Functional Embedded Telephony in Camelot

Hua Yang

Master of Science

School of Informatics

University of Edinburgh

2005

Abstract

This project would first develop an on-line game application in the Camelot programming language which would be compiled to run on an emulator for a Java-enabled mobile phone.

Java-enabled mobile telephones allow users to develop and download their own code onto the handset. Programming an embedded system such as a telephone is very different from programming a general-purpose workstation. Embedded system developers face problems, such as conserving the battery life, which have no analogue in traditional desktop computing.

One of the most significant costs of battery energy in embedded devices is memory usage. For this reason, developers benefit from using Camelot, an OCaml-like functional programming language, which can give precise guarantees of resource consumption inferred directly from the application code. Camelot uses the Hofmann/Jost type system to provide quantitative guarantees about the consumption of resources such as memory. And lfd_infer is the implementation for the static prediction of heap space usage, which relates to the Hofmann/Jost type system. This project will use this tool to get the resource consumption of the game.

On the other hand, although Camelot has been used in a wide range, most implementations of it so far are small and aim for particular problems. The implementation of this project faces a variety of problems. Thus, it is a good opportunity to test and improve the Camelot compiler, so this is the other task of this project.

Acknowledgements

First, I would like to thank my supervisor Professor Don Sannella for his continued guidance and many fruitful discussions throughout the project, and Dr Kenneth MacKenzie, this work would not have been possible without his insightful help. I would also like to thank Dr David Aspinall for his helpful advice and encouragement.

Thanks also to my family and friends for providing support and encouragement throughout my entire MSc.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Hua Yang*)

Table of Contents

Chapter 1 Introduction 1

1.1 Embedded System Development 1

1.1.1 Memory Overflow Concern 1

1.1.2 Related Work on Memory-Overflow Problem 2

1.2 Proof-Carrying Code 3

1.3 MRG Project 6

1.3.1 Architecture of MRG 6

1.3.2 Components of MRG 7

1.4 Description of the Project 10

1.4.1 Motivation and Description of the Project 10

1.4.2 Principle Goal of the Project 11

1.4.3 Preparation for the Project 11

1.5 Structure of the Dissertation 12

Chapter 2 Camelot and the Case Study 13

2.1 OCaml: Objective Caml 13

2.1.1 The Core Language 13

2.1.2 Object-Oriented Features of OCaml 14

2.2 Camelot 15

2.2.1 The Core Language: Comparison between Camelot and OCaml 15

2.2.2 OCamelot: Object-Oriented Camelot 18

2.3.3 Extended With Concurrency 21

2.3 Case Study: Functional Embedded Telephony in Camelot 23

2.3.1 Game Storyline 23

2.3.2 Facilities 24

2.3.3 Implementation in Detail 24

Chapter 3 Inference of Heap-Space Bounds 34

3.1 Linear Programming 34
3.2 The Program Logic 37
3.2.1 The Syntax and Operational Semantics of Grail 38
3.2.2 The Program Logic 42
3.3 Inference of Heap-Space Bounds 45
3.3.1 Introduction of lfd_infer 45
3.3.2 Further Discussion About Heap Space Usage 47
3.4 Practical Work on the Inference of Heap-Space Bounds 50
3.4.1 Verifying Correctness of the Inference Result 50
3.4.2 Experiment Problem Report And Analysis 53
3.4.3 Heap Consumption of This Project 59
Chapter 4 Conclusion 66
4.1 Project Summary 66
4.2 Further Work 68
Appendix A ML Family Evolution Tree 70
Appendix B The Syntax of Camelot 71
Appendix C Camelot's Built-in Funtions 73
Appendix D.1 The Map of The Game 74
Appendix D.2 Actions in The Game 75
Appendix D.3 The Categories of Objects 76
Appendix D.4 Self-Defined Datatypes 77
Appendix D.5 E-R Diagram Of the Database 78
Appendix E Pure Camelot Code in the Game 79
Bibliography 87

List of Figures

1.1 Overview of Proof Carrying Code …….. 4

1.2 The Architecture of MRG …… 6

2.1 Insertion Sort ……..……..17

2.2 Example for Match Statement in Camelot……...17

2.3 The Syntax of a Class Definition in Camelot ……..... 20

2.4 Derived Forms for Thread Creation and use in Camelot ……22

2.5 Layered Architecture ……...25

2.6 Using The Diamond Type ……...…29

2.7 the Room Class in the Project ……30

2.8 Definition of Thread **connect** ……31

3.1 Feasible Region for LP Program ……35

3.2 The Syntax of Grail ……38

3.3 The Dynamic Semantics of Grail ……41

3.4 The Program Logic for Grail ……..51

3.5 simple.constraints: All constraints for Inference ……51

3.6 An Unbounded Case in Linear Programming ……58

3.7 Changed Linear Problem ……59

List of Tables

2.1 Basic Features of Camelot's Object System ……...19
2.2 Explanation of items in the Class Definition Syntax …….....21
2.3 Explanation of the Game Architecture ……26

3.1 Explanation of Grail's Syntax ……39
3.2 Representation of Resource's Four Components ……42
3.3 Operations on Resource ……..42
3.4 Solutions for the Heap Usage Problem of Multi-Threads ……..49

Chapter 1 Introduction

1.1 Embedded System Development

Nowadays, computing systems are everywhere. When we talk about them, most of us think of “desktop” computers, for example, PC’s, laptops, mainframes, servers, etc. But there is another far more common type — the embedded system. An embedded system is a combination of computer hardware and software, either fixed in capability or programmable for a specific computational task. Compared with conventional desktop systems, embedded systems consist of fairly standard components, such as processors, memory units, buses, peripherals as well as real-time I/O devices, e.g. sensors and actuators. In addition to these, embedded systems also have their own characteristics:

- Single-functioned, which means each of them execute a single program repeatedly
- Tightly-constrained, including low cost, low power, small, fast, etc.
- Continually react to changes in the system’s environment
- Must compute certain results in real-time without delay

Embedded systems are already used in a broad area, from industrial machines, automobiles, medical equipment, airplanes to toys, vending machines, as well as cell phones.

1.1.1 Memory Overflow Concern

Out-of-memory error is a serious problem in computing, which becomes more critical in the context of the embedded system, because of its limited memory and particular storage organization. [1] points out that desktop systems can reduce the

ill-effects of the out-of-memory problem through their virtual memories in two ways. One is that additional space on hard disks can be provided by virtual memories the time a workload runs out of physical main memory (DRAM), and as a result the workload can continue making progress. The second is related to the hardware-assisted segment-level protection mechanism provided by virtual memory, which ensures that an application with an excessive memory requirement can be terminated by user without crashing the system[1]. Embedded systems, on the other hand, typically have neither hard disks, nor virtual memory support. This means that out-of-memory errors leave the system in greater peril. More specifically, since there is no swap space — the additional space allocated in the hard disk by the virtual memory, the memory-overflow application has no space to grow into, therefore, the system crashes. Furthermore, because of the absence of protection given by the virtual memory, there is a possibility that a segment exceeds the memory bound cannot be detected and hence no pre-crash remedial action can be performed. So for correct execution, programmers need to make accurate compile-time estimates of the maximum memory requirement of each task across all input data sets, and choose a physical memory size larger than the maximum memory requirement of the embedded application.

1.1.2 Related Work on Memory-Overflow Problem

Currently works on the memory-overflow problem have already been carried out, and some helpful solutions were proposed. Next I will give an introduction of two different approaches, which represent two directions of researches on this problem, one on hardware and one on software.

(1) Add a limited form of virtual memory which provides memory protection but not swap-space [1]. This method is mostly implemented in a few high-end embedded systems [2]. Unlike virtual memory for desktop systems which gives programmers the illusion of an unlimited usage of memory, all embedded

systems, regardless with or without virtual memory, are inherently constrained by the amount of available physical memories [3]. In other words, because of their lack of hard disks and hence of swap space, out-of-memory problem has not been thoroughly solved, and even programs running on embedded systems that have memory management hardware and virtual memory, is still possible to cause memory overflow. On the other hand, most commercial embedded processors, such as [4] [5] [6], do not go in for this method. The major reason for that is the cost of the hardware memory management units (MMUs) that provide virtual memory has been considered by processor vendor to be excessive in an embedded environment [7] [1].

(2) Software run-time checks for out-of-memory errors. This method is proposed in [1]. The basic idea consists of two parts. First, out-of-memory errors are detected just before they will happen by using carefully optimized compiler-inserted run-time check code. Such error detection enables the designer to incorporate system-specific remedial action, such as transfer to manual control, shutting down of non-critical tasks, or other actions. Second, grow the stack or heap segment after it is out of memory, into previously un-utilized space such as dead variables and space freed by compressed live variables. This technique can avoid the out-of-memory error if the extra space recovered is enough to complete execution [1]. Surpassing the hardware method, this method achieves space recovery and memory protection with low system overheads.

1.2 Proof-Carrying Code

Proof-Carrying Code (PCC) provides an alternative method for achieving memory safety. The same idea can be applied to other safety properties covering data access policies, resource usage bounds, and data abstraction boundaries [8].

What is Proof-Carrying Code?

Proof-Carrying Code, as pioneered by Necula and Lee [9] [10], is a general framework for verifying the safety properties of machine-language programs [11]. PCC enables a computer system to determine, automatically and with certainty that program code provided by another system is safe to install and execute without interpretation or run-time checking [8]. The key ideas behind PCC involve five concepts: *code producer*, *code consumer*, *proof producer*, *safety policy* and *proof*, which are:

- ✓ The *code consumer* requests code from the *code producer*, specifying a *safety policy*
- ✓ The *code producer* sends code to the *code consumer* together with a *proof* produced by the *proof producer*, which indicates that this code abides by the *safety policy*
- ✓ The *code consumer* checks whether the *proof* is valid for the code and the *safety policy*
- ✓ If the *proof* passes the check, the code then can be executed on the *code consumer* side.

