathsf{Type}} \mathsf{D}(A) \end{array}$$

Semantic values (elements of $\mathsf{D}(A)$*) are denoted by* d*.*

Definition 2.4 *(a) A* functional environment *is a mapping* $\xi : \mathsf{Var} \to \mathsf{D}$.
FEnv *denotes the set of all functional environments.*

(b) If Γ *is a context, then* $\xi : \Gamma$ *means* $\forall x \in \mathsf{dom}(\Gamma).\xi(x) \in \mathsf{D}(\Gamma(x))$.

Definition 2.5 (Denotational semantics of the simply typed λ-calculus) *For every typed* λ*-term* $\Gamma \vdash r : A$ *and every functional environment* $\xi : \Gamma$ *the denotational value* $[\![r]\!]\xi \in \mathsf{D}(A)$ *is defined by*

$$\begin{array}{rcl} [\![x]\!]\xi & = & \xi(x) \\ [\![f[r_1, \ldots, r_k]]\!]\xi & = & [\![f]\!]([\![r_1]\!]\xi, \ldots, [\![r_k]\!]\xi) \\ [\![r\ s]\!]\xi & = & [\![r]\!]\xi([\![s]\!]\xi) \\ [\![\lambda x^A.r]\!]\xi & = & \boldsymbol{\lambda} d \in \mathsf{D}(A).[\![r]\!]\xi[x \mapsto d] \end{array}$$

3 Translation of Typed λ-Terms into C++

In this section we describe how to translate simply typed λ-terms into C++ using the object-oriented concepts of classes and inheritance.

The translation generates new identifiers, which we need to disambiguate; in order for this to work, we restrict ourselves to the translation of finitely many λ-terms and types at a time. We first define an identifier $\mathsf{name}(A) : \mathsf{String}$ for finitely many $A : \mathsf{Type}$. Here String is the set of strings.

- If ρ is a native C++-type, $\mathsf{name}(\rho)$ is a C++ identifier obtained from ρ. This is ρ, if ρ is already an identifier, and the result of removing blanks and modifying symbols not allowed in identifiers (e.g. replacing $*$ by x), in case ρ is a compound type like `long int` or $* \ \rho$. [2]

- $\mathsf{name}(A \to B) :=$"C"$*\mathsf{name}(A)*$"_"$*\mathsf{name}(B)*$"D", where $*$ means concatenation. Here C stands for an open bracket, D for a closing bracket, and _ for the arrow in this identifier. By using these symbols we obtain valid C++-identifiers.[3]

For instance $\mathsf{name}(\mathtt{int} \to \mathtt{int}) =$"Cint_intD", $\mathsf{name}((\mathtt{int} \to \mathtt{int}) \to \mathtt{int}) =$ "CCint_intD_intD". In the following, we write CA_BD instead of $\mathsf{name}(A \to B)$ and CA_BD_aux instead of $\mathsf{name}(A \to B)*$"_aux" (that type will be introduced below), similarly for other types.

For every $A \in \mathsf{Type}$ we introduce a series of class definitions, after which $\mathsf{name}(A)$ is a valid C++ type (assuming class definitions for any native C++ type used):

- For native C++-types the sequence of class definitions is empty.

- The sequence of class definitions for $A \to B$ consists of the class definitions of A, the class definitions of B not contained in the class definitions of A and additionally

  ```
  class CA_BD_aux{
    public: virtual B operator () (A x)=0;};
  typedef CA_BD_aux * CA_BD;
  ```

  So, CA_BD_aux is a class with one virtual method used as application,[4] which maps an element of type A to an element of type B. CA_BD is the type of pointers to elements of this class. The body of this method will then be the body of the function to be invoked when applied to its arguments.

[2] This modification might result in name clashes, in which case one adds some string like _n for some integer n in order to disambiguate the names. Since we are translating only finitely many λ-types at any time, this way of avoiding name clashes is always possible.

[3] Again, we might need to disambiguate the identifiers as it was done for native C++ types.

[4] In C++, if an object `o` has a method with signature `B operator () (A x)`, invocation of this method is written like application, i.e. as `o(s)`. Note however that Java objects correspond in C++ to pointers to C++-objects. A pointer `o'` to an object `o` has first to be dereferenced, written as `(* o')`, and therefore `o'` applied to `s` is written as `(* o')(s)`. Note that, when creating an object using `new`, we obtain a pointer to an object.

Now we define for every λ-term r a sequence of C++-class definitions and a C++-term r^{C++}, such that if $r : A$, then r^{C++} is of type name(A). [5]

- If x is a variable, then the class definitions for introducing x are empty and $x^{C++} := x$.

- Let $t = \lambda x^A.r$ be of type $A \to B$. Assume the free variables of t are of type $x_1 : A_1, \ldots, x_k : A_k$ and that t is a new identifier. Assume name(A_i) = Ai, xi is the C++-representation for x_i, name(A) = A, name(B) = B, and r^{C++} = r. The class definition for t consists of the class definition for r together with

```
class t : CA_BD_aux{
  public:
    A1 x1;
    ...
    An xk;
    t(A1 x1,A2 x2, ... , Ak xk){
         this->x1 = x1;
         ...
         this->xk = xk;}
    virtual B operator () (A x){
          return r;};};
```

  t^{C++} := new t(x1, ..., xk).

  Therefore the class definition of t has instance variables xi of type Ai. The constructors has one argument for each variable xi and sets the instance variable xi to the value of that argument. The class has one method operator() with one argument x of type A. When invoked, the body of this method r, which is the translation of the body of the λ-term, is evaluated in the environment mapping x to the value of the argument of the method, and xi to the value of this instance variable. Note that no other variables are visible in the body of this method, since this environment might differ between when an object of this class was created and when it is used. That is the reason why one needs to copy first, when creating an object of such a class, the environment into some instance variables.[6]

  When applying an object of this class to an element, the body of the λ-term is invoked. The λ-term is translated into the statement which creates a new object with the instance variables set to the value they have in the current environment.

- Assume $t = r\ s$. Then the class definitions of t consist of the class definitions for r, and the class definitions for s (where the class definitions

[5] Strictly speaking, r^{C++} depends on the choice of identifiers for λ-types and C++-classes representing λ-terms. When defining the parse function P in Sect. 6, this will be made explicit by having the dependency of this function on the context Γ and the class environment C. Since in our abstract setting λ-types are represented by themselves, P does not depend on the choice of identifiers for those types.

[6] In C++ there are no inner classes like in Java, which allow references to the current environment.

corresponding to λ-abstractions occurring in both r and s need only to be introduced once).[7] Furthermore $t^{C++} := (*(r^{C++}))(s^{C++})$.

So t is interpreted as the result of applying the translation of r to the translation of s.

- Assume $t = f[r_1, \ldots, r_k]$. Then the class definitions for t are the class definitions for r_i (again class definitions for λ-terms occurring more than once need only to be introduced once). Furthermore, $t^{C++} := \mathtt{f}(r_1^{C++}, \ldots, r_k^{C++})$.

  So t is interpreted as the result of applying the native C++ function `f` to the translations of r_i.

Note that a λ-abstraction is interpreted as a function of its free variables in the form (new t(x₁, …, xₖ)). Hence, the evaluation of a λ-abstraction in an environment for the free variables is similar to a "closure" in implementations of functional programming languages.

So far, we have developed a program which parses λ-terms and translates them into the full language of C++. Our intention is to upgrade this to an extension of the language of C++ by λ-types and -terms together with a parser program which translates this extended language into native C++. For this purpose we introduce a syntax for representing λ-types and -terms in C++. We use functional style notation rather than overloading existing C++-notation, since we believe that this will improve readability and acceptability of our approach by functional programmers. In our extended language, we write `A -> B` for the function type $\mathtt{A} \to \mathtt{B}$, `r ^^s` for the application of `r` to `s`[8], and `\A x.B s` for λx^A.s if $s : B$. (If s is a term starting with λ, B will be omitted). For instance, the term $t = (\lambda f^{\mathsf{int}\to\mathsf{int}} \lambda x^{\mathsf{int}}.f\ (f\ x))\ (\lambda x^{\mathsf{int}}.x + 2)\ 3$ of the Introduction is written in our extended C++ syntax as

```
(\ int->int f. \ int x. int f^^(f^^x))^^(\ int x. int x+2)^^3
```

We now discuss informally how the translation of this example term t is evaluated. For the reader's convenience we repeat the generated C++ code in a reformatted form (the code shown in the introduction was the automatically generated output of our translation program):

```
class Cint_intD_aux { public : virtual int operator() (int x) = 0; };

typedef Cint_intD_aux*  Cint_intD;

class CCint_intD_Cint_intDD_aux
{  public : virtual Cint_intD operator() (Cint_intD x) = 0; };

typedef CCint_intD_Cint_intDD_aux*  CCint_intD_Cint_intDD;

class t1 : public Cint_intD_aux{
 public :Cint_intD f;
```

[7] A λ-abstraction is represented as a new instance of its corresponding class. Even if the classes for two occurrences of the same λ-abstraction coincide, for each occurrence a new instance is created. Therefore there is no problem, if a variable occurs as the same name, but with different referential meaning in two identical λ-expressions.

[8] Note that we cannot write r(s) here, since this notation will not translate into application, but into (* r)(s).

```
        t1( Cint_intD f)  {   this-> f = f;};
        virtual int operator () (int x) { return (*(f))((*(f))(x)); };};

class t0 : public CCint_intD_Cint_intDD_aux{
 public :t0( ) { };
        virtual Cint_intD operator () (Cint_intD f) {
          return new  t1( f); }};

class t2 : public Cint_intD_aux{
 public : t2( ) { };
         virtual int operator () (int x) { return x + 2; };};
```

When evaluating the expression t^{C++} = (*((*(new t0( )))(new t2( ))))(3); first the application of t0 to t2 is evaluated. To this end, instances l0, l2 of the classes t0 and t2 are created, and then the operator() method of l0 is called. This call creates an instance l1 of t1, with the instance variable f set to l2. The result of applying t0 to t2 is l1.

The next step in the evaluation of t^{C++} is to evaluate 3, and then to call the operator() method of l1. This will first make a call to the operator method of f, which is bound to l2, and apply it to 3. This will evaluate to 5. Then it will call the operator method of f again, which is still bound to l2, and apply it to the result 5. The result returned is 7.

We see that the evaluation of the expression above follows the call-by-value evaluation strategy.[9] Note that l0, l1, l2 were created on the heap, but have not been deleted afterwards. The deletion of l0, l1 and l2 relies on the use of a garbage collected version of C++, alternatively we could use smart pointers in order to enforce their deletion.

4 Modelling a Fragment of C++

In this section we construct a mathematical model of a fragment of C++ that contains the code created by the translation of λ-terms described in the previous section. We model the execution of C++ code by functions eval and apply, similar to the modelling of the λ-calculus in [ASS85]. However, in order to model the C++ implementation as truthfully as possible, we differ from [ASS85] by making the pointer structures for the classes and objects explicit and letting the functions eval and apply modify these pointer structures via side effects.

When we investigate what was needed from C++ in order to translate simply typed λ-terms, we see that the classes obtained have instance variables, one constructor, and one method corresponding to the operator() method. The constructor has one argument for each instance variable and sets the instance variables to these arguments. No other code is performed. The method has one argument, and the body consists of a simplified C++ expression. Here simplified C++ expressions are the C++ expressions occurring in the translation process, which were all translations of λ-terms. Simplified C++-expressions are variables, native C++ functions applied to simplified C++ expressions,

[9] Note that this computation causes some overhead, since for every subterm of the form $\lambda x.r$ a new object is created, which is in many cases used only once, and can be thrown away afterwards. One could optimise this, however at the price of having a much more complicated translation, and therefore a much more complex correctness proof of the translation.

the application of one simplified C++ expression to another simplified C++ expression (which corresponds to the method call in case the first applicative term is an object), and the construct `new` applied to a constructor and simplified C++ expressions.

We develop a language which formulates this fragment of C++. In this language, a class is given by a context representing its instance variables, the abstracted variable of the method and its type, and an applicative term. Applicative terms (which correspond to the simplified C++ expressions above) are variables, native C++ functions applied to applicative terms (where C++ functions with no arguments are constants), the application of one applicative term to another applicative term (which corresponds to the method call in case the first applicative term is an object), or a constructor applied to applicative terms (which corresponds to the `new`-construct).

This fragment could easily be extended in order to cover modification of instance variables and method calls in the body of a method, the possibility of having several methods, and other C++ constructs.

Definition 4.1 (Applicative terms, classes, class environments)

Let Constr *be an infinite set of* constructors *(i.e. class names), denoted by* c.

$$\begin{array}{llll} \textit{Applicative terms:} & \mathsf{App} \ni a, b & ::= & x \mid f[a_1, \ldots, a_k] \mid a\ b \mid c(a_1, \ldots, a_k) \\ \textit{Classes:} & \mathsf{Class} & ::= & (\Gamma; x : A; b) \\ \textit{Class environments:} & \mathsf{CEnv} \ni C & ::= & \mathsf{Constr} \rightarrow_{\mathsf{fin}} \mathsf{Class} \end{array}$$

Applicative terms ($\in$ App) correspond to the C++ constructs `x`, `f`$[\mathtt{a}_1, ..., \mathtt{a}_k]$, `(* (a))(b)` and `new` $\mathtt{c}(\mathtt{a}_1, \ldots, \mathtt{a}_k)$. A class $(\Gamma; x : A; b) \in \mathsf{Class}$, where $\Gamma = x_1 : A_1, \ldots, x_n : A_n$, corresponds to a C++ class definition of the form

```
class c : CA_BD_aux{
  public: A1 x1;
          ...
          An xk;
  c(A1 x1,A2 x2, ... , Ak xk){
                this->x1 = x1;
                ...
                this->xk = xk;}
  virtual B operator () (A x){
               return b;};};
```

Note that the type `B` is omitted in $(\Gamma; x : A; b)$ since it can be derived, and the class name `c` is associated with the class through the class environment CEnv.