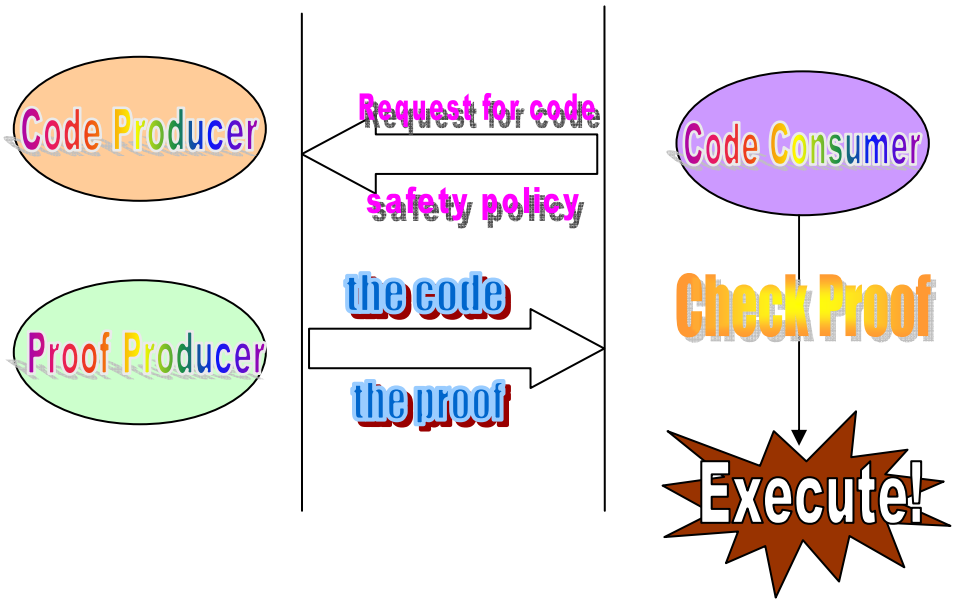


Figure 1.1: Overview of Proof Carrying Code

The principle of Proof-Carrying Code is to construct and verify a mathematical proof about the machine-language program itself, and this guarantees safety — but only if there's no bug in the verification-condition generator, or in the logical axioms, or typing rules, or the proof-checker [11]. As [12] points out, a major advantage of this approach is that it sidesteps the difficult issue of *trust*: there is no need to trust either the code producer, or a centralized certification authority. If some code comes with a proof that it does not violate a certain security property, and the proof can be verified, then it does not matter who wrote this code: the property is guaranteed to hold. But the user does need to trust certain elements of the infrastructure: the code that checks the proof; the soundness of the logical system in which the proof is expressed; and, of course, the correctness of the implementation of the virtual machine that runs the code — however these components are fixed and so can be checked once and for all.

Compared with [1]'s approach to achieving memory safety, Proof-Carrying Code has the potential to free the host-system designer from relying on run-time checking as the sole means of ensuring safety. Authors of [8] further argue that by being limited to memory protection and run-time checking for achieving memory safety, designers of that kind of systems must impose substantial restrictions on the structure and implementation of the entire system. Moreover, Proof-Carrying Code provides greater flexibility for designers of both the host system and then agents, and also allows safety policies to be used that are more abstract and fine-grained

than memory protection [8].

1.3 MRG Project

Mobile Resource Guarantees (MRG) is a European Commission funded project, belonging to the Laboratory for Foundations of Computer Science. MRG applied ideas from Proof-Carrying Code to the problem of resource certification for mobile code [12]. The aim of the project was to develop the infrastructure needed to endow mobile code with independently verifiable certificates describing its resource behaviour, in the form of condensed and formalized mathematical proofs of resource-related properties which are by their very nature self-evident, unforgeable, and independent of trust networks [13].

1.3.1 Architecture of MRG

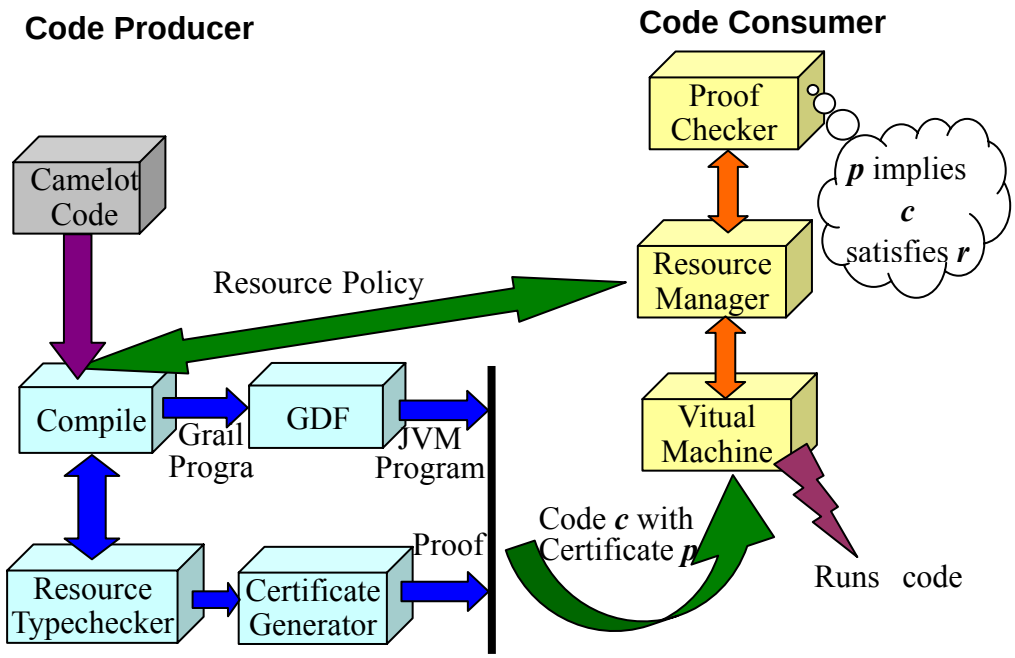


Figure 1.2: The Architecture of MRG

Figure 1.2 shows MRG's PCC-like architecture. Along with it is a novel protocol proposed by MRG. In this protocol, a *Resource Manager* is responsible for negotiating resource policies specified by the code consumer with the code producer, and verifying that the certificate attached to the required code ensures that it will run within the constraints required. A *Proof Checker*, invoked by the *Resource Manager*, is in charge of the practical verification work. If the check succeeds, we have an absolute guarantee that resource bounds are met, so it is not necessary to check for resource violations as the code runs. In detail, the phases in the protocol are [16]:

- **Initiate:** Code producer A wants to send code consumer B a piece of mobile code c
- **Policy:** A and B agree upon some resource policy r that the code must satisfy. The choice of policy may influence some aspect of the compilation (or re-compilation) of the code from a high-level language, in particular how the Resource Type-checker influences the Certificate Generator
- **Certify:** Provided the code meets the policy r, then A sends B the code c together with a certificate p that c abides by r
- **Check:** B checks the validity of the certificate p with respect to the code and the agreed resource policy.
- **Run:** Provided the check was successful, B then runs the code

1.3.2 Components of MRG

1.3.2.1 Programming Languages

The Mobile Resource Guarantees framework provides two language levels: Camelot, the high-level language, and Grail, the low-level language into which Camelot is compiled.

(1) Camelot

Camelot is a first-order OCaml-like resource-safe function language, which was developed as a test bed for different methods of analyzing heap usage [14]. Camelot provides the usual recursive datatypes and recursive functions definitions, using pattern matching, albeit restricted to flat patterns [12]. In order to create a system allowing inter-operation with external Java libraries and to compile an object system to JVML, Camelot further extends its type system to include object-oriented features. With respect to the concurrency problem, a simplified thread model is added in Camelot which abbreviates the use of threads in the language. Still, Camelot has its differences from OCaml, one of which lies in Camelot's memory model. Through a special type, Camelot enables programmers to precise control heap space usage. More information about this type is put in a later section about the space-aware type system. And a detailed introduction to this language is in chapter 2.

(2) Grail

Grail is a small typed language, which represents Java bytecode in a functional form. As a conventional strict first-order functional language, Grail has the following four characteristics: [15]

- call-by-value function invocation
- lexical scoping for variables
- mutually recursive local function declarations
- strict static typing

As a vehicle for Proof-Carrying-Code-like project, Grail is required to [12]:

- be the target for the Camelot compiler;
- serve as a basis for attaching resource assertions;
- be amenable to formal proof about resource usage;
- provide a uniform format for sending and receiving guaranteed code
- be executable.

Grail has two semantics: one functional and one imperative, both of which have direct connections with responsibilities listed above. Grail's compiler is a part of Camelot's compiler, so sometimes for simplification, we can directly get a Java bytecode from a Camelot program using the compiler of Camelot.

1.3.2.2 Space-aware Type System

Camelot, in combination with its space type system, enables to produce JVM bytecode endowed with guaranteed and certified bounds on heap space consumption. MRG uses Hofmann/Aspinall type system, which provides an abstract type denoted <>, as well as enforcing linear typing.

This abstract type, called Diamond, represents regions of heap-allocated memory. The motivation of designing a special diamond type is to allow better control of heap usage. All non-primitive types in a Camelot program are compiled to JVM objects of a single class Diamond, which contains appropriate fields to hold data for a single node of any datatype. Diamond values can be obtained through constructors for datatypes or released through match rules with special annotations. Considering that there will be times that one may not want to re-use a diamond value immediately, a freelist is provided for the storage of unused diamonds.

The type system provides two ways to achieve linear typing. First the Camelot compiler has an option that enforces linear use of all variables, or alternatively, the programmer himself ensures that the datatype he deconstructed using the match rule is not used anymore, because its contents will be overwritten subsequently. Linear typing schemes guarantee single-threadedness and so the soundness of in-place update with respect to a functional semantics [37]. However, this is a restrictive discipline in practice and rules out many sound programs. So in [37] an improved type system, that distinguishes between modifying and read-only access to a data structure and in particular allows multiple read-only accesses, is proposed [12].

1.3.2.3 Program Logic for Generating and Checking Proofs

The Camelot programming language is supported not only by a strong, expressive type system, but also by a program logic which supports reasoning about the time and space usage of programs in the language. Actually speaking, the program logic operates on programs written in Grail which is the target of Comelot's compiler. Serving as the target logic of a certifying compiler, the program logic exploits Grail's dual nature of combing a functional interpretation with object-oriented features and a cost model for the JVM [25]. This program logic, together with the resource-aware operational semantics of Grail, has been formalized in the theorem prover Isabelle/HOL. Certifications in the MRG framework contain a claim of resource usage together with a proof of the claim. And this proof is just expressed in this program logic. The logic exists not only on the code producer side, where it is used by the certifying compiler to generate certificates, but also on the code consumer side, where it is used by the proof checker to check whether the certificate attached to the code indicates that the required resource of the program will not violate the resource policy the consumer claimed, and whether this certificate is logically correct. A detailed introduction to this will be given in Chapter 3.

1.4 Description of the Project

1.4.1 Motivation and Description of the Project

In these years the worldwide market of mobile communication is growing at a rapid pace and has overtaken wired phone communications. We have heard for a while that the mobile industry will explode with millions of end-user playing games and interacting on billions of handsets. In particular, Screen Digest (June 2004) estimated that the total market for online games will double between 2004 and 2007, the Massive Multi Player Online Game (MMOG) market continues to grow, and

Europe, a relatively untapped MMOG market to date, will become the largest growth opportunity when the North American market approaches saturation.

On the other hand, as a kind of embedded systems, programming a mobile device is very different from programming a general-purpose workstation. Developers face problems, such as conserving the battery life, which have no analogue in traditional desktop computing. One of the most significant costs of battery energy in mobile devices is memory usage. For this reason, developers benefit from using programming languages which give precise guarantees of resource consumption that can be inferred directly from the application code. Camelot is just this sort of programming language, which in combination with its space-aware type system provides quantitative guarantees about the consumption of resources such as memory usage.

1.4.2 Principle Goal of the Project

The principle goal of this case study project is:

- developing an on-line game application in the Camelot programming language which would be compiled to run on an emulator for a Java-enabled mobile phone.
- most implementations of Camelot so far are small and aim for particular problems, and the implementation of this project will face a variety of problems. Thus, it is a good opportunity to test and improve the Camelot compiler
- using an implementation of a static inference on heap space usage to get quantitative results about the consumption of the resource used by our game.

1.4.3 Preparation for the Project

Before carrying out this project, some preparations have been made:

(1) Proof-Carrying Code Framework

The comprehension of this technology has been stated in previous sections.

(2) MRG Project

Besides for the components introduced in section 1.3.2, MRG still made some other achievement. Understanding some work of this project is required for this project.

(3) DEGAS

DEGAS, Design Environments for Global ApplicationS, mainly concerns **specification in UML** and the **qualitative and quantitative analysis** of global applications. In [17], DEGAS provides a case study of m-MMPORG (mobile Massive Multi Player Online Role Game), the one to be implemented in this project, in the form of a high-level UML design for a room-based game to be played over mobile telephones.

(4) Current technologies on mobile devices development

This provided me ideas for later practical implementation work.

1.5 Structure of the Dissertation

The organization of this dissertation in chapters occurs as follows:

Chapter 2 This chapter first gives a literature review about OCaml and Camelot, and then introduces the implementation of this game in detail.

Chapter 3 Work on inference of heap space usage is presented in this part. But before that, linear programming and a program logic are introduced first.

Chapter 4 This chapter concludes the work we have done for this project, followed by an expectation of further work.

Chapter 2 Camelot and the Case Study

2.1 OCaml: Objective Caml

OCaml, Objective Caml, is an advanced programming language which is a member of the ML family. OCaml shares the functional and imperative features of ML, but adds object-oriented concepts and has minor syntax differences [18]. (Appendix A shows a relationship tree which indicates how ML was influenced by and has influenced other languages, together with the development of ML. You might get a general knowledge about the ML family there.)

2.1.1 The Core Language

2.1.1.1 The Basics of OCaml

OCaml offers basic built-in types: bool, int, float, char, string, as well as predefined data structures: tuples, lists, and arrays. In addition to these, OCaml allows user-defined data structures, like records and variants. In OCaml, everything is an expression so everything returns a value. Variables of OCaml are immutable, that is, once they are bound to a value, they cannot be changed except by a new, fresh binding. While this seems very limiting to a programmer who is accustomed to other programming styles, it enforces safe programming by not allowing side effects.

Functions in OCaml are values that are bound to names and treated as first-class values. Strictly speaking, no function in OCaml takes more than one argument. Multiple argument functions are actually curried functions. OCaml enables conditional computation which is performed with the traditional if-then-else construct.

OCaml is not a pure function language. It has looping constructs like while and for

loops, as well as mutable data structures such as arrays [19]. Like ML, OCaml also provides exception signaling and handling mechanism.

2.1.1.2 The Module System

The module system is one of the important characteristics of OCaml. The primary motivation for it is to centralize related definitions, e.g. definitions of a data type and associated operations on that type, as well as enforce a consistent naming scheme for these definitions. This can efficiently avoid running out of names or accidentally confusing names [19]. ***Structure***s, ***Signature***s and ***Functor***s form the basis of this system.

2.1.2 Object-Oriented Features of OCaml

OCaml supports class definition. Like conventional object-oriented programming languages, OCaml provides concepts like private methods, abstract methods (but here called virtual methods), inheritance, class coercions, friend class, as well as *new* for creating objects. Features of OCaml which distinguish it from OO imperative languages are:

- OCaml provides a direct way to create an object without going through a class. The syntax is exactly the same as for class expressions, but the result is a single object rather than a class. Unlike classes, which cannot be defined inside an expression, immediate objects can appear anywhere, using variables from their environment [19].
- In the body of a class, polymorphic methods are allowed, but with a limitation, namely, a polymorphic method can only be called if its type is known at the call site. Otherwise, the method will be assumed to be monomorphic, and given an incompatible type [19].
- OCaml provides the library function Oo.copy for cloning objects. Usually, the instance variables have been copied but their contents are shared. Assigning a

new value to an instance variable of the copy will not influence the original's, and vice versa. However, in the case that the instance variable is a reference cell, assignments will influence both the original and the copy.

- OCaml allows recursive classes definitions where recursive classes can be used to define objects whose types are mutually recursive.
- OCaml provides a novel method, the binary method, which takes an argument of the same type as self.

2.2 Camelot

Camelot serves as the high-level programming language in MRG framework. Owing to its resource-safe character, it is developed as the test bed for different methods of analyzing heap usage.

2.2.1 The Core Language: Comparison between Camelot and OCaml

The core of Camelot is a standard polymorphic ML-like functional language whose syntax is based upon that of OCaml [20]. In section 2.2, we have introduced the core language of OCaml, so here we only focus on differences between them (Appendix C shows the complete syntax of Camelot).

(1) In OCaml, there is actually no way to modify a list in-place once it is built. For example, the sort function in figure 2.1(a) does not modify its input list, instead it builds and returns a new list containing the same elements as the input list, in ascending order. Most OCaml data structures like list are immutable, except for a few, like arrays which can be modified in-place at any time.

However, Camelot allows in-place modification through the diamond type and the freelist. The diamond type in Camelot is denoted by <> whose values represent blocks of heap-allocated memory. Camelot allows explicit manipulation of diamond

objects, and that is achieved by equipping constructors and match rules with special annotations referring to diamond values [14]. See the insertion sort example in figure 2.1(c). The annotation "@d" on the first occurrence tells the compiler that the space used by the list cell is to be made available for re-use via the diamond value d [22]. And the second annotation on the second occurrence points out that the new list cell should be constructed in the diamond object referred to by d. Considering that sometimes we don't want to re-use a diamond value immediately, Camelot provides a freelist for the storage of unused diamonds. The diamond annotated by "@_" will be placed on the freelist for later use. Figure 2.1(b) shows the usage of it. The related topic about static inference of heap space usage will be discussed in chapter 3.

```
let rec sort l =
match l with [ ]→[ ]
           | hd :: tl → insert hd (sort tl)

and insert a l =
      match l with [ ]→[a]
                 | hd :: tl → if a ⇐ hd then a :: l
                              else hd :: insert a tl
;;
```

Figure 2.1(a): insertion sort in ocaml

```
let insert a l =
      match l with [ ]→ a :: []
                 | hd :: tl @_ →
                     if a <= hd then a :: (hd :: tl)
                     else hd :: (insert a tl)

let sort l = match l with [ ]→[ ]
                        | hd :: tl @_ → insert hd (sort tl)
```

Figure 2.1(b): In-place Insertion Sort in Camelot

```
let insert a l =
        match l with [ ] → a :: []
                | (hd :: tl) @d →
                        if a <= hd then a :: (hd :: tl)@d
                        else (hd :: (insert a tl))@d

let sort l = match l with [ ] → [ ]
                        | (hd :: tl)@d → (insert hd (sort tl))@d
```

Figure 2.1(c): Another In-place Insertion Sort in Camelot

Figure 2.1: Insertion Sort

(2) OCaml supports higher-order functions but Camelot does not. In Camelot, any invocation of a function must supply exactly the same number of arguments as are specified in the definition of the function.

(3) Iterative constructs are not available in Camelot. Camelot only supports recursive definitions or invocations.

(4) Camelot does not support exception signaling and handling, as well as the module notion.

(5) Camelot also uses the match statement for datatype deconstruction, but the form of it is much more restricted than the one of OCaml [21]. This means that there must be exactly one rule for each constructor in the associated datatype, and each rule binds the values contained in the constructor to variables, or discards them by using the pseudo-variable " _ " [21]. An example of match statement usage in Camelot is given in figure 2.2.

```
type int list = Nil | Cons int * int list

let copylist (l1 : int list) (l2 : int list) =
        match l1 with Nil → l2
                | Cons (h, t) → let l2 = Cons (h, l2)
                                in copylist t l2
```

Figure 2.2: Example for Match Statement in Camelot

(6) Camelot also allows self-defined datatypes but with a requirement that datatype constructors must begin with an upper-case letter.

(7) *string* type in Camelot is immutable, different from the one in OCaml, where the ith element of a *string* type variable *s* can be obtained through *s.[i]*.

(8) In OCaml, we use *arr.(i)* to retrieve the ith element of an array *arr*, but in Camelot, we use the built-in function for array, *get arr i*. Besides for this, Camelot still has several other functions on arrays, like empty: int -> α -> α array, set: α array -> int -> unit and arraylength: α array -> int.

(9) OCaml provides an append operator @ on lists, but Camelot reserves this symbol for use with rules of the diamond type, which is a special type for explicit resource allocation.

(10) Camelot does not have the = = operator in OCaml. = in Camelot can be applied to strings and other objects. However, it is interpreted as equality of references and hence will usually fail to give the expected result; for strings, we should use the function same_string [22].

(11) Functions in Camelot are defined using the keyword *let*, rather than *fun* or *function*, which are used in OCaml.

(12) In addition to the primitive types supported by OCaml, Camelot still provides ***unit*** type, which has a single value ( ). Camelot has its own built-in functions, which are shown in Appendix C.

2.2.2 OCamelot: Object-Oriented Camelot

The original motivation for adding object-oriented features to Camelot is the need to create a system allowing inter-operation with external Java libraries, and to compile an object system to JVML [21]. Rather than creating a fundamentally different system, the object system for Camelot is drawn from the object system of the JVM. From another point of view, the power of Camelot's object system might be considered as a subset of OCaml's. Table 2.1 shows the basic features of

OCamelot.

Feature	*Description*	*Example*
Static Method Calls	Static methods and functions are conceptually equal, ignoring the use of classes for encapsulation.	java.lang.Double.parseDouble a
Static Field Access	Only accesses to constant static fields are supported	java.lang.Double.MAX_VALUE
Object creation	Use the *new* operator in the curried form	new java.lang.Double 3.1415926
Instance field access	Retrieve the value of an instance: Update the value of an instance:	object#field object#field ← value
Method invocation	introduce from OCaml's syntax, using a curried form	object#method para1 para2
Null values	Considering existence of null object, a function isnull is added to test whether the curried expression is a null value	isnull expr Test if the expression expr is a null value
Casts & typecase	cast objects up to superclasses: cast objects down to subclasses:	obj :> superclass match obj with o :> subclass1 → o.a() | o :> subclass2 → o.b() | _ → obj.c()

Table 2.1: Basic Features of Camelot's Object System

In addition, OCamelot also supports self-defined classes. This facility can be used to implement callbacks, such as in the Swing GUI system which requires us to write stateful adaptor classes [29], or to invoke Camelot codes in the context of Java, for

example to create a resource-certified library for use in a Java program [29]. Figure 2.3 shows the syntax of a class declaration, followed by table 2.2 explaining items in the syntax.

Examples of self-defined classes can be found in the next section which introduces implementation details of this project.