Definition 4.2 *We define the free variables* $\mathrm{FV}(a)$ *of an applicative term* $a \in \mathsf{App}$ *as follows:* $\mathrm{FV}(x) = \{x\}$, $\mathrm{FV}(f[a_1, \ldots, a_k]) := \mathrm{FV}(a_1) \cup \cdots \cup \mathrm{FV}(a_k)$, $\mathrm{FV}(a\ b) := \mathrm{FV}(a) \cup \mathrm{FV}(b)$, $\mathrm{FV}(c(a_1, \ldots, a_k)) := \mathrm{FV}(a_1) \cup \cdots \cup \mathrm{FV}(a_k)$.

When a constructor call of a class is evaluated, its arguments are first evaluated. Then, memory for the instance variables of this class is allocated on the heap, and these instance variables are set to the evaluated arguments. The address to this memory location is the result returned by evaluating this constructor call. The only other possible result of the evaluation of an applicative term is a number, so values are addresses or numbers.

Note as well that on the heap we store elements of the form $c(\vec{v})$, which can be represented as elements of $\mathsf{Constr} \times \mathsf{Val}^*$ (here Val is the set of values). Therefore we can model the heap as a finite function from addresses to $\mathsf{Constr} \times \mathsf{Val}^*$.

Definition 4.3 (Values, closures, heaps, and value environments)

Let Addr *be an infinite set of* addresses *denoted by h.*
Let n range over C++ constants, that is, elements of basic C++ types;

Values:	$\mathsf{Val} \ni v, w$	$::=$	$n \mid h$
Closures:	$\mathsf{Constr} \times \mathsf{Val}^*$	$::=$	$(c, v_1, \dots, v_k)$
Heaps:	$\mathsf{Heap} \ni H$	$::=$	$\mathsf{Addr} \to_{\mathsf{fin}} (\mathsf{Constr} \times \mathsf{Val}^*)$
Value environments	$\mathsf{VEnv} \ni \eta$	$::=$	$\mathsf{Var} \to_{\mathsf{fin}} \mathsf{Val}$

Note that n denotes both constants and elements of basic C++-types. Since constants are to be interpreted by themselves, this doesn't cause confusion.

The functions eval and apply defined below have side effects on the heap. This fact can be conveniently expressed using monads.

Definition 4.4 (State monad)

The partial state monad *for a given sets X (of states) is the functor* $\mathsf{M}_X\colon \mathsf{Set} \to \mathsf{Set}$ *(the object part of which is)*

$$\mathsf{M}_X(Y) := X \overset{\sim}{\to} Y \times X$$

where $X \overset{\sim}{\to} Y \times X$ is the set of partial functions from X to $Y \times X$.

Elements of $\mathsf{M}_X(Y)$ are called *actions* and can be viewed as elements of Y that may depend on a current state $x \in X$ and also may change the current state. So, an element of $\mathsf{M}_X(Y)$ is a partial function which, depending on the current state, returns a result and a new state (or fails). Monads are a category-theoretic concept whose computational significance was discovered by Moggi [Mog91].

We need to work with partial instead of total functions because the operations eval and apply defined below do not yield defined results in general. We will however prove that for inputs that can be typed the results will always be defined.

Notation 4.5 (do, return, mapM, read, add) *We use the following standard monadic notation (roughly following Haskell syntax): Suppose $e_1 : \mathsf{M}_X(Y_1)$, ..., $e_{k+1} : \mathsf{M}_X(Y_{k+1})$ are actions where e_i may depend on $y_1 : Y_1$, ..., $y_{i-1} : Y_{i-1}$. Then*

$$\mathsf{do}\{y_1 \leftarrow e_1 \ ; \ \dots \ ; \ y_k \leftarrow e_k \ ; \ e_{k+1}\} : \mathsf{M}_X(Y_{k+1})$$

is the action that maps any state $y_0 : Y$ to (y_{k+1}, x_{k+1}) where $(y_i, x_i) \simeq e_i\ x_{i-1}$, for $i = 1, \dots, k+1$ ($\simeq$ denotes the usual "partial equality"). The intuitive idea is that the do*-expression is computed by evaluating $e_0, \dots, e_{k+1}$ in sequence, where e_i can make use of the result y_j returned by e_j ($j < i$). The result returned is that of e_{k+1}, and the computation of each e_i might change the state.*

We also allow let-expressions with pattern matching within a do-construct (with the obvious meaning). We adopt the convention that computations are "strict", i.e. the result of a computation is undefined if one of its parts is. Furthermore, we use the standard monadic notations

$$\begin{array}{ll} \mathsf{return} : Y \to \mathsf{M}_X(Y) & \mathsf{return}\ y\ x = (y, x) \\ \mathsf{mapM} : (Z \to \mathsf{M}_X(Y)) \to Z^* \to \mathsf{M}_X(Y^*) & \mathsf{mapM}\ f\ \vec{z} = \mathsf{do}\{y_1 \leftarrow f\ z_1 \ ; \ \dots \\ & \quad \dots \ ; \ y_k \leftarrow f\ z_k \ ; \ \mathsf{return}\ (\vec{y})\} \end{array}$$

as well as

$$\begin{array}{ll} \mathsf{read} : X \to \mathsf{M}_{X \to_{\mathsf{fin}} Y}(Y), & \mathsf{read}\ x\ m \simeq (m\ x, m) \\ \mathsf{add} : Y \to \mathsf{M}_{X \to_{\mathsf{fin}} Y}(X), & \mathsf{add}\ y\ m \simeq (x, m[x \mapsto y]) \quad \textit{where } x = \mathsf{fresh}(m) \end{array}$$

Here, fresh *is a function with the property that if* $m : X \to_{\mathsf{fin}} Y$, *then* $\mathsf{fresh}(m) \in X \setminus \mathsf{dom}(m)$ [10].

Definition 4.6 *We define functions*

$$\begin{array}{rl} \mathsf{eval} : & \mathsf{CEnv} \to \mathsf{VEnv} \to \mathsf{App} \to \mathsf{M}_{\mathsf{Heap}}(\mathsf{Val}) \\ \mathsf{apply} : & \mathsf{CEnv} \to \mathsf{Val} \to \mathsf{Val} \to \mathsf{M}_{\mathsf{Heap}}(\mathsf{Val}) \end{array}$$

by mutual recursion as follows (in Sect. 6 we will omit the argument C, since it will be a global parameter):

$$\begin{array}{rcl} \mathsf{eval}\ C\ \eta\ x & = & \mathsf{return}\ (\eta\ x) \\ \mathsf{eval}\ C\ \eta\ f[\vec{a}] & = & \mathsf{do}\{\vec{n} \leftarrow \mathsf{mapM}\ (\mathsf{eval}\ C\ \eta)\ \vec{a}\ ;\ \mathsf{return}\ [\![f]\!](\vec{n})\} \\ \mathsf{eval}\ C\ \eta\ (a\ b) & = & \mathsf{do}\{(v, w) \leftarrow \mathsf{mapM}(\mathsf{eval}\ C\ \eta)\ (a, b)\ ;\ \mathsf{apply}\ C\ v\ w\} \\ \mathsf{eval}\ C\ \eta\ c(\vec{a}) & = & \mathsf{do}\{\vec{v} \leftarrow \mathsf{mapM}\ (\mathsf{eval}\ C\ \eta)\ \vec{a}\ ;\ \mathsf{add}\ (c, \vec{v})\} \\ & & \\ \mathsf{apply}\ C\ h\ v & = & \mathsf{do}\{(c, \vec{w}) \leftarrow \mathsf{read}\ h\ ;\ \mathsf{let}\ (\vec{y} : \vec{B}; x : A; a) = C\ c \\ & & \qquad\qquad\qquad\qquad\quad \mathsf{in}\ \mathsf{eval}\ C\ [\vec{y}, x \mapsto \vec{w}, v]\ a\} \\ \mathsf{apply}\ C\ n\ v & = & \emptyset \end{array}$$

where $\emptyset$ is the undefined action, i.e. the partial function with empty domain[11].

Lemma 4.7 *(1) If* $\mathsf{eval}\ C\ \eta\ a\ H = (v, H')$, *then* $H \subseteq H'$.

(2) If $\mathsf{apply}\ C\ v\ w\ H = (v', H')$, *then* $H \subseteq H'$.

Proof. Straightforward simultaneous induction on the definitions of eval and apply, i.e. by "fixed point induction" [Win93]. **q.e.d.**

Due to the complexity of C++ it would be a major task, which would require much more man power than was available in our research group, to formally prove that our mathematical model, given by eval and apply, coincides with the operational semantics of C++.[12] (Note that other models of fragments of object-oriented languages in the literature face the same problem and their correctness w.r.t. real languages is therefore usually not shown.) However, when going through the definitions we observe that the evaluation function eval is indeed defined in accordance with the expected behaviour of C++:

- A variable is evaluated by returning its value in the current environment η.

[10] In our applications X will be a space of addresses which we assume to be infinite, i.e. we assume that the allocation of a new address is always possible.

[11] It would be more appropriate to let apply $C\ n\ v$ result in a finite error, but, for simplicity, we identify errors with non-termination.

[12] The formalisation of the semantics of Java in [SSB01] was a major project, and still this book excludes some features of Java like inner classes. Note that C++ is much more complex than Java.

- The application of a native C++ function to arguments $a_1, \ldots, a_k$ is carried out by first evaluating $a_1, \ldots, a_k$ in sequence, and then applying the function f to those arguments.

- Note that constants are special cases of functions with arity 0, and therefore constants are evaluated by themselves.

- $(a\ b)$ corresponds in C++ to the construct $(*\ (a))(b)$. First a and b are evaluated. Because of type correctness, a must be an element of the type of pointers to a class, and the value of a will therefore be an address on the heap. On the heap the information about the class used and the values of the instance variables of that class are stored. Then $(*\ (a))(b)$ is computed by evaluating the body of the method of the class in the environment where the instance variables have the values as stored on the heap, and the abstracted variable has the result of evaluating b. This is what is computed by $\mathsf{eval}\ \eta\ (a\ b)$ (which makes use of the auxiliary function apply).

- The expression $c(\vec{a})$, which stands for the C++ expression new $\mathtt{c}(\mathtt{a}_1, \ldots, \mathtt{a}_k)$, is evaluated by first computing $\mathtt{a}_1, \ldots, \mathtt{a}_k$ in sequence. Then new storage on the heap is allocated. Note that in our simplified setting, the constructor of $\mathtt{c}$ simply assigns to the instance variables the values of $\mathtt{a}_1, \ldots, \mathtt{a}_k$. Consequently, the intended behaviour of C++ is that it stores on the heap the information about the class used and the result of evaluating $\mathtt{a}_1, \ldots, \mathtt{a}_k$, which is what is carried out by eval.

5 Formal Translation of Typed λ-Terms and its Correctness

Despite of the fact that we could describe only informally the connection of our mathematical model with the actual implementation of C++, we will be able to prove formally that the model as well as the translation of λ-terms described in Section 3 are correct in the following sense: As we did for λ-terms, we will define for C++ terms, $a \in \mathsf{App}$, a typing relation, $\Gamma \vdash a : A$, and a denotational semantics, $[\![a]\!]^H \xi \in \mathsf{D}(A)$. Similarly, we will define for values, $v \in \mathsf{Val}$, a relation $H \vdash v : A$ and a semantics $[\![v]\!]^H \in \mathsf{D}(A)$ (all these definitions will depend on a class environment $C \in \mathsf{CEnv}$). Our main results will be the correctness of the translation function, P (see below), and the evaluation function, eval, with respect to these typing relations and denotational semantics (Theorems 5.6 and 6.9).

In this section we carry out the first step, namely the introduction of the parsing relation and a proof that it is correct and complete. In the next section we will show that the evaluation of applicative terms is correct as well and obtain the correctness of our implementation.

Definition of the Parse Function P

We are going to define a formal analogue to the translation of λ-terms described in Section 3. We use the monadic notation from Section 4.