Clearly the presence of mutable objects in object-oriented Camelot provides for in-place update [29]. However the unbounded heap-usage problem solved for datatypes are replicated by allowing arbitrary object creation. Perhaps more seriously, invocating arbitrary Java code in Camelot programs may lead to unlimited heap space usage. Further discussion about this problem will be given in chapter 3.

$$
\begin{aligned}
&\textit{classdefl} ::= \text{class } \textit{cname} = \langle \textit{scname} \text{ with} \rangle \ \textit{body} \text{ end} \\
&\textit{body} ::= \langle \textit{interfaces} \rangle \langle \textit{fields} \rangle \langle \textit{methods} \rangle \\
&\textit{interfaces} ::= \text{implement } \textit{iname} \langle \textit{interfaces} \rangle \\
&\textit{fields} ::= \textit{field} \langle \textit{fields} \rangle \\
&\textit{methods} ::= \textit{method} \langle \textit{methods} \rangle \\
&\textit{field} ::= \text{field } x : \tau \mid \text{field mutable } x : \tau \mid \text{val } x : \tau \\
&\textit{method} ::= \text{maker } (x_1 : \tau_1) \ldots (x_n : \tau_n) \langle : \text{super } x_{i_1} \ldots x_{i_m} \rangle = \textit{exp} \\
&\qquad \mid \text{method } m(x_1 : \tau_1) \ldots (x_n : \tau_n) : \tau = \textit{exp} \\
&\qquad \mid \text{method } m() : \tau = \textit{exp} \\
&\qquad \mid \text{let } m(x_1 : \tau_1) \ldots (x_n : \tau_n) : \tau = \textit{exp} \\
&\qquad \mid \text{let } m() : \tau = \textit{exp}
\end{aligned}
$$

Figure 2.3: The Syntax of a Class Definition in Camelot

item	*explanation*
class *cname*	Define a class called *cname*
<*scname* with>	Inherit from a class called *scname*
implement *iname*	Implement an interface called *iname*
field <mutable> $x : \tau$	Instance field named x of type τ, optionally be declared to be mutable
val $x : \tau$	Static fields named x of type τ
maker $(x_1 : \tau_1) \ldots (x_n : \tau_n)$ <: super $x_{i1} \ldots x_{im}$> = exp	Constructor of the class with arguments x_1 of type τ_1, .. x_n of type τ_n, the superclass maker is optionally executed, and expression exp is executed.
method m $(x_1 : \tau_1) \ldots (x_n : \tau_n) : \tau$ = exp	Instance method named m of type τ with arguments x_1 of type τ_1, .. x_n of type τ_n, expression exp is executed.
method m () : τ = exp	Instance method named m of type τ with xero argument, expression exp is executed.
let m $(x_1 : \tau_1) \ldots (x_n : \tau_n) : \tau$ = exp	Static method named m of type with arguments x_1 of type τ_1, .. x_n of type τ_n, expression exp is executed.
let m () : τ = exp	Static method named m of type τ with xero argument, expression exp is executed.

Table 2.2: Explanation of items in the Class Definition Syntax

2.3.3 Extended With Concurrency

In Camelot, a simplified thread model is especially designed for concurrent programming, which abbreviates the use of threads in the language. These derived forms are implemented by class hoisting, moving a generated class definition to the

top level of the program [20]. Figure 2.4 shows concrete forms:

```
let rec threadname(args) =
    let locals = subexps in threadname(args)
let threadInstance =
    new threadname(actuals) in …

class threadnameHolder(args) = java.lang.Thread
with
    let rec threadname() =
        let locals = subexps in threadname()
    method run() : unit =
        let _ = this#setDaemon(true)
        in threadname()
    end
    let threadInstance#start()  in …
```

Figure 2.4: Derived Forms for Thread Creation and use in Camelot [20]

In order to retain predictability of memory behaviour in Camelot, there are several restrictions:

(1) The **stop** and **suspend** methods from Java's threads API are not allowed. This is because the former will cause objects to be exposed in a damaged state, the latter will freezes threads without releasing resources occupied by them, and as a result both of them can confuse the prediction of memory consumption.

(2) All threads are required to run, namely, all threads are started at the point where they are constructed. This is also required for the sake of predictability.

(3) Each class has one constructor, which implies that overloading is not supported by Camelot. So initial values for all the fields of the class need to be passed when an object is created.

2.3 Case Study: Functional Embedded Telephony in Camelot

This is a case study project to develop an on-line game application in the Camelot programming language. The design of the game is based on [17] where DEGAS provides a case study of m-MMPORG in the form of a high-level UML design for a room-based game to be played over mobile telephones. The motivation of this part is to implement a large practical system in Camelot taking advantage of the features of this language, as well as to test Camelot compiler.

2.3.1 Game Storyline

This Massive Multi-Player Online Role-playing Game (MMPORG) consists of a series of game levels. Each level is composed of a start area, a certain number of standard rooms, and a special room which can hold only one player at a time. The architecture of every level is different from level to level, this means that the number of rooms and the paths they are connected by may change. Appendix D.1 shows the map of this game.

Each player in the game has four associated parameters: health, strength, agility and cash. Players are considered as active entities who can operate on objects, change locations, interact with other players and so on. (Allowed actions are listed in Appendix D.2 with necessary explanation) Objects, including food, weapons, medicine, tools, etc., are divided into categories according to the player's parameter they affect. (Object Categories and corresponding effects are shown in Appendix D.3) Players have to explore rooms to collect as many points as possible to improve their personal parameters. They might encounter obstacle in the special room or be attacked by other players. On this occasion, the winner can have his points increased and obtain an object from the loser. At the same time, the loser may lose some points or an object. If one of the player's parameters reaches zero except for

the health, the player has to quit the current room and be transferred back to the start area of the current level. But if the health reaches zero, the player cannot play anymore. The winner of this game is the one who can pass through all challenges, successively reach the special room in the last level and pass the test there.

2.3.2 Facilities

There are several facilities required to establish as the basis of this project. First, from the point of location, the game can be generally divided into two parts: the server, which is responsible for clients' registration, login, information management and so on; and the client, where actual actions of the game occur. So setting up a web server to carry out jobs on the server side is pre-requisite for this project. Second, since this game is oriented to large numbers of players, so a database is required to store all players' information, as well as to provide query service when necessary. The third is a special package for mobile phone application development, and the corresponding platform for testing. Considering the cost and possibilities, we finally choose Apache Tomcat as our servlet container, postgreSQL as the underlying database management system, Java API provided by J2ME as the special development package, and KVM as the underlying design platform. All of them are free and have large installed bases of users.

2.3.3 Implementation in Detail

2.3.3.1 Architecture of the Game

As a whole, although the developed system involves a server, and clients, it actually follows a distributed architecture. The meaning of this is that some services of the system are provided by the Network. These kind of services are different from traditional client/server ones, because in client/server architecture clients always know where to locate the service (because the services is almost always at the same address), but in a distributed environment like the one implemented in this case

study the service moves from one location to another. And the entity responsible for some service is a set of peers instead of a central entity – the server. Thus in order to locate a service, peers must send messages to the Network to find out the information they need.

This system has a Peer-to-Peer nature, which means that most communication happens between the players' devices, so the figure of the Server is limited only to administrative procedure [17] such as registration, user login, logout, and account management. In the logic of the Game the Server has only little participation, instead the Client takes on most of these jobs. Figure 2.5 shows the structure of the application on the Client side, followed by Table 2.3 explaining the work of each layer.

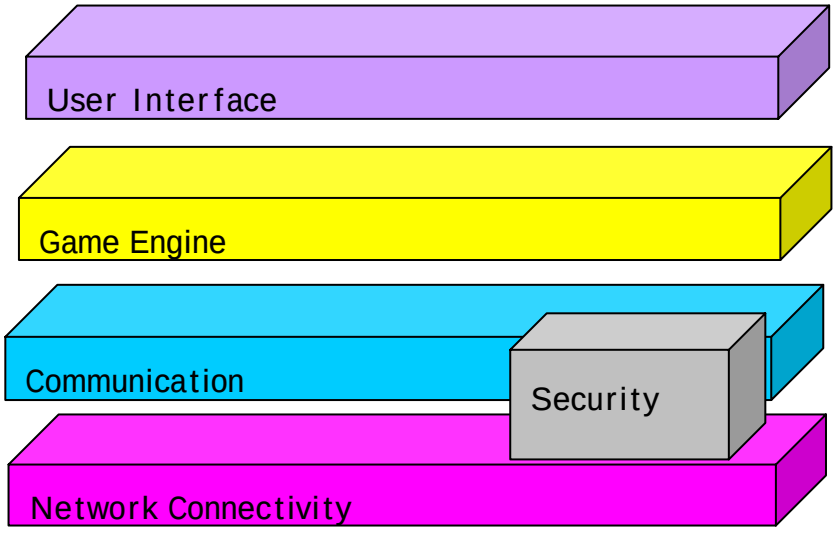


Figure 2.5: Layered Architecture

Layer	*Duty*
User Interface	Enable user to interact with the game
Game Engine	Implement all the game logic, i.e. actions of the player
Communication	implement all the communication activities of the game, i.e. p2p, client/server communication
Network Connectivity	provide essential Network Connectivity allowing the system to relay on an appropriate transfer protocol, i.e. HTTP
* Security Support (need to be improved)	On the 3rd and 4th layers, provide necessary security support for communication applications and connectivity between peers.

Table 2.3: Explanation of the Game Architecture

2.4.3.2 Major Implementation Issues:

In the previous section we have pointed out that this is a distributed system. Issues discussed in this section relate to this feature.

(1) Mobility & Distributed Environment

This system is highly mobile and has a P2P nature. Players inside a Room constitute a virtual community that they can join and leave, and interactions between them can only occur inside the same room. Room object is the most important mobile object of the system. This object contains all the information about the virtual environment in which players are playing and provide necessary services for players during the game. When entering a room of the game, every player receives a copy of the Room object. If the entering player is the first one in a room, it is the server that creates and sends a room copy to the player's device. Other players will be forwarded to players already inside the room and a peer-to-peer procedure will allow the newcomer to receive its copy of the Room object from some other player. These

instances of the Room object must be kept synchronized in order that every player has the same version of the status of the Room he is in. This problem is solved by introducing a second mobile object: a Token. Each room has only one Token, and only the player holding the Token has the right to perform any Environment Action (The definition of Environment Action is given in Appendix D.2). If any other player wants to do some Environment Action, before he can be satisfied, the Token must move to the requesting player's device first.

(2) Synchronization

Environment Actions are defined as Local and Synchronous Actions. It means that this kind of actions is performed directly on the local device of the player doing them, but before this is done, players must synchronize on something. Room is the obvious synchronized point in this game. For example several players may request to take objects or weapons from the same room at the same time. The system must notice this race and grant exclusive access to these resources. And the way to do this is implementing the Token as briefly described above. The token can be thought of as a mobile object moving from one peer to another peer in the same room. Only the player owning the Token can access resources and perform the action he requested. Other players must wait and their requests will be stored in a queue of the Token. After the peer has finished processing the action, the new status of the room needs to be broadcasted to all the other peers in the room for the sake of keeping the consistency of the room. Those peers receiving the information must update their clone of the room object. After this procedure has succeeded the Token can move somewhere else. On the other hand, the request of the other peer will be enqueued, and after the action is finished the request will be dequeued.

In addition to these two issues, the design of the database is also a concern of this project. Appendix E.7 shows the E-R diagram of this project.