Definition 5.1 (Definition of the Parse Function P) *We define a function*

$$\mathsf{P}: \quad \mathsf{Context} \to \mathsf{Term} \to \mathsf{M}_{\mathsf{CEnv}}(\mathsf{App})$$

by recursion on terms as follows:

$$\begin{array}{rcl}
\mathsf{P}\,\Gamma\,x & = & \mathsf{return}\ x, \textit{ if } x \textit{ is a variable} \\
\mathsf{P}\,\Gamma\,f[\vec{r}] & = & \mathsf{do}\{\vec{a} \leftarrow \mathsf{mapM}\ (\mathsf{P}\,\Gamma)\ \vec{r}\,;\ \mathsf{return}\ f[\vec{a}]\} \\
\mathsf{P}\,\Gamma\,(r\ s) & = & \mathsf{do}\{(a,b) \leftarrow \mathsf{mapM}\ (\mathsf{P}\,\Gamma)\ (r,s)\ ;\ \mathsf{return}\ (a\ b)\} \\
\mathsf{P}\,\Gamma\,(\lambda x^A.r) & = & \mathsf{do}\{a \leftarrow \mathsf{P}\ (\Gamma, x : A)\ r\ ;\ c \leftarrow \mathsf{add}(\Gamma; x : A; a)\ ;\ \mathsf{return}\ c(\mathsf{dom}(\Gamma))\}
\end{array}$$

Hence, the translation has a side effect on the class environment.

Lemma 5.2 $\mathsf{P}\,\Gamma\,r$ *is total and if* $\mathsf{P}\,\Gamma\,r\,C = (a, C')$*, then* $C \subseteq C'$*.*

Proof. Induction on the term r. **q.e.d.**

Typing and denotational semantics of applicative terms

Definition 5.3 (Typing of Applicative Terms) *We define inductively a typing relation* $C; \Gamma \vdash a : A$ *:*

$$C; \Gamma, x : A \vdash x : A$$

$$\frac{\begin{array}{c} f \in \mathsf{F},\ f : (\rho_1, \ldots, \rho_k) \to \sigma \\ C; \Gamma \vdash a_i : \rho_i \quad (i = 1, \ldots, k) \end{array}}{C; \Gamma \vdash f[a_1, \ldots, a_k] : \sigma}$$

$$\frac{C; \Gamma \vdash a : A \to B \qquad C; \Gamma \vdash b : A}{C; \Gamma \vdash a\,b : B}$$

$$\frac{\begin{array}{c} C(c) = (\Delta; x : A; a) \quad \Delta = x_1 : A_1, \ldots, x_k : A_k \\ C; \Delta, x : A \vdash a : B \\ C; \Gamma \vdash a_i : A_i \quad (i = 1, \ldots, k) \end{array}}{C; \Gamma \vdash c(a_1, \ldots, a_k) : A \to B}$$

Definition 5.4 (Denotational semantics of applicative terms) *If* $C; \Gamma \vdash a : A$*, then for every functional environment* $\xi \in \mathsf{FEnv}$ *such that* $\xi : \Gamma$ *we define* $[\![a]\!]^C \xi \in \mathsf{D}(A)$ *(we write* $[\![a]\!]\xi$ *if* C *is a fixed global parameter):*

$$\begin{array}{rcl}
[\![x]\!]^C \xi & := & \xi(x) \\
[\![f[a_1, \ldots, a_k]]\!]^C \xi & := & [\![f]\!]([\![a_1]\!]^C \xi, \ldots, [\![a_k]\!]^C \xi) \\
[\![a\,b]\!]^C \xi & := & [\![a]\!]^C \xi([\![b]\!]^C \xi) \\
[\![c(a_1, \ldots, a_k)]\!]^C \xi & := & \lambda d \in \mathsf{D}(A).[\![a]\!]^C \xi'[x \mapsto d]
\end{array}$$

where in the last clause it is assumed that we have $C(c) = (\Delta; x : A; a)$ *with* $C; \Delta, x : A \vdash a : B$*,* $\Delta = x_1 : A_1, \ldots, x_k : A_k$ *and* $C; \Gamma \vdash a_i : A_i$ *(*$i = 1, \ldots, k$*), and* ξ' *is defined by* $\xi'(x_i) := [\![a_i]\!]^C \xi$ *for* $i = 1, \ldots, k$*.*

Lemma 5.5 *(a) If* $C \subseteq C'$*,* $\Gamma \subseteq \Gamma'$*,* $C; \Gamma \vdash a : A$*, then* $C'; \Gamma' \vdash a : A$*.*

(b) If $C;\Gamma \vdash a : A$, $C;\Gamma \vdash a : A'$, then $A = A'$.

(c) If $C \subseteq C'$, $\Gamma \subseteq \Gamma'$, $\xi \subseteq \xi'$, $C;\Gamma \vdash a : A$, and $\xi : \Gamma$, $\xi' : \Gamma'$, then $\llbracket a \rrbracket^{C}\xi = \llbracket a \rrbracket^{C'}\xi'$.

Proof: Straightforward.

Correctness of the Parse Function P

Theorem 5.6 *If r is a λ-term, $\Gamma \vdash r : A$, then $\mathsf{P}\ \Gamma\ r\ C \simeq (a, C')$ for some C', a such that $C';\Gamma \vdash a : A$, and for all $\xi : \Gamma$ we have $\llbracket r \rrbracket\xi = \llbracket a \rrbracket^{C'}\xi$.*

Proof: Induction on the derivation of $\Gamma \vdash r : A$. The only interesting case is $r = \lambda x^B.r'$, where we have $\Gamma, x : B \vdash r' : A$. Assume $\Gamma = x_1 : A_1, \ldots, x_k : A_k$ and let $\vec{x} := x_1, \ldots, x_k$. By induction hypothesis $\mathsf{P}\ (\Gamma, x : B)\ r'\ C = (a', C_1)$ for some C_1, a' s.t. $C_1;\Gamma, x : B \vdash a' : A$, and for $d \in \mathsf{D}(A)$ we have $\llbracket r' \rrbracket\xi[x \mapsto d] = \llbracket a' \rrbracket^{C_1}\xi[x \mapsto d]$. Then $\mathsf{P}\ \Gamma\ r\ C = (c(\vec{x}), C_2)$, where $C_2 := C_1[c \mapsto (\Gamma; x : B; a')]$, for some fresh c. $C_2;\Gamma \vdash x_i : A_i$, and by monotonicity $C_2;\Gamma, x : B \vdash a' : A$, therefore $C_2;\Gamma \vdash c(\vec{x}) : B \to A$. Furthermore, $\llbracket x_i \rrbracket^{C_2}\xi = \xi(x_i)$, therefore $\xi' \subseteq \xi$ for the ξ' as in the definition of $\llbracket c(\vec{x}) \rrbracket^{C_2}\xi$. Therefore, $\llbracket c(\vec{x}) \rrbracket^{C_2}\xi = \boldsymbol{\lambda} d.\llbracket a' \rrbracket^{C_2}\xi'[x \mapsto d] = \boldsymbol{\lambda} d.\llbracket a' \rrbracket^{C_1}\xi[x \mapsto d] = \boldsymbol{\lambda} d.\llbracket r' \rrbracket\xi[x \mapsto d] = \llbracket r \rrbracket\xi$. **q.e.d.**

Completeness of the Parse Function P

In addition to the correctness of the translation function P we now show the opposite direction, namely completeness: The translated versions of typed λ-terms are already essentially all typed elements of App. The only restriction is that constructors are only applied to variables, and that they are applied to all variables in the context, independently of whether the variables occur in the body of the class or not.

Definition 5.7 *Let $C;\Gamma \vdash' a : A$ be defined by the same rules as for $C;\Gamma \vdash a : A$, except for the rule of deriving $C;\Gamma \vdash c(a_1, \ldots, a_k) : A \to B$, which is replaced by the following:*

$$\frac{C(c) = (\Gamma; x : A; a) \quad \Gamma = x_1 : A_1, \ldots, x_k : A_k \qquad C;\Gamma, x : A \vdash' a : B}{C;\Gamma \vdash' c(x_1, \ldots, x_k) : A \to B}$$

Remark 5.8 *In Theorem 5.6 we have as well $C';\Gamma \vdash' a : A$.*

We have no control over the choice of class names (constructors) introduced by the parse function. So a class term will in general only be reached by the parse function up to renaming of class names. Furthermore, if in a λ-term there exist the same λ-term twice as a subterm, the parse function will assign different class names to each occurrence of it. (One could improve the parse function so that this doesn't take place.) Therefore, if we want to obtain an element a of App by parsing a λ-term r, it might be that in the parsed λ-term a' there are two different constructors which correspond to the same constructor in a. So we obtain an element of App by parsing a λ-term only up to renaming and possibly identification of class names. The following definition of a class homomorphism makes this explicit:

Definition 5.9 *(a) Let $\theta : \mathsf{Constr} \to_{\mathsf{fin}} \mathsf{Constr}$. Then $\theta(a)$ is defined if each constructor occurring in a is an element of $\mathsf{dom}(\theta)$. If $\theta(a)$ is defined, then $\theta(a)$ is the result of replacing each occurrence of $c \in \mathsf{Constr}$ in a by $\theta(c)$. Furthermore, for $(\Gamma; x : B; a) \in \mathsf{Class}$ we define $\theta(\Gamma; x : B; a) :\simeq (\Gamma; x : B; \theta(a))$.*

(b) Let $C, C' \in \mathsf{CEnv}$. $\theta : \mathsf{dom}(C') \to \mathsf{dom}(C)$ is a CEnv-homomorphism, if $\forall c \in \mathsf{dom}(C').\theta(C'(c)) = C(\theta(c))$.

Theorem 5.10 (Completeness of the Parse Function P) *Assume $C; \Gamma \vdash' a : A$, and $C' \in \mathsf{CEnv}$. Then there exists a λ-term r, a CEnv-homomorphism $\theta : C'' \to C$ such that $\mathsf{dom}(C'') \cap \mathsf{dom}(C') = \emptyset$, and an $a' \in \mathsf{App}$, such that the following holds:*

$$\begin{array}{rcl} \Gamma & \vdash & r : A \ , \\ \mathsf{P}\,\Gamma\, r\, C' & = & (a', C' \cup C'') \ , \\ C' \cup C''; \Gamma & \vdash' & a' : A \ , \\ \theta(a') & = & a \ . \end{array}$$

Proof: Induction on $C; \Gamma \vdash' a : A$. Case $\Gamma = x_1 : A_1, \ldots, x_k : A_k$, $a = c(x_1, \ldots, x_k)$, $A = A' \to B'$, $C(c) = (\Gamma; x : A'; a')$, $C; \Gamma, x : A' \vdash' a' : B'$. Let $C' \in \mathsf{CEnv}$. By induction hypothesis, we find a CEnv-homomorphism $\theta' : C_0 \to C$, a λ-term r' and $a'' \in \mathsf{App}$ such that $\Gamma, x : A' \vdash r' : B'$, $\mathsf{P}\ (\Gamma, x : A')\ r'\ C' = (a'', C' \cup C_0)$, $C' \cup C_0; \Gamma, x : A' \vdash' a'' : B'$, and $\theta'(a'') = a'$. Then $\mathsf{P}\ \Gamma\ (\lambda x^{A'}.r')\ C' = (c_0(x_1, \ldots, x_k), C' \cup C_0[c_0 \mapsto (\Gamma; x : A'; a'')])$ for some fresh c_0. Let $a' := c_0(x_1, \ldots, x_k)$, $C'' := C_0[c_0 \mapsto (\Gamma; x : B; a'')]$, $\theta := \theta'[c_0 \mapsto c]$, $r := \lambda x^{A'}.r'$.

The other cases are straightforward. **q.e.d.**

6 Correctness of the Evaluation of Applicative Terms

In this Section, except for the main theorem 6.10 at the end, the class environment C will not change. We will therefore omit this parameter in all notations (including apply, eval).

Typing, Semantics, and Semantic Typing of Values

Definition 6.1 (Typing of Values) *The typing relation $H \vdash v : A$ is defined inductively by the following rules:*

If ρ is a native C++ type and $n \in [\![\rho]\!]$, then

$$H \vdash n : \rho$$

If $h \in \mathsf{Addr}$, $H(h) = (c, v_1, \ldots, v_k)$, $C(c) = (\Delta; x : A; a)$
$\Delta = x_1 : A_1, \ldots, x_k : A_k$ then

$$\frac{\begin{array}{c}\Delta, x : A \vdash a : B \\ H \vdash v_i : A_i \quad (i = 1, \ldots, k)\end{array}}{H \vdash h : A \to B}$$

Definition 6.2 (Denotational Semantics of Values) *(a) Assuming $H \vdash v : A$ we define the denotational semantics of v, $\llbracket v \rrbracket^{H} \in \mathsf{D}(A)$, by recursion on the derivation of $H \vdash v : A$:*

$$\begin{array}{lcl} \llbracket n \rrbracket^{H} & := & n \\ \llbracket h \rrbracket^{H} & := & \boldsymbol{\lambda} d \in \mathsf{D}(A).\llbracket a \rrbracket \xi[x \mapsto d] \end{array}$$

where $H(h) = (c, v_1, \ldots, v_k)$, $C(c) = (\Delta; x : A; a)$, $\Delta = x_1 : A_1, \ldots, x_k : A_k$, $\Delta, x : A \vdash a : B$, $H \vdash v_i : A_i$ and $\xi(x_i) := \llbracket v_i \rrbracket^{H}$ $(i = 1, \ldots, k)$.

If C needs to be mentioned we write $\llbracket v \rrbracket^{C,H}$ instead of $\llbracket v \rrbracket^{H}$.

(b) If $\eta \in \mathsf{FEnv}$, $\eta : \Gamma$, we define $\llbracket \eta \rrbracket^{H} \in \mathsf{VEnv}$ by $\llbracket \eta \rrbracket^{H} := \boldsymbol{\lambda} x \in \mathsf{dom}(\eta).\llbracket \eta(x) \rrbracket^{H}$.

Lemma 6.3 *(a) If $H \vdash v : A$ and $H \subseteq H'$, then $H' \vdash v : A$ and $\llbracket v \rrbracket^{H} = \llbracket v \rrbracket^{H'}$.*

(b) If $H \vdash v : A$ and $H \vdash v : A'$, then $A = A'$.

The next definition is motivated by the following consideration: We want to show that $\mathsf{eval}\ \eta\ a\ H$ is defined, whenever we have $\Gamma \vdash a : A$, and if the result is (v, H'), then $\llbracket a \rrbracket = \llbracket v \rrbracket^{H'}$ (more precisely we have a dependency on an environment η). The problem is that $\llbracket a\ b \rrbracket = \llbracket a \rrbracket(\llbracket b \rrbracket)$, whereas $\mathsf{eval}\ \eta\ (a\ b)\ H :\simeq \mathsf{apply}\ v\ w\ H''$ where v, w are obtained by applying eval to a and b, and H'' is the heap obtained when evaluating a and b. Even so in the proof of the correctness of eval we might know by induction hypothesis (for some suitable heap H''') that $\llbracket a \rrbracket = \llbracket v \rrbracket^{H'''}$ and $\llbracket b \rrbracket = \llbracket w \rrbracket^{H'''}$, we will not be able to conclude that $\llbracket a \rrbracket(\llbracket b \rrbracket) = \llbracket v' \rrbracket^{H'''}$, if $\mathsf{apply}\ v\ w\ H'' \simeq (v', H''')$, unless we know for a already that it respects apply. If $a = c(a_1, \ldots, a_k)$, this is no problem, but a might be a variable or an application term $a'\ b'$.

So, when proving the correctness for an applicative term of type $A \to B$, we need to show that it respects apply as well. We can achieve this if we know that apply is only applied to terms which respect apply themselves. We also need to assume that all variables have this property as well. The correct condition is expressed by the following Kripke-style logical relation $H \models v \sim d : A$ between a C++ value $v \in \mathsf{Val}$ and a denotational value ($d \in \mathsf{D}(A)$). This relation, which depends on a class environment C, a heap H, and the type A, can be viewed as a semantic analogue of the (proof-theoretic) typing relation.

Definition 6.4 *(a) We define a relation $H \models v \sim d : A$ by recursion on the type A.*

For the sake of readability we will write $\models \mathsf{apply}\ v\ w\ H \sim d : A$ as a shorthand for

$$\exists v', H'\ (\mathsf{apply}\ v\ w\ H \simeq (v', H') \wedge H' \models v' \sim d : A).$$

(Note that $\models \mathsf{apply}\ v\ w\ H \sim d : A$ implies $\mathsf{apply}\ v\ w\ H$ is defined):

$$\begin{array}{rcl} H \models v \sim n : \rho & :\Longleftrightarrow & v = n \in \llbracket \rho \rrbracket \\ H \models v \sim f : A \to B & :\Longleftrightarrow & H \vdash v : A \to B \wedge \llbracket v \rrbracket^{H} = f \\ & & \wedge \forall H' \supseteq H, \forall (w, d) \in \mathsf{Val} \times \mathsf{D}(A). \\ & & \quad H' \models w \sim d : A \\ & & \quad \Longrightarrow \models \mathsf{apply}\ v\ w\ H' \sim f(d) : B \end{array}$$

(b) We also define

$$H \models \eta \sim \xi : \Gamma :\Longleftrightarrow \mathsf{dom}(\Gamma) \subseteq \mathsf{dom}(\eta) \cap \mathsf{dom}(\xi) \wedge \\ \forall x \in \mathsf{dom}(\Gamma) H \models \eta(x) \sim \xi(x) : \Gamma(x)$$

Remark. Strictly speaking, in the notation $H \models v \sim d : A$, the parameter d is superfluous, since we have $H \models v \sim d : A \Rightarrow d = [\![v]\!]^H$. So we could have written instead $H \models v : A$. The reason for using the slightly redundant notation is that it facilitates the (frequent) references to d.

Lemma 6.5 *If $H \models v \sim d : A$ and $H \subseteq H'$ then $H' \models v \sim d : A$.*

Proof. Easy by the definition and Lemma 6.3 (a). **q.e.d.**

Lemma 6.6 *If $H \models v \sim d : A$, then $H \vdash v : A$. Hence, if $H \models \eta \sim \xi : \Gamma$, then $H \vdash \eta : \Gamma$.*

Proof. Clear, by definition. **q.e.d.**

Proof of Correctness of eval

The main result below corresponds to the usual "Fundamental Lemma" or "Adequacy Theorem" for logical relations.

Similarly to the notation $\models \mathsf{apply}\ v\ w\ H \sim d : A$ we write $\models \mathsf{eval}\ \eta\ a\ H \sim d : A$ as a shorthand for

$$\exists v', H' \ (\mathsf{eval}\ \eta\ a\ H \simeq (v', H') \wedge H' \models v' \sim d : A).$$

Lemma 6.7 *Assume $\Gamma \vdash a : A$. Then for all η, ξ we have*

$$H \models \eta \sim \xi : \Gamma \Longrightarrow \models \mathsf{eval}\ \eta\ a\ H \sim [\![a]\!]\xi : A \ .$$

Proof. The proof is by induction on the typing judgement $\Gamma \vdash a : A$. In the proof we will refer to Lemma 6.5 as "monotonicity". We consider the four possible cases of how $\Gamma \vdash a : A$ can be derived.

$\Gamma, x : A \vdash x : A$. Assume $\xi : (\Gamma, x : A)$ and $H \models \eta \sim \xi : (\Gamma, x : A)$. We need to show $\models \mathsf{eval}\ \eta\ x\ H \sim [\![x]\!]\xi : A$. We have $\mathsf{eval}\ \eta\ x\ H = (\eta(x), H)$ and $[\![x]\!]\xi = \xi(x) \in \mathsf{D}(A)$. Furthermore, $H \models \eta \sim \xi : (\Gamma, x : A)$ entails $H \models \eta(x) \sim \xi(x) : A$, and therefore the assertion follows.

$\Gamma \vdash c(\vec{a}) : A \to B$, derived from $C(c) = (\Delta; x : A; a)$, where $\Delta = \vec{y} : \vec{A}$, $\Gamma \vdash \vec{a} : \vec{A}$, and $\Gamma, x : A \vdash a : B$. Assume $\xi : \Gamma$ and $H \models \eta \sim \xi : \Gamma$.

We need to show $\models \mathsf{eval}\ \eta\ c(\vec{a})\ H \sim [\![c(\vec{a})]\!]\xi : A \to B$. By induction hypothesis for $\Gamma \vdash a_i : A_i$ and monotonicity we get that $\mathsf{mapM}\ (\mathsf{eval}\ \eta)\ \vec{a}\ H = (\vec{v}, H_1)$ for some H_1, $\vec{v}$ s.t. $H_1 \models v_i \sim [\![a_i]\!]\xi : A_i$ Therefore $\mathsf{eval}\ \eta\ c(\vec{a})\ H = (h, H')$, where $H' = H_1[h \mapsto (c, \vec{v})]$) with $h = \mathsf{fresh}(H_1)$. We need to show $H' \models h \sim [\![c(\vec{a})]\!]\xi : A \to B$, which is a conjunction of three statements (i), (ii), (iii):

(i) We need to show $H' \vdash h : A \to B$, which follows, since $H'(h) = (c, \vec{v})$, $C(c) = (\Delta; x : A; a)$, $\Gamma, x : A \vdash a : B$ and $H' \vdash v_i : A_i$, where the last statement follows by $H_1 \models v_i \sim [\![a_i]\!]\xi : A_i$.

(ii) We need to show $[\![h]\!]^{H'} = [\![c(\vec{a})]\!]\xi$:

$$\begin{aligned} [\![h]\!]^{H'} &= \lambda d \in \mathsf{D}(A).[\![a]\!][\![\eta_0]\!]^{H'}[x \mapsto d] \ , \\ [\![c(\vec{a})]\!]\xi &= \lambda d \in \mathsf{D}(A).[\![a]\!]\xi_0[x \mapsto d] \end{aligned}$$

where $\eta_0 := [\vec{y} \mapsto \vec{v}]$, $[\![\eta_0]\!]^{H'} := \lambda y \in \mathsf{dom}(\eta_0).[\![\eta_0(y)]\!]^{H'}$, and $\xi_0 = [\vec{y} \mapsto [\![\vec{a}]\!]\xi]$. By $H_1 \models v_i \sim [\![a_i]\!]\xi : A_i$ and monotonicity we have $H' \models \eta_0 \sim \xi_0 : \Delta$. Therefore $[\![\eta_0(y_i)]\!]^{H'} = \xi_0(y_i)$ and we are done.

(iii) Assume $H'' \supseteq H'$ and $H'' \models w \sim d : A$. We need to show

$$\models \mathsf{apply}\ h\ w\ H'' \sim ([\![c(\vec{a})]\!]\xi)(d) : B \ .$$

First,

$$\begin{aligned} \mathsf{apply}\ h\ w\ H'' &\simeq \mathsf{eval}\ \eta_0[x \mapsto w]\ a\ H'' \ , \\ ([\![c(\vec{a})]\!]\xi)(d) &= [\![a]\!]\xi_0[x \mapsto d] \ . \end{aligned}$$

By $H' \models \eta_0 \sim \xi_0 : \Delta$, $H'' \models w \sim d : A$, and monotonicity we obtain $H'' \models \eta_0[x \mapsto w] \sim \xi_0[x \mapsto d] : (\Delta, x : A)$. Using the induction hypothesis we obtain $\models \mathsf{eval}\ \eta_0[x \mapsto w]\ a\ H'' \sim [\![a]\!]\xi_0[x \mapsto d] : B$.

$\Gamma \vdash a\ b : B$, derived from $\Gamma \vdash a : A \to B$ and $\Gamma \vdash b : A$. Assume $\xi : \Gamma$ and $H \models \eta \sim \xi : \Gamma$. We need to show $\models \mathsf{eval}\ \eta\ (a\ b)\ H \sim [\![a\ b]\!]\xi : A$. By induction hypothesis and Lemma 4.7 (1), $\mathsf{eval}\ \eta\ a\ H = (v, H_1)$ for some $H_1 \supseteq H$ with $H_1 \models v \sim [\![a]\!]\xi : A \to B$ and, using monotonicity, $\mathsf{eval}\ \eta\ b\ H_1 = (w, H_2)$ for some $H_2 \supseteq H_1$ with $H_2 \models w \sim [\![b]\!]\xi : A$. By the definition of $H_1 \models v \sim [\![a]\!]\xi : A \to B$ we obtain $\models \mathsf{apply}\ v\ w\ H_2 \sim [\![a]\!]\xi([\![b]\!]\xi) : B$ and we are done, since $\mathsf{eval}\ \eta\ (a\ b)\ H \simeq \mathsf{apply}\ v\ w\ H_2$ and $[\![a\ b]\!]\xi = [\![a]\!]\xi([\![b]\!]\xi)$.

$\Gamma \vdash f[a_1, \dots, a_k] : B$, derived from $\Gamma \vdash a_i : A_i$, $i = 1, \dots, k$, where $f : (A_1, \dots, A_k) \to B$. Assume $\xi : \Gamma$ and $H \models \eta \sim \xi : \Gamma$. We need to show $\models \mathsf{eval}\ \eta\ f[\vec{a}]\ H \sim [\![f[\vec{a}]]\!]\xi : B$. By induction hypothesis and Lemma 4.7 (1), $\mathsf{eval}\ \eta\ a_1\ H = (n_1, H_1)$ for some $n_1 \in [\![A_1]\!]$ and $H_1 \supseteq H$ with $H_1 \models n_1 \sim [\![a_1]\!]\xi : A_1$, especially $n_1 = [\![a_1]\!]\xi$. Similarly, using monotonicity and Lemma 4.7 (1), for $i = 1, \dots, k-1$ we have $\mathsf{eval}\ \eta\ a_{i+1}\ H_i = (n_{i+1}, H_{i+1})$ for some $n_{i+1} \in [\![A_i]\!]$ and $H_{i+1} \supseteq H_i$ with $n_{i+1} = [\![a_{i+1}]\!]\xi$. It follows $\mathsf{eval}\ \eta\ f[\vec{a}]\ H = ([\![f]\!](\vec{n}), H_k) = ([\![f[\vec{a}]]\!]\xi, H_k)$. **q.e.d.**

Lemma 6.8 *If $H \vdash v : A$ then $H \models v \sim [\![v]\!]^H : A$.*

Proof. Induction on $H \vdash v : A$.

The case $A \in \mathsf{basetype}$ is trivial.

Case $H \vdash h : A \to B$, derived from $H(h) = (c, \vec{v})$, $C(c) = (\vec{x} : \vec{A}; x : B; a)$, $\vec{x} : \vec{A}, x : A \vdash a : B$, $H \vdash \vec{v} : \vec{A}$. We need to show that for $H' \supseteq H$, w, d s.t. $H' \models w \sim d : A$ we have $\models \mathsf{apply}\ h\ w\ H' \sim [\![h]\!]^H(d) : B$. So assume H', w, d as stated.