2.4.3.3 Programming in Camelot

This case study takes full advantage of the facilities provided by Camelot, i.e.

diamond type for heap-space allocation, self-defined classes, threads for concurrency, and so on. Therefore, it's a good opportunity to help test and the Camelot compiler.

(1) Using the Diamond Type

Diamond type in Camelot enables in-place operations and precise control of heap usage. A freelist for the storage of unused diamonds is provided for the case where one may not always want to reuse a diamond value immediately. Since a safe and static automation of the decision about whether a pattern match is destructive or not is not available, it is our programmers' responsibility to ensure that the datatype we are going to deconstruct will not be used anymore because its heap space will be reclaimed and its contents will be overwritten subsequently. We have mentioned in the previous section that there are two kinds of destructive match patterns in Camelot, one uses "@d" annotation and the other uses "@_". For the inference implementation of the static prediction of heap space usage (introduced in the next section), both of them are treated as @_, as this will not affect the inference in any way. So it is not a big matter whether to use @d or @_, but, it is a matter of distinguishing whether to use a destructive pattern match or a read-only pattern match. Following codes are the examples of their usage in this project.

```
type card = !NullCard | Card of int* string * string * int* int* int* string
let attAbility c =
    match c with Weapon (_, _, _, attak, _, _, _) → attak
               | NullWeapon → 0
```

Figure 2.6(a): A Method in the Project Using Read-Only Match

```
let concatList (l1 : cardpairlist) (l2 : cardpairlist) =
  match l1 with Nil6 → ()
              | Cons6(a, b, t)@ d →
                    concatList t (Cons6(a, b, l2)@ d)
```

Figure 2.6(b): A Method in the Project

Using Deconstructive Pattern Match with "@d"

```
let clearStringIntList (l : stringintlist) =
        match l with Nil3 → ()
                   | Cons3 (a,b,t)@_ → clearStringIntList t
```

Figure 2.6(c): A Method in the Project Using Deconstructive Pattern Match with "@_"

Figure 2.6: Using The Diamond Type

(2) Defining Our Own Classes

Section 2.4.3.1 describes the layer architecture of this game. From the point of implementation view, each layer corresponds to a Camelot class. So in the program, we have three major classes: ***roomgame*** class, realizing the Game Engine layer; ***userinterface***, realizing the User Interface; ***connectionManager***, realizing the Connection Manager layer. Thread connect assumes jobs like the Network Connectivity layer, but we put its introduction in the next section. Other than this, ***plr*** class, room class (actually it is a Thread), ***messageSender*** (Thread), ***messageReceiver*** (Thread) are assistance classes. Rather than implementing the object element and the weapon element of the game as classes, we design special datatypes for them in order to enable the inference experiment conducted in the next chapter. Concrete definitions of object type, weapon type, and related datatypes are given in Appendix D.4. Figure 2.7 shows part of the declaration of the ***roomgame*** class to give a straightforward example of our usage of Camelot's OO features.

```
class roomgame = javax.microedition.midlet.MIDlet with
    field connM : connectionManager
    field ui : userInterface
    field mrecv : messageReceiver
    ⋮
    (* Constructor of the class *)
    maker() =
          let _ = ui ← (new userInterface this)
        in let _ = connM ← (new connectionManager this)
        in ()

    (* Signals the MIDlet that it has entered the Active state *)
    method startApp () : unit =
          let _ = this#ui#startMenu ()
        in let _ = this#connM#openMessageListener ()
        in ()
    ⋮
end
```

Figure 2.7: the ***Room*** Class in the Project

(3) Concurrent Programming

Thread is a frequently used concept in this project. As we can see, **connect**, which is responsible for the communication between the peer and the server, **messageSender**, which is responsible for sending messages to other peers, **messageReceiver**, which is responsible for listening and receiving messages from other peers, and **room**, which is responsible for operations on the room, i.e. update the object/weapon list, enqueue the incoming request for the Token, process query about a specific object/weapon in the room and so on. The common ground of them is when they are doing their jobs, the game is still going on without interruption. Camelot provides two kinds of forms for thread declarations. One is the standard form used for class declarations, and the other are derived forms we mentioned in section 2.3.3. Derived forms for thread creation and use in Camelot contain a Java method **java.lang.Thread.setDaemon** of class **java.lang.Thread**. However, in this project,

we use the Java APIs provided by J2ME instead of J2SE, the former is much smaller than the latter. Some of the methods defined in J2SE are not included in J2ME, and **java.lang.Tharead.setDaemon** is one of these. There is no **java.lang.Thread.setDaemon** defined in J2ME's Thread class. Therefore, we have to use Camelot's standard form for class definition to define treads in our game. Figure 2.8 shows the definition of **connect**.

```
class connect = java.lang.Thread with
    field url : string
    field c : javax.microedition.io.HttpConnection
    field connM : connectionManager
    field tag : int
    ⋮
    ⋮
    maker (cManager : connectionManager) (urlocation : string) (cTag : int) =
        let _ = connM ← cManager
      in let _ = url ← urlocation
      in let _ = tag ← cTag
      in ()

    method run () : unit =
        let tag = (this#tag)
     in let conn = javax.microedition.io.Connector.open (this#url)
     in let _ = match conn with
                    hconn :> javax.microedition.io.HttpConnection →
                          let _ = c ← hconn
                        in let _ = this#c#setRequestMethod
                                   javax.microedition.io.HttpConnection.POST
                        in
                          ⋮
                          ⋮
end
```

Figure 2.8 Definition of Thread **connect**

2.4.3.4 Camelot Compiler Test Report

Up to now, although Camelot has been used in a wide range, most implementations

of it are small and aim for particular problems. The implementation of this project faces to a variety of problems. Thus, it is a good opportunity to test and improve the Camelot compiler. Followings are the improvements made to the compiler during this period. (All the fixing works on Camelot compiler were done by Dr. Kenneth MacKenzie, LFCS, Univ of Edinburgh.)

(1) Added new types, **byte**, **long**, and so on, for our need to access data like byte arrays. For example, **storeMyInfo** is a method in the **connectionManager** class, which is responsible for telling the Server the current status of the player. Since the output stream only accepts **byte** data, so the player's information has to be changed to **byte array** first.

(2) Fixed a parsing problem. In this project, there are some contexts where we only can put long name like "java.microedition.lcdui.Alert", but short class name like "connectionManager" are sometimes needed as well. The previous Camelot compiler cannot recognize the short class name. The solution to this problem perfects Camelot's OO features, since now we not only can define our own classes, but also can use them.

(3) Fixed the problem that the compiler sometimes failed to find methods which were defined in superclasses.

(4) Fixed the problem where the compiler confused the character type with the integer type, and therefore it couldn't find the correct method.

(5) Originally, datatypes weren't allowed to contain objects, for example, it was considered as a syntax error if we defined a type like

$$type\ t = A\ of\ \ java.math.BigInteger * \text{int} \mid \ldots$$

(6) Added the **isnull** construct. In [21], it is mentioned that considering in Java, any method with object return type may return the **null** object, Camelot provides a construct

isnull e

which tests if the expression e is a null value. However, in the previous compiler,

this construct has not been implemented yet.

(7) Released the constraint that variable names had to begin with lower-case letters. This improvement tones with some programmers' habits that they get used to have classes names begin with upper-case letters.

(8) Enabled the compiler to work on Windows system. There was a problem in the previous compiler where the compiler couldn't recognize filenames with colons in them like "E:\WTK22\lib", so at that time in Windows, the compiler could not recognize the name of the file which includes the Java classes we need.

Chapter 3 Inference of Heap-Space Bounds

The Goal of the Mobile Resource Guarantees (MRG) project is to develop Proof-Carrying Code (PCC) technology to endow mobile code with certificate of bounded resource consumption. These certificates are generated by a compiler (actually the combined compiler for Camelot and Grail) which, in addition to translating high-level programs into machine code, derives formal proofs based on programmer annotations and program analysis [22]. Programmer annotations enable programmers to express and manage storage, allocation explicitly. The program analysis takes space reuse by explicit deallocation into account and also furnishes an upper bound on the heap usage in the presence of garbage collection [23]. The program analysis relies on the type system which makes reference to the above programmer annotations. Linear Programming (LP) is used to automatically infer derivations in this enriched type system [23]. Following this line, this chapter first gives a literature review about linear programming, the program logic that connects with the outcome of space inference, and the core, inference for heap-space usage. After that, analysis of the practical inference work on our program will be given as well.

3.1 Linear Programming

A linear programming problem is one in which we are to find the maximum or minimum value of a linear expression

$$ax + by + cy + \ldots \quad \text{(called the objective function)}$$

subject to a number of linear constraints of the form

$$Ax + By + Cy + \ldots \leq N$$

or

$$Ax + By + Cy + \ldots \geq N \text{ (called linear inequality)}$$

The largest or smallest value of the objective function is called the optimal value, and a collection of values of x, y, z, … that gives the optimal value constitutes an optimal solution. The variables x, y, z, … are called the decision variables. And the feasible region determined by a collection of linear inequalities is the collection of points that satisfy all of the inequalities.

Use of the graphic method gives you a more straightforward comprehension of this problem and its solution set. For instance,

$$\begin{aligned} \text{minimize} \quad & Z = 3x + 4y \\ \text{subject to} \quad & 3x - 4y \le 12 \\ & x + 2y \ge 4 \\ & x \ge 1,\ y \ge 0 \end{aligned}$$

The feasible region for this set of constraints is shown below.

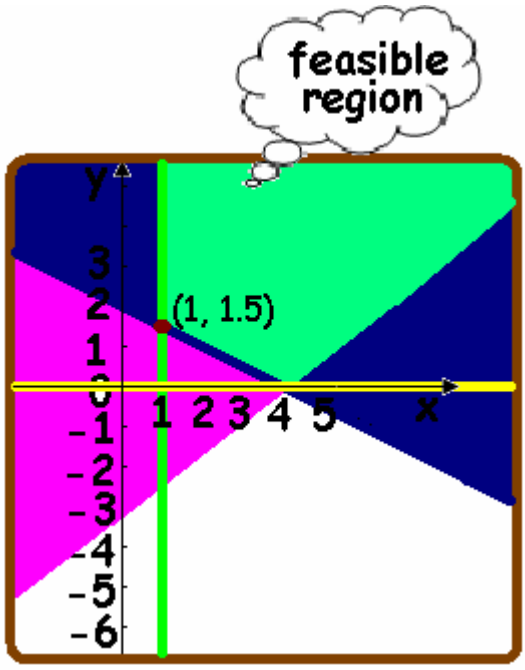


Figure 3.1: Feasible Region for LP Program

The following table shows the value of Z at each corner point:

Point	Z = 3x + 4y	
(1, 1.5)	3(1) + 4(1.5) = 9	**minimum**
(4, 0)	3(4) + 4(0) = 12	

Therefore, the solution is x = 1, y = 1.5, giving the minimum value Z = 9.

The graphic method can only effectively solve those linear programming problems involving two variables.

Usually, all minimization problems can be expressed in a standard form as follows [24]:

$$\text{Determine } x_1 \geq 0,\ x_2 \geq 0,\ \ldots\ x_n \geq 0 \qquad (3.1)$$

so as to

$$\text{minimize } Z = c_1x_1 + c_2x_2 + \ldots + c_nx_n \qquad (3.2)$$

subject to the constraint conditions expressed as equalities:

$$\begin{aligned} &a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1 \\ &a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2 \\ &\vdots \\ &a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n = b_m \end{aligned} \qquad (3.3)$$

The constants $c_j\,(j = 1, 2, \ldots, n)$ in the objective function are called *cost coefficients*; the constants $b_i (i = 1, 2, \ldots, m)$ defining the constraint requirements are called *stipulations*; and the constants $a_{ij} (i = 1, 2, \ldots, m$ and $j = 1, 2, \ldots, n)$ are called *structural coefficients*. The sign conditions imposed by (3.1) are known as the *non-negativity requirements*.

In practice, the constraint conditions generally appear as inequalities, with greater-than or less-than signs, or in a mixed form. In order to replace the inequalities constraint conditions by constraint equations, slack variables, which are positive, are introduced. Let, in a particular problem, the constraint condition appear as

$$a_{i1}x_1 + a_{i2}x_2 + \ldots + a_{ip}x_p \leq b_i$$

By adding a suitable positive quantity x_{p+1} to the left-hand side, the two sides can be equated, therefore the inequality constraint can be written as:

$$a_{i1}x_1 + a_{i2}x_2 + \ldots + a_{ip}x_p + x_{p+1} = b_i$$

If the constraint condition appears with a greater-than or equal-to sign, this can also be changed into an equation by subtracting the positive quantity x_{p+1} from the left-hand side:

$$a_{i1}x_1 + a_{i2}x_2 + \ldots + a_{ip}x_p - x_{p+1} = b_i$$

This standard form is adopted in the simplex method which is a widely used solution algorithm for solving linear programs. In the experiment section, we will use the simplex method to solve a linear programming problem generated by the type system and the specific problem logic.

3.2 The Program Logic

The program logic for reasoning about resource consumption of programs written in Grail is one of the essential components of the MRG project. The introduction to the program logic in this section will be based on [25] [26] [27].

Serving as the target logic of a certifying compiler, the program logic exploits Grail's dual nature of combing a functional interpretation with object-oriented features and a cost model for the JVM [25]. This program logic, together with the resource-aware operational semantics of Grail, has been formalized in the theorem prover Isabelle/HOL. The logic exists not only on the code producer side, where it is used by the certifying compiler to generate certificates, but also on the code consumer side, where it is used by the proof checker to check whether the certificate attached to the code indicates that the required resource of the program will not violate the resource policy the consumer claimed, and whether this certificate is logically correct.

This project uses the Hofmann/Jost type system to provide quantitative guarantees about the resource consumption. The concern of this type system is that given a functional program containing a function f of type, say, L(B) -> L(B), i.e., turning

lists of Booleans into lists of Booleans, find a function v such that the computation f(w) requires no more than v(w) additional heap cells [25]. By encoding a Grail-level interpretation of the Hofmann/Jost type system, a smooth transition from program analysis to program verification can be obtained.

3.2.1 The Syntax and Operational Semantics of Grail

The main characteristic of Grail is its dual identity: its (impure) call-by-value functional semantics may be shown to coincide with an imperative interpretation of the expansion of Grail programs into the Java Virtual Machine Language, provided that some mild syntactic conditions are met [27]. Certificates expressed in the program logic are obtained by computing in Isabelle/HOL some resource properties of heap-manipulating Grail programs that were produced by compiling Camelot programs. The formal syntax of Grail is shown in figure 3.2 [25] followed by a table explaining symbols appearing in the syntax:

$$
\begin{aligned}
a \in \mathrm{arg}\, s ::= \;& \mathrm{var}\, x \mid null \mid i \\
e \in \mathrm{exp}\, r ::= \;& \mathrm{null} \\
& \mid \mathrm{int}\, i \\
& \mid \mathrm{var}\, x \\
& \mid prim\ op\ x\ x \\
& \mid new\ c\left[\overline{t_i := x_i}\right] \\
& \mid x.t \\
& \mid x.t = y \\
& \mid c\diamondsuit t \\
& \mid c\diamondsuit t := x \\
& \mid \mathrm{let}\, x = e\ \mathrm{in}\ e \\
& \mid e; e \\
& \mid \mathrm{if}\, x\ \mathrm{then}\ e\ \mathrm{else}\ e \\
& \mid \mathrm{call}\, f \\
& \mid x \bullet m(\overline{a}) \\
& \mid c\diamondsuit m(\overline{a})
\end{aligned}
$$

Figure 3.2: The Syntax of Grail

Name	Representation
x	Variable
i	Integer
m	Method Name
c	Class Name
f	Function Name
	(i.e. labels of basic blocks)
t	(static) field names
op	primitive operator: $V=>V=>V$
	(i.e. arithmetic or comparison operator)
V	Semantic category of values, Comprising integers, references r, and the special symbol $\perp$, absence of a value
$null$	Null Heap Reference
$x.t$	Same as the non-static getfield instruction
$x.t:=y$	Same as the non-static putfield instruction
$c◊t$	Same as the static getfield instruction
$c◊t:=x$	Same as the static putfield instruction
let $x=e_1$ in e_2	The evaluation of e_1 returns an integer
	or reference value on top of the JVM stack
e_1 ; e_2	Sequential composition
new $c[t_1:=x_1,..., t_n:=x_n]$	Create and initial an object of class c according to the argument list
call f	Function call (i.e. immediate jumps) without arguments
$c.m(\bar{a})$	Static method invocation
$x.m(\bar{a})$	Instance method invocation

Table 3.1: Explanation of Grail's Syntax

Operational semantics of Grail based on its functional interpretation are shown in figure 3.3, with judgements in the form:

$$E \vdash h, e \Downarrow (h', v, p)$$

meaning that "in variable environment E and initial heap h, code e evaluates to the value v, yielding the heap h' and consuming p resources" [25]. Here "resources" refers to a tuple of four counters: $p = \langle clock \;\; callc \;\; invkc \;\; invkdpth \rangle$. Representation of these four components is shown in table 3.2. The operational semantics and the program logic employ two operators on resources, p ⊕ q and p ∪ q. For their effects on the resources components, refer to table 3.3.

$$\frac{}{E \vdash h, \text{null} \Downarrow (h, null, \langle 1\ 0\ 0\ 0 \rangle)}(\text{NULL}) \quad \frac{}{E \vdash h, \text{int } i \Downarrow (h, i, \langle 1\ 0\ 0\ 0 \rangle)}(\text{INT})$$

$$\frac{}{E \vdash h, \text{var } x \Downarrow (h, E\langle x \rangle, \langle 1\ 0\ 0\ 0 \rangle)}(\text{VAR})$$

$$\frac{}{E \vdash h, \text{prim } op\ x\ y \Downarrow (h, op\ (E\langle x \rangle)\ (E\langle y \rangle), \langle 3\ 0\ 0\ 0 \rangle)}(\text{PRIM})$$

$$\frac{E\langle x \rangle = \text{Ref } l}{E \vdash h, x.t \Downarrow (h, h(l).t, \langle 2\ 0\ 0\ 0 \rangle)}(\text{GETF}) \quad (\text{Ref } l: \text{ heap reference of location l})$$

$$\frac{E\langle x \rangle = \text{Ref } l}{E \vdash h, x.t := y \Downarrow (h[l.t \mapsto E\langle y \rangle], \perp, \langle 3\ 0\ 0\ 0 \rangle)}(\text{PUTF})$$

$$\frac{}{E \vdash h, c \diamond t \Downarrow (h, h(c).t, \langle 2\ 0\ 0\ 0 \rangle)}(\text{GFST})$$

$$\frac{}{E \vdash h, c \diamond t := y \Downarrow (h[c.t \mapsto E\langle y \rangle], \perp, \langle 3\ 0\ 0\ 0 \rangle)}(\text{PFST})$$

$$\frac{l = freshloc(h)}{E \vdash h, new\ c[\overline{t_i := x_i}] \Downarrow (h[l \mapsto (c, \{\overline{t_i := E\langle x_i \rangle}\})], \text{Ref } l, \langle (n+1)\ 0\ 0\ 0 \rangle)}(\text{NEW})$$

(*freshloc*(*h*) returns a fresh location outside the domain of *h*)

$$\frac{E\langle x\rangle = \text{true} \quad E \vdash h, e_1 \Downarrow (h_1, v, p)}{E \vdash h, \text{if } x \text{ then } e_1 \text{ else } e_2 \Downarrow (h_1, v, \langle 2\ 0\ 0\ 0\rangle \oplus p)} \text{(IFTRUE)}$$

$$\frac{E\langle x\rangle = \text{false} \quad E \vdash h, e_2 \Downarrow (h_1, v, p)}{E \vdash h, \text{if } x \text{ then } e_1 \text{ else } e_2 \Downarrow (h_1, v, \langle 2\ 0\ 0\ 0\rangle \oplus p)} \text{(IFFALSE)}$$

$$\frac{E \vdash h, e_1 \Downarrow (h_1, w, p) \quad w \neq \perp \quad E\langle x := w\rangle \vdash h_1, e_2 \Downarrow (h_2, v, q)}{E \vdash h, \text{let } x = e_1 \text{ in } e_2 \Downarrow (h_2, v, \langle 1\ 0\ 0\ 0\rangle \oplus (p \cup q))} \text{(LET)}$$

$$\frac{E \vdash h, e_1 \Downarrow (h_1, \perp, p) \quad E \vdash h_1, e_2 \Downarrow (h_2, v, q)}{E \vdash h, e_1; e_2 \Downarrow (h_2, v, (p \cup q))} \text{(COMP)}$$

$$\frac{E \vdash h, snd(FT\ f) \Downarrow (h_1, v, p)}{E \vdash h, \text{call } f \Downarrow (h_1, v, \langle 1\ 0\ 0\ 0\rangle \oplus p)} \text{(CALL)}$$

(FT, the function table, used to obtain function bodies from names)

$$\frac{\left(newframe\ null\ fst(MT\ c\ m)\ \bar{a}\ E\right) \vdash h, snd(MT\ c\ m) \Downarrow (h_1, v, p)}{E \vdash h, c \diamondsuit m(\bar{a}) \Downarrow (h_1, v, \langle (2 + |\bar{a}|)\ 0\ 1\ 1\rangle \oplus p)} \text{(SINV)}$$

(MT, the method table, used to obtain method bodies from names)

(*newframe* creates an appropriate environment)

$$\frac{classOf\ E\ h\ x\ c \quad \left(newframe\ E\langle x\rangle\ fst(MT\ c\ m)\ \bar{a}\ E\right) \vdash h, snd(MT\ c\ m) \Downarrow (h_1, v, p)}{E \vdash h, x \bullet m(\bar{a}) \Downarrow (h_1, v, \langle (4 + |\bar{a}|)\ 0\ 1\ 1\rangle \oplus p)} \text{(VINV)}$$

(*classOf* retrieves the dynamic class name c
associated to the object pointed to by x)

Figure 3.3: The Dynamic Semantics of Grail

Component	Representation
clock	a global abstract instruction counter
callc	the number of function calls (jump instructions)
invkc	the number of method invocations
invkdpth	the maximal invocation depth

Table 3.2: Representation of Resource's Four Components

Component	Operator	Implementation
clock	$\oplus$, $\cup$	point-wise addition
callc	$\oplus$, $\cup$	point-wise addition
invkc	$\oplus$, $\cup$	point-wise addition
invkdpth	$\oplus$	point-wise addition
	$\cup$	pick the maximum

Table 3.3: Operations on Resource

Notice that instead of being explicitly recorded in the resource model, the size of heap is deduced from $|dom(h)|$ which represents the domain of the object heap.