Let $\eta := [\vec{x} \mapsto \vec{v}]$, $\xi := [\![\eta]\!]^H$. Then $[\![h]\!]^H(d) = [\![h]\!]^{H'}(d) = [\![a]\!]\xi[x \mapsto d]$. By induction hypothesis $H \models v_i \sim [\![v_i]\!]^H : A_i$, and by $H' \models w \sim d : A$ we therefore get $H' \models \eta[x \mapsto w] \sim \xi[x \mapsto d] : \Gamma$. Furthermore, $\mathsf{apply}\ h\ w\ H' \simeq \mathsf{eval}\ (\eta[x \mapsto w])\ a\ H'$. By Lemma 6.7 $\models \mathsf{eval}\ (\eta[x \mapsto w])\ a\ H' \sim [\![a]\!]\xi[x \mapsto d] : B$, which proves the assertion.

Theorem 6.9 (Correctness of eval) *If* $\Gamma \vdash a : A$ *and* $H \vdash \eta : \Gamma$*, then there exist* H', v *such that* $\mathsf{eval}\ \eta\ a\ H \simeq (v, H')$, $H' \vdash v : A$ *and* $[\![a]\!]([\![\eta]\!]^{H'}) = [\![v]\!]^{H'}$.

Proof: Immediate by Lemmas 6.7 and 6.8.

Main Theorem

Theorem 6.10 (Overall Correctness) *Assume* $\vdash r : A$ *and let* $C \in \mathsf{CEnv}$*. Then* $\mathsf{P}\ \emptyset\ r\ C = (a, C')$ *for some* $C' \supseteq C$*. Furthermore, for any heap* H*, any environment* η *and any* $C'' \supseteq C'$ *we have*

$$\mathsf{eval}\ C''\ \eta\ a\ H = (v, H')$$

for some H'*,* v *s.t.* $[\![r]\!]\emptyset = [\![v]\!]^{C,H'}$.

Especially, in case $A = \mathsf{int}$ *we have that* $r \twoheadrightarrow_\beta n$ *for some* n*, and therefore* $[\![r]\!]\emptyset = n$ *and*

$$\mathsf{eval}\ C''\ \eta\ a\ H = (n, H') \enspace .$$

Proof. By Theorem 5.6 and 6.9.

Remark. In the correctness proof for our implementation we had to explicitely manipulate the class environment and the heap, which made the proof rather "low level". It would be desirable, in particular with regard to a formalisation in a proof assistant, to lift the proof to the same abstract monadic level at which the functions P, eval and apply are defined. A framework for carrying this out might be provided by suitable versions of Moggi's Computational λ–Calculus, Pitts' Evaluation Logic [Pit91] and special logical relations for monads [GLN02].

7 Conclusion

In this paper we showed how to introduce functional concepts into C++ in a provably correct way. The modelling and the correctness proof used monadic concepts as well as denotational semantics and logical relations.

This work lends itself to a number of extensions, for example, the integration of recursive higher-order functions, polymorphic and dependent type systems, the integration lazy evaluation and infinite structures as well as the combination of larger parts of C++ with the λ-calculus. The accurate description of these extensions would require more sophisticated, e.g. domain-theoretic constructions. We believe that if our approach is extended to cover full C++, we obtain a language in which the worlds of functional and object-oriented programming are merged, and that we will see many examples, where the combination of both language concepts (e.g. the use of λ-terms with side-effects) will result in interesting new programming techniques.

The remarkable fact that it is possible to have a denotational semantics at a description level where pointers are manipulated explicitely entails that the well-known benefits of denotational semantics, extensionality and compositionality, are still available at that level. This has already paid off in this paper where we were able to give a short and concise correctness proof for our C++ fragment using the denotational semantics (instead of a complicated operational argument). More benefits are to be expected when it comes to verifying programs written in this C++ fragment or in one of the future extensions mentioned above.

8 List of Identifiers and Notations

Finite functions	$X \rightharpoonup_{\mathsf{fin}} Y$	Def. 1.1
Extension of fin. fns.	$g[x \mapsto y]$ or $f, x : y$	Def. 1.1
Explicitly given fin. fns.	$x_1 : y_1, \dots, x_k : y_k$	Def. 1.1
Base types	$\mathsf{basetype} \ni \rho, \sigma$	Ass. 2.1 (a)
Base type int	int	Ass. 2.1 (a)
Basic functions	$\mathsf{F} \ni f$	Ass. 2.1 (b)
Denot. sem. of base type	$[\![\rho]\!]$	Ass. 2.1 (d)
Denot. sem. of basic fn.	$[\![f]\!]$	Ass. 2.1 (e)
Constants (of any base type)	n	Ass. 2.1 (c)
Variables	$\mathsf{Var} \ni x, y, z$	Def. 2.2 (a)
Types	$\mathsf{Type} \ni A, B$	Def. 2.2 (b)
Contexts	$\mathsf{Context} \ni \Gamma, \Delta$	Def. 2.2 (b)
Extension of Contexts	$\Gamma, x : A$	Def. 2.2 (b)
λ-terms	$\mathsf{Term} \ni r, s, t$	Def. 2.2 (b)
Typed λ-terms	$\Gamma \vdash r : A$	Def. 2.2 (b)
Functionals of type A	$\mathsf{D}(A) \ni d$	Def. 2.3
Functional environment	$\mathsf{FEnv} \ni \xi$	Def. 2.4 (a)
Typed contexts	$\xi : \Gamma$	Def. 2.4 (b)
Denot. sem. of λ-terms	$[\![r]\!] \xi$	Def. 2.5
Name of a type	$\mathsf{name}(A)$	Sect. 3
Translation of λ-terms	$t^{C{+}{+}}$	Sect. 3
Constructors or classnames	$\mathsf{Constr} \ni c$	Def. 4.1
Applicative terms	$\mathsf{App} \ni a, b$	Def. 4.1
Classes	$\mathsf{Class} \ni (\Gamma; x : A; b)$	Def. 4.1
Class environments	$\mathsf{CEnv} = \mathsf{Constr} \ni C$	Def. 4.1
Free variables of a	$\mathrm{FV}(a)$	Def. 4.2
Heap addresses	$\mathsf{Addr} \ni h$	Def. 4.3
C++ constants	n	Def. 4.3
Values	$\mathsf{Val} \ni v, w$	Def. 4.3
Closures	$\mathsf{Constr} \times \mathsf{Val}^* \ni (c, v_1, \dots, v_k)$	Def. 4.3
Heaps	$\mathsf{Heap} = \mathsf{Addr} \ni H$	Def. 4.3
Value environments	$\mathsf{VEnv} \ni \eta$	Def. 4.3
State Monad	$\mathsf{M}_X(Y)$	Def. 4.4
Monadic notations	$\mathsf{do}, \mathsf{return}, \mathsf{mapM}, \mathsf{read}, \mathsf{add}$	Not. 4.5
Fresh element	$\mathsf{fresh}(X)$	Not. 4.5
Eval and Apply	$\mathsf{eval}, \mathsf{apply}$	Def. 4.6
Parse function	P	Def. 5.1
Typed applicative terms	$C; \Gamma \vdash a : A$	Def. 5.3
Denot. sem. of appl. terms	$[\![a]\!]^C \xi$ or $[\![a]\!] \xi$	Def. 5.4
Variant of above	$C; \Gamma \vdash' a : A$	Def. 5.7
CEnv-homomorphism	θ	Def. 5.9 (b)
Typed values	$H \vdash v : A$	Def. 6.1
Denot. sem. of values	$[\![v]\!]^H$, $[\![v]\!]^{C,H}$	Def. 6.2 (a)
Denot. sem. of value envs.	$[\![\eta]\!]^H$	Def. 6.2 (b)
Semantic typing of values	$H \models v \sim d : A$	Def. 6.4
Special not. for the above	$\models \mathsf{apply}\ v\ w\ H \sim d : A$	
	$\models \mathsf{eval}\ \eta\ a\ H \sim d : A$	Def. 6.4

References

[ASS85] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and interpretation of computer programs.* MIT Press, 1985.

[GLN02] J. Goubault-Larrecq, S. Lasota, and D. Nowak. Logical relations for monadic types. In Julian C. Bradfield, editor, *Proceedings of the 16th International Workshop on Computer Science Logic (CSL'02)*, volume 2471 of *Lecture Notes in Computer Science*, pages 553–568, Edinburgh, Scotland, UK, September 2002. Springer.

[IPW99] Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In Loren Meissner, editor, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34(10), pages 132–146, N. Y., 1999.

[JT93] A. Jung and J. Tiuryn. A new characterization of lambda definability. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 245–257. Springer, 1993.

[Kis98] O. Kiselyov. Functional style in C++: Closures, late binding, and lambda abstractions. In *ICFP '98: Proceedings of the third ACM SIGPLAN International conference on Functional programming*, page 337, New York, NY, USA, 1998. ACM Press.

[Läu95] K. Läufer. A framework for higher-order functions in C++. In *COOTS*, 1995.

[Mog91] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[MS00] B. McNamara and Y. Smaragdakis. Functional programming in C++. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 118–129, New York, NY, USA, 2000. ACM Press.

[Pit91] A. M. Pitts. Evaluation logic. In G. Birtwistle, editor, *IVth Higher Order Workshop, Banff 1990*, Workshops in Computing, pages 162–189. Springer, 1991.

[Plo77] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[Plo80] G. D. Plotkin. Lambda definability in the full type hierarchy. In R. Hindley and J. Seldin, editors, *To H.B. Curry: Essays in Combinatory Logic, lambda calculus and Formalisms*, pages 363 – 373. Academic Press, 1980.

[Rey83] J. C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP'83*, pages 513–523. North-Holland, 1983.

[SSB01] R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine - Definition, Verification, Validation.* Springer, 2001.

[Sta85] R. Statman. Logical relations and the typed lambda-calculus. *Information and Control*, 65:85 - 97, 1985.

[Str] J. Striegnitz. FACT! – the functional side of C++. http://www.fz-juelich.de/zam/FACT.

[Tai67] W. W. Tait. Intensional interpretation of functionals of finite type. *Journal of Symbolic Logic*, 32:198 - 212, 1967.

[Win93] G. Winskel. *The formal semantics of programming languages: an introduction.* MIT Press, Cambridge, MA, USA, 1993.

Appendix D

Functional Concepts in C++

Functional Concepts in C++

Rose H. Abdul Rauf, Ulrich Berger, Anton Setzer

Abstract

We describe a parser-translator program that translates typed λ-terms into C++ classes so as to integrate functional concepts. We prove the correctness of the translation of λ-terms into C++ with respect to a denotational semantics using Kripke-style logical relations. We introduce a general technique for introducing lazy evaluation into C++, and illustrate it by carrying out in C++ the example of computing the Fibonacci numbers efficiently using infinite streams and lazy evaluation. Finally, we show how merge higher-order λ-terms with imperative C++ code.

1 Introduction

C++ is a general purpose language that supports object oriented programming as well as procedural and generic programming, but unfortunately not directly functional programming. We have developed a parser-translator program that translates typed λ-term into C++ statements so as to integrate functional concepts. This translated code uses the object oriented approach of programming that involves creation of classes for the λ-term. That a translated λ-term is an element of the translation of a function type is achieved by using inheritance.

The paper is organised as follows: First, we introduce the translation and discuss how the translated code is executed including a description of the memory allocation (Sect. 2). The correctness of our implementation is proved with respect to the usual (set-theoretic) denotational semantics of the simply typed λ-calculus on the one hand, and a mathematical model of a sufficiently large fragment of C++ on the other hand. The proof is based on a Kripke-style logical relation between the C++ class and the denotational model (Sect. 3). In Sect. 4 we introduce a general technique for introducing lazy evaluation into C++ by introducing a data type of lazy elements of type A. Finally, in Sect. 5 we discuss a few general features arising if we allow λ-terms in C++ to have side-effects.

Related work. Several researchers [Kis98], [Läu95] have discovered that C++ can be used for functional programming by representing higher order functions using classes. Our representation is based on similar ideas. There are other approaches that have made C++ a language that can be used for functional programming such as the FC++ library [MS00] (a very elaborate approach) as well as FACT! [Str] (extensive use of templates and overloading) and [Kis98] (creating macros that allow creation of single macro-closure in C++). The advantages of our solution are that it is very simple, it uses classes and inheritance in an essential way, it can be used for implementing λ-terms with side-effects, and, most importantly, we have a formal correctness proof.

The approach of using denotational semantics and logical relations in a proof of the correctness of programs has been used before by Plotkin [Plo77] and others. The method of logical relations can be traced back at least to Tait [Tai67] and has been used for various purposes (e.g. Jung and Tiuryn [JT93], Statman [Sta85] and Plotkin [Plo80]). To our knowledge the verification of the implementation of the λ-calculus in C++ using logical relations is new.

Lazy evaluation in C++ has been studied in the literature (see [Sch00], [MS00], [Kel97]). To our knowledge all implementations are restricted to lazy lists, whereas we introduce a general type of lazy elements of an arbitrary type.

2 Translation of typed λ-terms into C++

In this section we describe how to translate simply typed λ-terms into C++ using the object-oriented concepts of classes and inheritance.