3.2.2 The Program Logic

The MRG project employs the Proof-Carrying Code infrastructure which equips Grail programs with certificates concerning their resource consumption. Certificates contain a claim of resource usage together with a proof of the claim [12]. The proof expressed in a program logic for Grail follows a custom logic of partial correctness. Sequents are of the form:

$$\Gamma \triangleright e : P$$

which means that a Grail expression $e \in expr$ is related to a specification P under some set of assumptions Γ of the same form. The specification P denotes a predicate which can constrain possible executions of e with respect to the dynamic semantic

pointed out above. Specifications can refer to the initial and final heaps of a program expression, the initial environment, the resources consumed and the result value [25]. The uniform judgement in the program logic is:

$$\vDash e : \lambda\ E\ h\ h'\ v\ p.\ P\ E\ h\ h'\ v\ p$$

which means that whenever the execution of e for initial heap h and environment E terminated and delivers final heap h', result v and resources p, P is satisfied, that is that $E \vdash h, e \Downarrow (h', v, p)$ implies $P\ E\ h\ h'\ v\ p$. Figure 3.4 shows the concrete rules of this logic. As we will see, besides rules for each form of Grail expression e∈expr, two basic rules are provided as well.

$$\frac{(e,P) \in \Gamma}{\Gamma \rhd e : P}(\text{VAX}) \qquad \frac{\Gamma \rhd e : P \quad \forall Ehh'vp.PEhh'vp \rightarrow QEhh'vp}{\Gamma \rhd e : P}(\text{VCONSEQ})$$

$$\frac{}{\Gamma \rhd \text{null} : \lambda Ehh'vp.h' = h \wedge v = null \wedge p = \langle 1\ 0\ 0\ 0 \rangle}(\text{VNULL})$$

$$\frac{}{\Gamma \rhd \text{int}\, i : \lambda Ehh'vp.h' = h \wedge v = i \wedge p = \langle 1\ 0\ 0\ 0 \rangle}(\text{VINT})$$

$$\frac{}{\Gamma \rhd \text{var}\, x : \lambda Ehh'vp.h' = h \wedge v = E\langle x \rangle \wedge p = \langle 1\ 0\ 0\ 0 \rangle}(\text{VVAR})$$

$$\frac{}{\Gamma \rhd \text{prim}\ op\ x\ y : \lambda Ehh'vp.v = op\ E\langle x \rangle E\langle y \rangle \wedge h' = h \wedge p = \langle 3\ 0\ 0\ 0 \rangle}(\text{VPRIM})$$

$$\frac{}{\Gamma \rhd x.t : \lambda Ehh'vp.\exists l.E\langle x \rangle = \text{Ref}\ l \wedge h' = h \wedge v = h'(l).t \wedge p = \langle 2\ 0\ 0\ 0 \rangle}(\text{VGETF})$$

$$\frac{}{\Gamma \rhd x.t := y : \lambda Ehh'vp.\exists l.E\langle x \rangle = \text{Ref}\ l \wedge p = \langle 3\ 0\ 0\ 0 \rangle \wedge h' = h[l.t \mapsto E\langle y \rangle] \wedge v = \perp}(\text{VPUTF})$$

$$\frac{}{\Gamma \rhd c \diamondsuit t : \lambda Ehh'vp.h' = h \wedge v = h(c).t \wedge p = \langle 2\ 0\ 0\ 0 \rangle}(\text{VGETST})$$

$$\frac{}{\Gamma \rhd c \diamondsuit t : \lambda Ehh'vp.h' = h[c.t \mapsto E\langle y \rangle] \wedge v = \perp \wedge p = \langle 3\ 0\ 0\ 0 \rangle}(\text{VPUTST})$$

$$\frac{}{\Gamma \rhd \text{new } c\left[\overline{t_i := x_i}\right] : \lambda Ehh'vp.\exists l.l = \mathit{freshloc}(h) \wedge p = \langle (n+1)\ 0\ 0\ 0\rangle} (\text{VNEW})$$
$$\wedge h' = h\left[l \mapsto \left(c, \left\{\overline{t_i := E\langle x_i \rangle}\right\}\right)\right] \wedge v = \text{Ref } l$$

$$\frac{\Gamma \rhd e_1 : P_1 \quad \Gamma \rhd e_2 : P_2}{\Gamma \rhd \text{if } x \text{ then } e_1 \text{ else } e_2 : \lambda Ehh'vp.\exists p'.p = p' \oplus \langle 2\ 0\ 0\ 0\rangle \wedge} (\text{VIF})$$
$$\left(E\langle x \rangle = \mathit{true} \rightarrow P_1 Ehh'vp'\right) \wedge$$
$$\left(E\langle x \rangle = \mathit{false} \rightarrow P_2 Ehh'vp'\right) \wedge$$
$$\left(E\langle x \rangle = \mathit{true} \vee E\langle x \rangle = \mathit{false}\right)$$

$$\frac{\Gamma \rhd e_1 : P_1 \quad \Gamma \rhd e_2 : P_2}{\Gamma \rhd \text{let } x = e_1 \text{ in } e_2 : \lambda Ehh'vp.\exists p_1 p_2 h_1 w.\left(P_1 Ehh_1 wp_1\right) \wedge w \neq \bot \wedge} (\text{VLET})$$
$$\left(P_2\left(E\langle x := w\rangle\right)h_1 h'vp_2\right) \wedge$$
$$p = \langle 1\ 0\ 0\ 0\rangle \oplus \left(p_1 \cup p_2\right)$$

$$\frac{\Gamma \rhd e_1 : P_1 \quad \Gamma \rhd e_2 : P_2}{\Gamma \rhd e_1; e_2 : \lambda Ehh'vp.\exists p_1 p_2 h_1.P_1 Ehh_1 \perp p_1 \wedge} (\text{VCOMP})$$
$$P_2 Eh_1 h'vp_2 \wedge p = \left(p_1 \cup p_2\right)$$

$$\frac{\Gamma \cup \left\{\left(\text{call } f, P\right)\right\} \rhd \mathit{snd}\left(FT\ f\right) : \lambda Ehh'vp.PEhh'v\langle 1\ 1\ 0\ 0\rangle \oplus p}{\Gamma \rhd \text{call } f : P} (\text{VCALL})$$

$$\frac{\Gamma \cup \left\{\left(c \diamondsuit m\left(\overline{a}\right), P\right)\right\} \rhd \mathit{snd}\left(MT\ c\ m\right) : \lambda Ehh'vp.\forall E'.E = \left(\mathit{newframe}\ \mathit{null}\ \mathit{fst}\left(MT\ c\ m\right)\overline{a}\ E'\right) \rightarrow PE'hh'v\left\langle\left(2+\overline{a}\right)\ 0\ 1\ 1\right\rangle \oplus p}{\Gamma \rhd c \diamondsuit m\left(\overline{a}\right) : P} (\text{VSINV})$$

$$\frac{\Gamma \cup \left\{\left(x \bullet m\left(\overline{a}\right), P\right)\right\} \rhd \mathit{snd}\left(MT\ c\ m\right) : \lambda Ehh'vp.\forall E'.(\mathit{classOf}\ E\ h\ x\ c \wedge E = (\mathit{newframe}\ (E'\langle x\rangle)\ \mathit{fst}\left(MT\ c\ m\right)\overline{a}\ E')) \rightarrow (E', h, h', v\left\langle\left(4+\overline{a}\right)\ 0\ 1\ 1\right\rangle \oplus p) \in P}{\Gamma \rhd x \bullet m\left(\overline{a}\right) : P} (\text{VVINV})$$

$$\frac{finite(D)\ \ D \rhd e : P\ \ G \subseteq D\ \ provable(D,G)}{G \rhd e : P}(\text{VCUT})$$

$$\frac{goodContext\ MST\ G\quad finite(G)\quad \left(c.m(y), MST\ m\ \left(y@\left[null\right]\right)\right) \in G}{0 \rhd c.m(z) : MST\ m\left(z@\left[null\right]\right)}(\text{VADAPTS})$$

Figure 3.4: The Program Logic for Grail

The ***goodContext*** property requires that whenever a method invocation is associated to its specification table entry in G, the method body satisfies the specification for any arguments passed to the body via the formal parameters. [25] also proves soundness and completeness of this program logic, which establishes a solid and convincing basis for the usage of the logic.

3.3 Inference of Heap-Space Bounds

The Diamond type in Camelot, together with its resource-aware type system, allows the heap usage of Camelot programs to be inferred. Next we use lfd_infer, the practical implementation to understand the procedure of getting an inference for heap-space usage.

3.3.1 Introduction of lfd_infer

lfd_infer is the implementation aiming for the static prediction of heap space usage for first-order functional programs as proposed by M.Hofmann and S.Jost in 2003. The process of type inference first sets up a set of linear inequalities over the rational variables on the fly by reconstructing a typing derivation for a given program. These equalities, together with a rather arbitrary objective function, form a linear program, which is then fed to an external LP-solver where an optimal solution for this linear programming problem is calculated. Making use of the solution set, an annotated typing for each of the program's functions is established, as well as a linear bound on each function's heap space consumption. Some major features of this implementation are:

(1) All build-in types are assumed to be unboxed, i.e. heap free, including the string type. This is not a bug although strings indeed use up heap-space. The reason for that is in most cases the use of strings is merely providing a convenient way for screen-printing. If heap space consumption of strings is a concern, they can be treated as lists of characters, or alternatively, be restricted to a predefined length.

(2) For compatibility with Camelot, lfd_infer distinguishes a destructive- from a read-only pattern match. The syntax is as:

```
match x with
| Cons(a, b, c) -> …        (* Read-only match *)
| Cons(a, b, c)@_ -> …      (* Destructive match, cell goes to freelist *)
| Cons(a, b, c)@d -> …      (* Destructive match, cell bound to d of type <> *)

Cons(a, b, c)               (* Allocate heapcell from freelist *)
Cons(a, b, c)@_             (* Allocate heapcell from freelist *)
Cons(a, b, c)@d             (* Use heapcell bound to d *)
```

Of course, there is only one match operation is allowed for each constructor among the above possibilities. Furthermore, in this system, a node of a data structure on the heap can only be destroyed and built anew, but it cannot be updated.

(3) lfd_infer allows programmers to enforce some annotations as they see fit. This can be done using the enriched or annotated typing facility that it provides.