The ***simply typed** λ**-calculus*** over the base type of integers with constants for arithmetic functions, *λ-calculus* for short, is given as follows: *Types* are built from the base type, Int, and and function types, $A \to B$. *Terms* are of the form x (variables), n (numerals $\in \mathrm{N} = \{0, 1, 2, , \ldots\}$), $\lambda x^A r$ (abstraction), $r\ s$ (application), $f[r_1, \ldots, r_n]$ (function application). In the last case f is taken from a set F of names of number-theoretic C++ functions. A *context* is a finite set of pairs $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ (all x_i distinct) which is, as usual, identified with a finite map. We let Type, Var, Term, Context denote the set of types, variables, terms and contexts, respectively. The *typing rules* are as expected:

$$\Gamma, x : A \vdash x : A \qquad \Gamma \vdash n : \mathrm{Int}$$

$$\frac{\Gamma, x : A \vdash r : B}{\Gamma \vdash \lambda x^A r : A \to B} \qquad \frac{\Gamma \vdash r : A \to B \qquad \Gamma \vdash s : A}{\Gamma \vdash r\ s : B}$$

$$\frac{\Gamma \vdash r_1 : \mathrm{Int} \ldots \Gamma \vdash r_k : \mathrm{Int}}{\Gamma \vdash f[r_1, \ldots, r_k] : \mathrm{Int}} \quad (f \text{ a } k\text{-ary arithmetic function})$$

Rather than explaining in general how a λ-term is translated into its equivalent C++ statements, we describe the translation of the example term

$$t = (\lambda f^{\mathrm{Int} \to \mathrm{Int}} \lambda x^{\mathrm{Int}}. f\ (f\ x))\ (\lambda x^{\mathrm{Int}}. x + 2)\ 3\ .$$

The concrete ASCII notation for this term (i.e. the input for the parser program) is

```
(\int->int f. \int x. int f^^(f^^x))^^(\int x. int x+2)^^3 ·
```

For each function type we first define an abstract class with a virtual operator that will be overloaded in the definition of the λ-term. The type itself is the type of pointers to an object of this class. In the class names the letters `C` and `D` are used to represent opening and closing brackets and an underscore stands for an arrow. Hence `Cint_intD` means Int $\to$ Int. The type of t, (Int $\to$ Int) $\to$ (Int $\to$ Int), is represented in stages:[1]

[1] In order to obtain fast compilation of the translated code, we do not use templates in our machine-generated code. This will as well ease the correctness proof to be given in Sect. 3. If one defines λ-types by hand it is of course easier to define a general C++-template for the class corresponding to the type $X \to Y$.

```
class Cint_intD_aux
{ public : virtual int operator() (int x) = 0; };

typedef Cint_intD_aux*  Cint_intD;

class CCint_intD_Cint_intDD_aux
{  public : virtual Cint_intD operator()
                            (Cint_intD x) = 0; };

typedef CCint_intD_Cint_intDD_aux*
        CCint_intD_Cint_intDD;
```

The term t is translated in stages as well. The subterm $\lambda x^{\mathrm{Int}}.f\ (f\ x)$ is translated as an instance of the class Cint_intD_aux:

```
class lambda1 : public Cint_intD_aux{
 public :Cint_intD f;
 lambda1( Cint_intD f)  {   this-> f = f;};
 virtual int operator () (int x)
 { return (*(f))((*(f))(x)); };
};
```

The subterm $\lambda f^{\mathrm{Int}\to\mathrm{Int}}\lambda x^{\mathrm{Int}}.f(fx)$ is translated as follows:

```
class lambda0 : public CCint_intD_Cint_intDD_aux{
 public :
 lambda0( ) { };
 virtual Cint_intD operator () (Cint_intD f)
 { return new  lambda1( f); }
};
```

The other λ-term, $\lambda x^{\mathrm{Int}}.2 + x$, is translated in a similar way:

```
 class lambda2 : public Cint_intD_aux{
 public :
 lambda2( ) { };
 virtual int operator () (int x)
 { return x + 2; };
};
```

The defining equation for t will be finally translated into the expression

```
int t = (*((*( new lambda0( )))( new lambda2( ))))(3);
```

When evaluating the expression t, first the application of lambda0 to lambda2 is evaluated. For this first instances l0, l2 of the classes lambda0 and lambda2 are created. Then the operator()-method of l0 is called. That call will create an instance l1 of lambda1, with the instance variable f set to l2. The result of application of lambda0 to lambda2 is l1.

The next step in the evaluation of t is to evaluate 3, which evaluates to 3, and then to call the operator() method of l1. This will first make a call to the operator method of f, which is bound to l2, and apply it to 3. This will evaluate to 5. Then it will call the operator method of f again, which is still bound to l2, and apply it to the result 5. The result returned is 7.

We see that the evaluation of the expression above follows the call-by-value evaluation strategy. Note that l0, l1, l2 were created on the heap, but have not been deleted afterwards. The deletion of 10, 11 and 12 relies on the use of a garbage collected version of C++, otherwise we could use smart pointers in order to enforce their deletion.

3 Proof of Correctness

We sketch how to prove the correctness of our C++ implementation of the λ-calculus. By "correctness" we mean that every closed term r of type Int is evaluated by our implementation to a numeral which coincides with the *value* of r. The value of a term can be defined either *operationally* as the normal form w.r.t. β-reduction, $(\lambda x^A r)s \to_\beta r[s/x]$, and function reduction, $f[n_1, \ldots, n_k] \to_f n$ (n the value of f at $n_1, \ldots, n_k$), or, equivalently, *denotationally* as the natural value in a suitable domain of functionals of finite types. Since our calculus doesn't allow for recursive definitions, the details of the operational and denotational semantics do not matter: Operationally, any sufficiently complete reduction strategy (call-by-value, call-by-name, full normalisation) will do, and denotationally, any Cartesian closed category containing the type of integers can be used. For our purposes it will be most convenient to work with a denotational model. For simplicity we take the naive set-theoretic hierarchy D of functionals of finite types over the integers (setting $\mathrm{N} = \{0, 1, 2, \ldots\}$ and $X \to Y = \{f \mid f : X \to Y\}$):

$$\mathrm{D}(\mathrm{Int}) = \mathrm{N}, \quad \mathrm{D}(A \to B) = \mathrm{D}(A) \to \mathrm{D}(B), \quad D = \bigcup_{A \in \mathrm{Type}} \mathrm{D}(A)$$

A *functional environment* is a mapping $\xi : \mathrm{Var} \to \mathrm{D}$. FEnv denotes the set of all functional environments. If Γ is a context, then $\xi : \Gamma$ means $\forall x \in \mathrm{dom}(\Gamma).\xi(x) \in \mathrm{D}(\Gamma(x))$.

For every typed λ-term $\Gamma \vdash r : A$ and every functional environment $\xi : \Gamma$ the denotational value $[\![r]\!]\xi \in \mathrm{D}(A)$ is defined as follows:

i) $[\![n]\!]\xi = n$

ii) $[\![x]\!]\xi = \xi(x)$

iii) $[\![r\ s]\!]\xi = [\![r]\!]\xi([\![s]\!]\xi)$

iv) $[\![\lambda x^A.r]\!]\xi(a) = [\![r]\!]\xi[x \mapsto a]$

v) $[\![f[\vec{r}]]\!] = [\![f]\!]([\![\vec{r}]\!]\xi)$

where in the last clause $[\![f]\!]$ is the number-theoretic function denoted by f. Our implementation of the λ-calculus is modelled in a similar way as e.g. in [ASS85] using functions eval and apply. In order to model the C++ implementation as truthfully as possible, we will introduce a simplified model of C++, in which we will make the pointer structures for the classes and objects explicit. The functions eval and apply modify these pointer structures via side effects.

In our model all classes will have instance variables, one constructor, and one method corresponding to the operator() method. The constructor has one argument for each instance variable, and will set the instance variables to these arguments. No other code is performed. The method has one argument, and the body consists of an applicative term, where applicative terms are simplified

C++ expressions in our model. So, a class is given by a context representing its instance variables, the abstracted variable of the method and its type, and an applicative term.

Applicative terms will be numbers, variables, function terms applied to applicative terms, the application of one applicative term to another applicative term (which corresponds to the method call in case the first applicative term is an object), or a constructor applied to applicative terms.

When a constructor call of a class is evaluated, its arguments are first evaluated. Then, memory for the instance variables of this class will be allocated on the heap, and these instance variables will be set to the evaluated arguments. The address to this memory location is the result returned by evaluating this constructor call. The only other possible result of the evaluation of an applicative term is a number, so values are addresses or numbers.

The data sets associated with our model of C++ classes are defined as follows (letting $X + Y$ and $X \times Y$ denote the disjoint sum and Cartesian product of X and Y, X^* the set of finite lists of elements in X and $X \rightarrow_{\mathrm{fin}} Y$ the set of finite maps from X to Y):

$$\begin{array}{lcl}
\mathrm{Addr} & = & \text{a set of numbers denoting addresses of classes on the heap} \\
\mathrm{Constr} & = & \text{a set of strings denoting constructors, i.e. class names} \\
\mathrm{Val} & = & \mathrm{N} + \mathrm{Addr} \\
\mathrm{F} & = & \text{a set of names for arithmetic C++ functions} \\
\mathrm{App} & = & \mathrm{N} + \mathrm{Var} + \mathrm{F} \times \mathrm{App}^* + \mathrm{App} \times \mathrm{App} + \mathrm{Constr} \times \mathrm{App}^* \\
\mathrm{Context} & = & \mathrm{Var} \rightarrow_{\mathrm{fin}} \mathrm{Type} \\
\mathrm{Class} & = & \mathrm{Context} \times \mathrm{Var} \times \mathrm{Type} \times \mathrm{App} \\
\mathrm{VEnv} & = & \mathrm{Var} \rightarrow_{\mathrm{fin}} \mathrm{Val} \\
\mathrm{Heap} & = & \mathrm{Addr} \rightarrow_{\mathrm{fin}} \mathrm{Constr} \times \mathrm{Val}^* \\
\mathrm{CEnv} & = & \mathrm{Constr} \rightarrow_{\mathrm{fin}} \mathrm{Class}
\end{array}$$

We write applicative terms ($\in$ App) that are neither numbers nor variables as $f[\vec{a}]$, $a\ b$ and $c[\vec{a}]$. Classes ($\in$ Class) are written as $(\Gamma; x : A; r)$.

The fact that the parsing function as well as the functions eval and apply have side effects on the classes and the heap can be conveniently expressed using the familiar *state monad*

$$\mathrm{M}_X(Y) := X \rightarrow Y \times X$$

Elements of $\mathrm{M}_X(Y)$ are usually called *actions* and can be viewed as elements of Y that may depend on a current state $x \in X$ and also may change the current state. Monads are a category-theoretic concept whose computational significance was discovered by Moggi [Mog91]. The functionalities of the parsing function P and the operations eval and apply can now be written as

$$\begin{array}{rl}
\mathrm{P}: & \mathrm{Context} \rightarrow \mathrm{Term} \rightarrow \mathrm{M}_{\mathrm{CEnv}}(\mathrm{App}) \\
\mathrm{eval}: & \mathrm{CEnv} \rightarrow \mathrm{VEnv} \rightarrow \mathrm{App} \rightarrow \mathrm{M}_{\mathrm{Heap}}(\mathrm{Val}) \\
\mathrm{apply}: & \mathrm{CEnv} \rightarrow \mathrm{Val} \rightarrow \mathrm{Val} \rightarrow \mathrm{M}_{\mathrm{Heap}}(\mathrm{Val})
\end{array}$$

Hence, parsing has a side effect on the class environment, while eval and apply have side effects on the heap.

Strictly speaking, the definitions above (and below) are not quite accurate since, for example, applicative terms ($\in$ App) are untyped, but are supposed to represent valid (and hence typed) C++ expressions. A corresponding typing discipline could easily be introduced, but would probably be more distracting than insightful. Similarly, actions as well as the functions P, eval and apply should be modelled by partial rather than total functions. For example, in order for an expression of the form apply $C\ v\ w$ to make sense the value, v, must not be a numeral, n, but an address, h, in the heap (pointing to the C++ implementation of a λ-abstraction). The defining equation for apply reflects the assumption that this is the case. Similar soundness assumptions are tacitly made elsewhere. In case these assumptions are violated it is assumed that the computation results in an error. Again, the error cases could be modelled by a suitable monad, and one could show that, if we refer to correctly typed C++ expressions, this error never occurs. For the sake of brevity, we refrain from carrying this out. There is as well another reason for partiality, namely the fact that C++ programs can be recursive and therefore evaluation of C++ programs, even if correctly typed, might not terminate. However, it is a consequence of Theorem 3.1 that the evaluation of parsed λ-terms terminates and does not result in an error.

Also note that the use of monads for describing the C++ implementation is just a notational convenience. We do not make any assumptions of monadic structures to exist in C++.

We use the following standard monadic notation (roughly following Haskell syntax): Suppose $e_1 : \mathrm{M}_X(Y_1)$, ..., $e_{k+1} : \mathrm{M}_X(Y_{k+1})$ are actions where e_i may depend on $y_1 : Y_1$, ..., $y_{i-1} : Y_{i-1}$. Then

$$\mathrm{do}\{y_1 \leftarrow e_1 \ ; \ \dots \ ; \ y_k \leftarrow e_k \ ; \ e_{k+1}\} : \mathrm{M}_X(Y_{k+1})$$

is the action that maps any state $x_0 : X$ to (y_{k+1}, x_{k+1}) where $(y_i, x_i) = e_i\ x_{i-1}$, for $i = 1, \dots, k+1$. If y_i does not occur in any e_j with $j > i$, then we suppress "$y_i \leftarrow$". We also allow let-expressions with pattern matching within a do-construct. Furthermore, we use

$$\begin{array}{ll}
\mathrm{return} : Y \to \mathrm{M}_X(Y) & \mathrm{return}\ y\ x = (y, x) \\
\mathrm{mapM} : (Z \to \mathrm{M}_X(Y)) \to Z^* \to \mathrm{M}_X(Y^*) & \mathrm{mapM}\ f\ \vec{a} = \mathrm{do}\{y_1 \leftarrow f\ a_1 \ ; \ \dots \\
& \quad \dots \ ; \ y_k \leftarrow f\ a_k \ ; \ \mathrm{return}\ (y_1, \dots, y_k)\} \\
\mathrm{get} : \mathrm{M}_X(X), & \mathrm{get}\ x = (x, x) \\
\mathrm{put} : X \to \mathrm{M}_X(\{*\}) & \mathrm{put}\ x\ x' = (*, x)
\end{array}$$

With these notations the definitions of P, eval and apply read as follows (we use a function fresh with the property that if $m : X \to_{\mathrm{fin}} Y$, where X is infinite, then $\mathrm{fresh}(m) \in X \setminus \mathrm{dom}(m)$):

$$\begin{array}{rcl}
\mathrm{P}\ \Gamma\ u & = & \mathrm{return}\ u, \text{ if } u \text{ is a numeral or a variable} \\
\mathrm{P}\ \Gamma\ f[\vec{r}] & = & \mathrm{do}\{\vec{a} \leftarrow \mathrm{mapM}\ (\mathrm{P}\ \Gamma)\ \vec{r}\ ; \ \mathrm{return}\ f[\vec{a}]\} \\
\mathrm{P}\ \Gamma\ (r\ s) & = & \mathrm{do}\{(a, b) \leftarrow \mathrm{mapM}\ (\mathrm{P}\ \Gamma)\ (r, s)\ ; \ \mathrm{return}\ (a\ b)\} \\
\mathrm{P}\ \Gamma\ (\lambda x^A.r) & = & \mathrm{do}\{a \leftarrow \mathrm{P}\ \Gamma[x \mapsto A]\ r\ ; \ C \leftarrow \mathrm{get}\ ; \ \mathrm{let}\ c = \mathrm{fresh}(C)\ ; \\
& & \quad \mathrm{put}(C[c \mapsto (\Gamma; x : A; a)])\ ; \ \mathrm{return}(c[\mathrm{dom}(\Gamma)])\}
\end{array}$$

$$\begin{array}{rcl}
\text{eval}\ C\ \eta\ n & = & \text{return}\ n \\
\text{eval}\ C\ \eta\ x & = & \text{return}\ (\eta\ x) \\
\text{eval}\ C\ \eta\ f[\vec{a}] & = & \text{do}\{\vec{n} \leftarrow \text{mapM}\ (\text{eval}\ C\ \eta)\ \vec{a}\ ;\ \text{return}\ [\![f]\!](\vec{n})\} \\
\text{eval}\ C\ \eta\ (a\ b) & = & \text{do}\{(v,w) \leftarrow \text{mapM}(\text{eval}\ C\ \eta)\ (a,b)\ ;\ \text{apply}\ C\ v\ w\} \\
\text{eval}\ C\ \eta\ c[\vec{a}] & = & \text{do}\{\vec{v} \leftarrow \text{mapM}\ (\text{eval}\ C\ \eta)\ \vec{a}\ ;\ H \leftarrow \text{get}\ ;\ \text{let}\ h = \text{fresh}(H)\ ; \\
 & & \quad \text{put}(H[h \mapsto (c,\vec{v})])\ ;\ \text{return}(h)\} \\
\text{apply}\ C\ h\ v & = & \text{do}\{H \leftarrow \text{get}\ ;\ \text{let}\ (c,\vec{w}) = H\ h\ ;\ \text{let}\ (\vec{y} : \vec{B}; x : A; a) = C\ c\ ; \\
 & & \quad \text{eval}\ C\ [\vec{y}, x \mapsto \vec{w}, v]\ a\}
\end{array}$$

The correctness proof of the translated code is based on a Kripke-style relation between the C++ representation of a term ($\in$ Val × Heap) and its denotational value ($\in \mathrm{D}(A)$). The relation is indexed by the class environment C and the type A of the term. Since in the case of an arrow type, $A \to B$, extensions of the heap and the class environment have to be taken into account, this definition has some similarity with Kripke models. The relation

$$\sim^C_A \subseteq (\text{Val} \times \text{Heap}) \times \mathrm{D}(A),$$

where $A \in \text{Type}, C \in \text{CEnv}$, is defined by recursion on A as follows:

$$\begin{array}{rcl}
(v,H) \sim^C_{\text{Int}} n & :\Longleftrightarrow & v = n \\
(v,H) \sim^C_{A \to B} f & :\Longleftrightarrow & \forall C \subseteq C', \forall H \subseteq H', \forall (w,d) \in \text{Val} \times \mathrm{D}(A) : \\
 & & (w,H') \sim^{C'}_A d \Longrightarrow \text{apply}\ C'\ v\ w\ H' \sim^{C'}_B f(d)
\end{array}$$

We also set $(\eta, H) \sim^C_\Gamma \xi :\Longleftrightarrow \forall x \in \text{dom}(\Gamma)(\eta(x), H) \sim^C_{\Gamma(x)} \xi(x)$.

The main result below corresponds to the usual "Fundamental Lemma" or "Adequacy Theorem" for logical relations:

Theorem 3.1 *Assume* $\eta : \text{VEnv}, \xi : \text{FEnv}, \Gamma \vdash r : A$. *Assume* $\xi : \Gamma$, $\mathrm{P}\ \Gamma\ r\ C = (a, C')$, $C' \subseteq C''$, $(\eta, H) \sim^{C''}_\Gamma \xi$, *and* $H \subseteq H'$. *Then* $\text{eval}\ C''\ \eta\ a\ H' \sim^{C''}_A [\![r]\!]\xi$.

The theorem can be proven by induction on the typing judgement $\Gamma \vdash r : A$. Due to limited space we omit details.

Corollary 3.2 (Correctness of the implementation) *Assume* $\vdash r : \text{Int}$, $\mathrm{P}\ \emptyset\ r\ \emptyset = (a, C)$ *and* $C \subseteq C'$. *Then for any heap* H *and environment* η *we have* $\text{eval}\ C'\ \eta\ a\ H = ([\![r]\!], H')$ *for some* $H' \supseteq H$.

4 Lazy Evaluation in C++

Haskell is famous for its programming techniques using infinite lists. A well-known example are the Fibonacci numbers, which are computed efficiently by using the following code:

```
fib = 1:1:(zipWith (+) fib (tail fib))
```

This example requires that we have infinite streams of natural numbers, and relies heavily on lazy evaluation. We will show how to translate this code into efficient C++ code. This requires that we are able to deal with lazy evaluation.

The standard technique for replacing call-by-value by call-by-name is to replace types A by $() \to A$ where $()$ is the empty type (i.e. `void`). This delays execution, but does not cater for the reuse of evaluated expressions, so that an expression is only evaluated once. In order to obtain this, we will define a new type `Lazy(A)`. This type delays evaluation of an element of type A in such a way that, if needed, the evaluation is carried out – however, only once. Once the value is computed, the result is stored in a variable for later reuse. The definition is as follows (we make use of the extended C++ syntax introduced in Sect. 2, especially `r ^^ t` for functional application, `\` for λ, and `A -> B` for the type of functions from `A` to `B`):

```
template<typename X> class lazy{
  bool is_evaluated;
  union {X        result;
         () -> X compute_function;};
public:
  lazy(()-> X compute_function){
    is_evaluated = false;
    this->compute_function = compute_function;};
  X eval(){
    if (not is_evaluated){
            result = compute_function ^^ ();
            is_evaluated = true;};
    return result;};};
#define Lazy(X) lazy<X>*
```

Using this class we can now easily define lazy streams of natural numbers (lazy lists, i.e. possibly terminating streams, can be defined similarly, but require the usual technique based on the composite design pattern for formalising algebraic data types as classes by introducing a main class for the main type which has subclasses for each constructor, each of which stores the arguments of the constructor)

```
template<typename X>class lazy_stream{
public: Lazy(X) head;
        Lazy(lazy_stream<X>*) tail;
        ... Constructor as usual ... }
#define Lazy_Stream(X) lazy_stream<X>*
```

In order to deal with the example of the Fibonacci numbers, one needs to define the operators used in the above mentioned definition of `fib`:

- `lazy_cons<X>` computes the cons-operation on streams:

  ```
  Lazy(Lazy_Stream(X)) lazy_cons<X>
          (Lazy(X) head, Lazy(Lazy_Stream(X)) tail)
  ```

- `lazy_tail<X>` computes the tail of a stream lazily:

  ```
  Lazy(Lazy_Stream(X)) lazy_tail<X>
                       (Lazy(Lazy_Stream(X)) s)
  ```

- `lazy_zip_with<X>` computes the usual zip_with function (i.e. $\text{zip\_with}(f,[a,b,..],[c,d,..]) = [f\ a\ c, f\ b\ d,\ldots]$):

  ```
  Lazy(Lazy_Stream(X)) lazy_zip_with<X>
      (X -> X -> X f,
       Lazy(Lazy_Stream(X)) s0,
       Lazy(Lazy_Stream(X)) s1)
  ```

The definition of these operation is straightforwards, once one has introduced a few combinators for dealing with `Lazy(X)`.

Now we can define the stream of Fibonacci numbers as follows (`plus` is $\lambda x, y.x + y$, `one_lazy` is the numeral 1 converted into an element of `Lazy(int)`, `create_lazy` transforms elements of type `()->A` into `Lazy(A)`, and `eval` evaluates an element of type `Lazy(A)` to an element of type `A`):

```
()-><Lazy_Stream(int)> fib_aux =
 \() x. Lazy_Stream(int)
        eval(
         lazy_cons(
           one_lazy,
           lazy_cons(
             one_lazy,
             lazy_zip_with(
               plus,
               create_lazy(this),
               lazy_tail(create_lazy(this))))));
Lazy_Stream(int) fib = eval(create_lazy(fib_aux))
```

Note that here we were using the keyword `this` in the definition of `fib_aux`. This is how a recursive call should be written. If we instead put `fib_aux` here, C++ will, when instantiating `fib_aux`, first instantiate `fib_aux` as an empty class, and then use this value when evaluating the right hand side. Only when using `this` we obtain a truely recursive definition.

When evaluated, one sees that the nth element of `fib` obtains fib(n) and this computation is the efficient computation in which previous calls of fib(k) are memorized. If one replaces `Lazy(X)` by `() -> X`, one obtains an implementation of the Fibonacci numbers, which computes the correct results. However, since that implementation doesn't memorize values, it has exponential running time and on our laptop we were not able to compute fib(25).

Generalization. The above technique can easily be generalized to general algebraic types, in fact to all class structures avilable in C++. If one replaces in a tree structure all types by lazy types, then only a trunk of the tree structure is evaluated and kept in memory, namely the trunk which has been used already by any function accessing this structure.

5 Combining Functional and Imperative Programming

When combining functional and imperative programming we obtain more than just the disjoint union of both constructs. The translation can be extended to λ-terms which execute imperative C++-code with side effects. Assume

```
class Student{
public: int student_number;
        Student(int x){student_number = x;};}
```

Assume some standard (lazy or non-lazy – in fact the full range of C++ implementations of lists is possible) implementation `list<X>` of lists of type `X`. Then it is easy to define the map functional

```
template<class X,class Y> list<Y>*
                            map (X->Y f, list<X>* l);
```

which takes a function and a list and returns the result of applying each element to this list.

Now, if we extend the above language by allowing λ-terms to have arbitrary sequences of expressions, we can define

```
S->() print_Student_Number
  = \(Student s).() {cout >> Student.student >> endl;
                     return ();};
```

If we map this function to an element of `list <Student*>*`, then it will print out all the students, without any need for using an explicit loop. Similarly we can define functions, which update student numbers in a list of students, etc, using functional style programming in C++.

Having λ-terms with side effects seems to be an intereresting feature in programming and is not covered by usual implementations of λ-terms in C++, since those implementations require the body of a λ-term to be an expression – a sequence of statements is not allowed.

6 Conclusion

In this paper we showed how to introduce functional concepts into C++ in a provably correct way. The modelling and the correctness proof used monadic concepts as well as logical relations. We also showed how to integrate lazy evaluation and infinite structures into C++ and gave examples indicating how to model higher-order functions with side effects. This work lends itself to a number of extensions, for example, the integration of recursive higher-order functions, polymorphic and dependent type systems as well as the combination of larger parts of C++ with the λ-calculus. The accurate description of these extensions will require more sophisticated, e.g. domain-theoretic constructions and a more systematic mathematical modelling of C++. We believe that if our approach is extended to cover full C++, one obtains a language in which the worlds of functional and object-oriented programming are merged, and that we will see many examples, where the combination of both language concepts will result in interesting new programming techniques.

References

[ASS85] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and interpretation of computer programs.* MIT Press, 1985.

[JT93] A. Jung and J. Tiuryn. A new characterization of lambda definability. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 245–257. Springer Verlag, 1993.

[Kel97] R. M. Keller. The Polya C++ Library. Version 2.0. Available via `http://www.cs.hmc.edu/ keller/Polya/`, 1997.

[Kis98] O. Kiselyov. Functional style in C++: Closures, late binding, and lambda abstractions. In *ICFP '98: Proceedings of the third ACM SIGPLAN International conference on Functional programming*, page 337, New York, NY, USA, 1998. ACM Press.

[KL05] O. Kiselyov and R. Lämmel. Haskell's overlooked object system. Draft. Submitted for journal publication. Online since 30 Sep. 2004. Full version released 10 September 2005, 2005.

[Läu95] K. Läufer. A framework for higher-order functions in C++. In *COOTS*, 1995.

[Mog91] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(3):55–2, 1991.

[MS00] B. McNamara and Y. Smaragdakis. Functional programming in C++. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 118–129, New York, NY, USA, 2000. ACM Press.

[Plo77] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[Plo80] G. D. Plotkin. Lambda definability in the full type hierarchy. In R. Hindley and J. Seldin, editors, *To H.B. Curry: Essays in Combinatory Logic, lambda calculus and Formalisms*, pages 363 – 373. Academic Press, 1980.

[Pol81] W. Polak. Program verification based on denotation semantics. In *POPL '81: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 149–158, New York, NY, USA, 1981. ACM Press.

[Sch00] S. Schupp. Lazy lists in C++. *SIGPLAN Not.*, 35(6):47–54, 2000.

[Set03] A. Setzer. Java as a functional programming language. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs: International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002. Selected Papers.*, pages 279 – 298. LNCS 2646, 2003.

[SS71] D. Scott and C. Strachey. Toward a mathematical semantics for computer languages. Programming Research Group Technical Monograph PRG-6, Oxford Univ. Computing Lab., 1971.

[Sta85] R. Statman. Logical relations and the typed lambda-calculus. *Inf. and Control*, 65:85 – 97, 1985.

[Sto77] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, MA, USA, 1977.

[Str] J. Striegnitz. FACT! – the functional side of C++.
http://www.fz-juelich.de/zam/FACT.

[Tai67] W. W. Tait. Intensional interpretation of functionals of finite type. *J. Symbolic Logic*, 32:198 - 212, 1967.

[Win93] G. Winskel. *The formal semantics of programming languages: an introduction.* MIT Press, Cambridge, MA, USA, 1993.

Bibliography

[Ab06] R. H. Ab.Rauf. Integrating Functional Programming into C++:Implementation and Verfication. *In Arnold Beckmann, Ulrich Berger, Benedikt Löwe*, John V. Tucker (Eds):Logical Approaches to Computational Barriers.Second Conference on Computability in Europe, CIE 2006. Swansea, UK. University of Wales Swansea Report Series, Report # CSR 7-2006, 2006

[ABS08] R. H. Ab. Rauf, Ulrich Berger, Anton Setzer. A Provably Correct Translation of the Lambda-Calculus into a Mathematical Model of C++. *To appear in Jour. Theory Computing System*, 2008.

[ABS06a] R.H. Ab. Rauf, U. Berger, A. Setzer. Functional Concepts in C++. *In:Conference Proceedings of TFP 2006*, 2006

[ABS06b] R. H. Ab. Rauf, U. Berger, A. Setzer. Functional Concepts in C++.. *In: Henrik Nilson (Ed.): Trends in Functional Programming.*, Volume 7, Series Trends in Functional Programming, Intellect, Bristol and Chicago,pg. 163-179, 2007.

[AG05] D. Abraham and A. Gurtovry. *C++ Template Metaprogramming: Concepts, Tools and Techniques from Boost and Beyond.* Addison Wesley, 2005.

[AIW99] B. Pierce A. Igarashi and P. Wadler. Fetherweight Java: A Minimal Core Calculus for Java and GJ. *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming System, Languages & Applications (OOPSLA '99)*, 34(10):132–146, 1999.

[All87] L. Allison. *A Practical Introduction to Denotational Semantics.* Cambridge Univerity Press, 1987.

[And72] B. Anderson. Documentation for Lib Pico-Planner. *School Of AI, Edinburgh University*, 1972.

[AU01] L. Ammeraal and H. V. Utrecht. *C++ For Programming 3rdEdition.* John Wiley & Sons Ltd., 2001.

[Aus99] M. H. Austern. *Generic Programming and the STL.* Addison Wesley., 1999.

[Bar84] Barendegt, H. Pieter. *The Lambda Calculus:2nd Edition.* nh, 1984.

[Bau72] B. Baumgart. Micro Planner Alternate Ref. Manual. *Stanford AI Lab*, 1972.

[BG96] T.J. Bergin and R.G. Gibson. *History Of Programming Languages II.* Addison Wesley, Reading, MA, 1996.

[Bo02] C++ Boost Community. http://www.boost.org, 2002.

[Bra04] G. Bracha. *Generics in Programming Languages.* Addison Wesley,2004.

[BW88] R. Bird and P. Wadler. *Introduction To Functional Programming.* Prentice Hall International, 1988.

[CM98] G. Cossineau and M. Mauny. *The Functional Approach to Programming.* Cambridge University Press, 1998.

[Cox86] B. Cox. *Object Oriented Programming : An Evolutionary Approach.* Addison Wesley, Reading, MA, 1986.

[Dav73] J. Davies. Popler 1.6 Ref. Manual. *TPU Report, University of Ediburgh*, 1973.

[DD01] H. M. Dietel and P.J. Dietel. *How to Program: Introducting Object-oriented Design with UML.* Prentice Hall, 2001.

[Eck00] B. Eckel. *Thinking in C++ 2nd Edition.* Prentice Hall, 2000.

[Eti94] J. Etinger. *Programming In C++.* McMillian Press Ltd., 1994.

[FA00] The Fact! Library Home Page. http://www.fz-juelich.de/zam/FACT, 2000.

[FH88] A. J. Field and P. G. Harrison. *Functional Programming.* Addison Wesley, 1988.

[GJ98] C. Ghezzi and M. Jazayeri. *Programming Language Concepts.* John Wiley and Sons, 1998.

[Hal02] T. Hallock. Funk: A Framework for Functional Programming Style in C++. thomashallock.com/template_metaprogramming.pdf, 2002.

[Har97] J. Harrison. Introduction to Functional Programming. Cambridge University, 1997.

[Hew06] C. Hewitt. The Repeated Demise of Logic Programming and Why It Will be Reincarnated:What Went Wrong and Why. *Lessons from AI Research & Applications, Technical Report SS-06-08,AAI Press*, March 2006.

[Hig73] B. Higman. *A Comparative Study Of Programming Languages.* MacDonald:London and American Elsevier Inc : New York, 1973.

[HM07] J. Heering and M. Mernik. *Domain-Specific Languages in Perspective* Report SEN-E0702, September 2007.

[HS02] S. P. Harbison and G. C. Steele. *C: A Reference Manual 15th Edition.* Paperback, 2002.

[Hud89] P. Hudak. Conception, Evolution, and Application of Functional Programming lamguages. *ACM Computing Surveys*, 21(3), September 1989.

[Hug89] J. Hughes. *Why Functional Programming Language Matters.* Comput. J.32(2):98-107, 1989.

[Hut06] G. Hutton. *Programming in Haskell.* Cambridge University Press, 2006.

[ISO96] EBNF ISO/IEC. Information Technology-Syntactic Metalanguage - Extended BNF. EBNF ISO/IEC 14977:1996(E), 1996.

[JGLN02] S. Lasota J. Goubalt-Larrecq and D. Nowak. Logical Relations for Monadic Types. *Proceedings of the 16th International Workshop on Computer Science Logic(CSL '02), Lecture Notes in Computer Science*, 2471:553–568, 2002.

[JP00] J. Jarvi and G. Powell. The Lambda Library: Lambda Abstraction in C++. *Technical Report 378, Turku Centre For Computer Science*, November 2000.

[JT93] A. Jung and J. Tiuryn. A New Characterization of Lambda Definability. *Typed Lambda Calculi and Applications, Lecture Notes Computer Science*, pages 245–257, 1993.

[Kis98] O. Kiselyov. Functional Style in C++: Closures, Late Binding and Lambda Abstractions. *ICFP:Proceedings of the third ACM SIGPLAN International Conference on Functional Programming*, page 337, 1998.

[Kow88] R. Kowalski. The Early Years of Logic Programming. *ACM*, 1988.

[LA96] LinAlg: A Numerical Maths Library. http://pobox.com/ oleg/fttp/LinAlg.README.txt, 1996.

[Laf94] M. Beaudon Lafon. *Object - Oriented Languages - Basic Principles and Programming Techniques*. Chapman & Hall, 1994.

[Lau95] K. Laufer. A Framework for Higher Order Functions in C++. *COOTS*, 1995.

[Lig73] J. Lightwill. Artificial Intelligence: A General Survey of AI. *A paper Symposium UK Science reserach Council*, 1973.

[May87] H. G. Mayer. *Programming Languages*. MacMillian Publishing Company, 1987.

[Mog91] E. Moggi. Notions of Computation and Monads. *Information and Computation*, 93(1):55–92, 1991.

[MS00] B. McNamara and Y. Smaragdakis. Functional Programming in C++. *International Conference on Functional Programming (ICFP2000)*, 2000.

[MS01] B. McNamara and Y. Smaragdakis. Functional Programming in C++ Using FC++ Library. *SIGPLAN Notices*, April 2001.

[MS03] B. McNamara and Y. Smaragdakis. Syntax Sugar for FC++: Lambda, Infix, Monads and more. *DPCOOL '03*, 2003.

[MV97] S.Miller and T. Vitale The Miranda Programming Language. http://www.engin.umd.umich.edu/CIS/course.des/CIS400/miranda/miranda.html, 1997.

[Pau00] L.C. Paulson. Foundations of Functional Programming. Computer Science Tripos Part IB, Easter Term, University of Cambridge, 2000.

[Pit91] A. M. Pitts. Evaluation Logic. *Workshop In Computing, Springer*, pages 162–189, 1991.

[Plo77] G. D. Plotkin. LCF Considered as a Programming Language. *Theoretical Computer Science*, 5:223–255, 1977.

[Plo80] G. D. Plotkin. Lambda Definability in Full Type Hierarchy. *To H.B. Curry; Essays on Combinatoric Logic, Lambda Calculus and Formalism*, pages 363–373, 1980.

[Ree04] T. Reenskaug. Empowering People with BabyUML. *A Sixth Generation Programming Language, ECOOP2004*, 2004.

[Ree07] T. Reenskaug. Programming with Roles and Classes:the BabyUML Approach, a Chapter in Computer Software Engineering Research. *Nova Publishers, Happauge NY*, 2007.

[RS06] G. D. Reis and B. Stroustrup. Specifying C++ Concepts. *Proceedings of the 2006 POPL Conference, ACM SIGPLAN Notices Archive*, 41(1): 295 – 308, January, 2006.

[SA05] H. Sutter and A. Alexandrescui. *C++ Coding Standards: 101 Rules, Guidelines and Best Practices*. Addison Wesley, 2005.

[Sch00] S. Schupp. Lazy List in C++. *SIGPLAN Not.*, 35(6):47 – 54, 2000.

[SL00] J. G. Siek and A. Lumsdaine. C++ Concept Checking. *Dr. Dobb's Journal*, June, 2001.

[SL01] J. G. Siek and A. Lumsdaine. Concept Checking: Binding Parametric Polymorphism in C++. *First Workshop on C++ Template Programming, Germany* , 2000.

[SS71] D. Scott and C. Strachey. C: Mathematical Semantics for Computer Language. *Tech. Monograph PRG-6, Programming Research Group*, 1971.

[SS00] J. Striegnitz and S. A. Smith. An Expression Template Aware Lambda Function. *First Workshop on C++ Template Programming*, 2000.

[Sta85] R. Statman. Logical Relation and the Typed Lambda Calculus. *Information and Control*, 65:85–97, 1985.

[STL00] The SGI Standard Template Library. http://www.sgi.com/tech/stl, 2000.

[Str95] B. Stroustrup. Why C++ is Not Just an Object-oriented Programming Language. *OOPSLA '95*, 1995.

[Str03] B. Stroustrup. Concept Checking – A More Abstract Complement to Type Checking. Committee paper N1510-03-0093 Paper for the C++ Committee, October 22, 2003. http://www.reseaech.att.comm/ bs/n1510-conceptchecking.pdf.

[Tai67] W. Tait. International Intrepretation of Functional of Finite Type I. *Journal of Symbolic Logic*, 32(2):198–212, 1967.

[Tan04] G. Tan A Brief History of Functional Programming http://www.cs.bc.edu/ gtan/historyOfFP.html, 2004.

[Vel95] T. L. Veldhuizen. Expression Templates. *C++ Report*, 1995.

[Wat90] D. A. Watt. *Programming Language Concepts and Paradigms.* Prentice Hall International, 1990.

[Win93] G. Winskel. *The Formal Semantics of Programming Languages: an Introduction.* Massachussets Institute of Technology, 1993.

[WS03] B. K. Williams and S.T. Sawyer. *Using Information Technology (A Parctical Introduction to Computers and Communications), Fifth Edition.* McGraw Hill, 2003.