(4) Since modularity is inherent to the inference, lfd_infer supports partial inference [28]. Any function, which is defined but not declared, i.e. there is no val-statement corresponding to it, will be ignored in the inference. However, the inference will fail if such functions are called by any other declared and defined function of the program. On the contrary, any defined function is checked based on their given declaration. If a function is undefined, then all the annotations related to it are assumed to be zero, or accepted as given.

We have mentioned lfd_infer's enriched typing above, which calculates resource annotations for functions and is therefore able to provide programmers with helpful information on resource assumption analysis. This annotated type is like this:

$$\text{insert} : 1,\ \text{int} \to \mathit{list}[\mathit{Nil}(0) \mid \mathit{Cons}(\text{int}, \#, 0)]$$
$$\to \mathit{list}[\mathit{Nil}(0) \mid \mathit{Cons}(\text{int}, \#, 0)], 0;$$

where for each occurring type, all its constructors are listed, and for each constructor, a number indicating the amount of heap cells that must be supplied for each node contained in the input of that type is added. The first-hand information from the annotated type is that at most how many heap cells are required at the beginning for the sake of normally starting and executing a function, and that how many heap cells will be returned after its execution. So, for example, the type above says that a call to **insert** may allocate one heap cell during its execution, and do not return any heap cell after executing.

On the other hand, lfd_infer has an obvious deficiency which was also admitted by its creators. Leaving programmers to designate for each pattern match whether it is destructive or not is a kind of burden for them, and is dangerous for programs. A program might crash if a destructive pattern match is not the last access to that data. Although there is an ongoing research aiming at this problem, programmers currently still have to devote some attention on this problem when designing their programs.

3.3.2 Further Discussion About Heap Space Usage

3.3.2.1 Heap Usage Problem Caused by Camelot's Object-Oriented Feature

As described earlier, the presence of mutable objects in object-oriented Camelot provides in-place update [20]. However by allowing arbitrary object creation, the unbounded heap-usage problem solved for datatypes recurs on this occasion. Perhaps more seriously, invocating arbitrary Java code in Camelot programs might lead to an unlimited heap space usage.

For the first problem, the first attempt to directly adapt the idea of diamonds seems unrealistic, since it is hardly appropriate to represent every Java object uniformly by

an object of one class. So an abstract diamond is proposed instead, which represents the heap storage but used by an arbitrary object. Any requirement is satisfied by using **new** to supply one of these diamonds. But a problem of this approach is that reclamation of such abstract diamonds would only correspond to making an object available for garbage collection, rather than definitely being able to re-use the storage. Even so, such a system might be able to give a measure of the total number of objects created and the maximum number in active use simultaneously [20].

For the second problem, although there is some way to place a bound on the heap space used by the new OO features within a Camelot program, external Java code may use arbitrary amounts of heap. With respect to this problem, [29] proposes and analyzes three approaches. The first is that only external classes which come with proofs of bounded heap usage are allowed to be used. But constructing a resource-bounded Java class library or inferring resource bounds for an existing library would be massive work, even for the smaller class libraries used with mobile devices. The second approach is to leave programmers or library creators to state the resource usage of the external methods they produce. This requires extending the trusted computing base in the sense of resources, but seems a more reasonable solution [29]. The third is to only take account of resources consumed by Camelot code. This suggestion seems somewhat unrealistic, as one could easily cheat by using Java libraries to do some memory-consuming “dirty work” [29].

3.3.2.2 Heap Usage Problem Caused by Threads

The analysis of memory consumption of Camelot programs is based on the consumption of memory by heap-allocated data structures. The present analysis of Camelot programs is based on a single-threaded architecture. To assist with the development of the thread management system in Camelot, analysis methods for multiple threads’ memory usage need to be considered. Table 3.4 concludes solutions for this problem proposed in [20].

Approach	*Description*	*Strongpoint*	*Shortcoming*
Shared free list	A single free list of storage is shared across all threads	Efficient memory usage	Run-time penalty caused by synchronization (an overhead of locking and unlocking the parent of the field occurs when entering and leaving a critical section)
Private free list	Each thread separately maintains its own local instance of the free list	No requirement for access to the free list to be synchronized, therefore time is saved	Memory use penalty: There will be times when one thread allocates memory while another thread has unused memory on its local free list.
Hybrid free list	Each thread has a local free list, meanwhile, a global free list exits as well	Reduce the overhead of calls to access the synchronized global free list, while preventing threads from keeping too many unused memory cells locally.	The analysis of this approach is complicated.

Table 3.4: Solutions for the Heap Usage Problem of Multi-Threads

Camelot chooses the first scheme. In addition to it, a requirement is imposed to ease the predictability of memory usage, which is that data structures in a multi-threaded Camelot program are not shared across threads. This requirement means that the space consumption of a multi-threaded Camelot program is obtained as the sum of per-thread space allocation plus the space requirements of the threads themselves [20].

3.4 Practical Work on the Inference of Heap-Space Bounds

This section will use the lfd_infer to analyze the heap space usage of our game program. Since functional objects allowed in Camelot are currently not yet accounted for in lfd_infer, so we have to pick out some pure Camelot code and deploy the experiment on them.

3.4.1 Verifying Correctness of the Inference Result

The Camelot code named simple.cmlt is quite simple:

```
let hd l = match l with [] -> 0  | (h::_) -> h
```

The inference result of the heap consumption is:

```
hd : 0, list_1[0|int,#,0] -> int, 0;
```

which means that ***hd*** is a function taking as argument an integer list, and return an integer after computation. There is no extra heap-cell required excepts for those originally allocated to hold the input list.

Figure 5 shows all generated constraints which are used for inference.

```
/*
   This file is an automatically generated lp for 'lp_solve'.
(simple.constraints)
   Contains 4 inequalties in 6 variables.
*/

/*
   hd              : x01, list_1[u01|int,#,u02] -> int, y01;
*/

MIN:   +4*u01 +4*u02 +2*x01 -1*y01 ;

hd__11_Val:     -1*a02     +1*y01     <= 0 ;
hd__11_M'2:      +1*a02     -1*u02     -1*x01     <= 0 ;
hd__10_Val:     -1*a01     +1*y01     <= 0 ;
hd__10_M'1:      +1*a01     -1*u01     -1*x01     <= 0 ;
a01 >= 0 ;
a02 >= 0 ;
u01 >= 0 ;
u02 >= 0 ;
x01 >= 0 ;
y01 >= 0 ;
a01 <= 10000 ;
a02 <= 10000 ;
u01 <= 10000 ;
u02 <= 10000 ;
x01 <= 10000 ;
```

Figure 3.5: simple.constraints: All constraints for Inference

The linear problem we pick out from it is:

Minimize: +4*u01 +4*u02 +2*x01 -1*y01 ;

Subject to:

hd__11_Val: -1*a02 +1*y01 <= 0 ; (1)

hd__11_M'2: +1*a02 -1*u02 -1*x01 <= 0 ; (2)

hd__10_Val: -1*a01 +1*y01 <= 0 ; (3)

hd__10_M'1: +1*a01 -1*u01 -1*x01 <= 0 ; (4)

a01, a02, u01, u02, x01, y01 >= 0

(a01, a02, u01, u02, x01, y01 <= 1000 is the upper bound which are predefined by the system)

Add (1) and (2) to remove a02, and the same to (3) and (4) to remove a01, we can get:

Minimize: Z = +4*u01 +4*u02 +2*x01 -1*y01 ;

Subject to:

y01 – u02 – x01 ≤ 0

y01 – u01 – x01 ≤ 0

u01, u02, x01, y01 >= 0

Then we will use the Simplex Method introduced in Section 3.1 to find the optimal solution for this problem.

Step 1. Introduce slack variables to the constraints and rewrite the objective function in standard form.

-u02 – x01 + y01 + s01 = 0

-u01 – x01 + y01 + s02 = 0

4u01 + 4u02 + 2x01 – y01 – Z = 0

Step 2. Write down the initial tableau.

	u01	**u02**	**x01**	**y01**	**s01**	**s02**	**Z**	**Ans**
Cons1	0	-1	-1	1	1	0	0	0
Cons2	-1	0	-1	1	0	1	0	0
Min	4	4	2	-1	0	0	1	0

Step 3. Select the pivot column:

For this example, the pivot column is y01-column.

Step 4. Select the pivot in the pivot column.

	u01	**u02**	**x01**	**y01**	**s01**	**s02**	**Z**	**Ans**	**Ratio**
Cons1	0	-1	-1	1	1	0	0	0	0/1=0
Cons2	-1	0	-1	1	0	1	0	0	0/1=0
Min	4	4	2	-1	0	0	1	0	

So we pick up the 1 in the first row as the pivot.

Step 5. Use the pivot to clear the column in the normal manner.

	u01	u02	x01	y01	s01	s02	Z	Ans
Cons1	0	-1	-1	1	1	0	0	0
Cons2	-1	1	0	0	-1	1	0	0
Min	4	3	1	0	1	0	1	0

Step 6. Go to step 3.

All the entries on the last row are positive, so we can get the solution now, which is u01 = 0, u02 = 0, x01 = 0, y01 = 0, s01 = 0, s02 = 0, Z = 0

Put the values to their proper position in the structure:

hd : x01, list_1[u01|int,#,u02] -> int, y01

Then we can get the result

hd : 0, list_1[0|int,#,0] -> int, 0;

which is the same as the one lfd_infer computes.

3.4.2 Experiment Problem Report And Analysis

Since the whole Camelot code is quite large, for convenience, we separate them into several parts based on datatypes they operate on, and then carry out the experiment on each of these parts. During this period, some problems have been notified by lfd_infer, but after analysis, we find out that most of them result from shortcomings of lfd_infer itself. Next we will describe and analyze these problems.

Problem 1: LP For the Whole Program is Infeasible

Reason: Misuse of Destructive Pattern Match

This problem happens when lfd_infer tries to solve **stringpairlist.cmlt** which collects all functions on the datatype **stringpairlist**, and **stringintlist.cmlt** which collects all functions on the datatype **stringintlist**. Actual implementation can be

found in Appendix F. Take **stringpairlist.cmlt** for example, we finally discovered that the problem is ascribed to the function:

```
let showStringPairList (l : stringpairlist) (k : int) (n : int)
                       (sink : string) : string =
            if k < n
            then begin
                  match ssElementAt l k with
                        Pair2 (key, value) ->
                          let sink = sink ^ "player: " ^ key ^ "
                                     phone: " ^ value ^ "\n"
                              in showStringPairList l (k+1) n sink
                      | None2 -> sink
            end
            else sink
```

And this function further calls the function:

```
let ssElementAt (l : stringpairlist) k =
                  match l with Nil4 -> None2
                               Cons4 (a,b,t) ->
                                       if k > 0
                                       then ssElementAt t (k-1)
                                       else Pair2 (a,b)
```

which has LFD type:

```
ssElementAt: 1, stringpairlist[0|string,string,#,0] → int
             → stringstringpair[1|string,string,0], 0;
```

This means that to execute function **ssElementAt** without running out of memory, we need at least one free heap cell available before the function starts. This is due to the possibility of executing the constructor *Pair2 (a,b)*, which needs a heap cell to hold the pair. On the other hand, since we don't know values of parameters ***n*** and ***k*** in advance, it is impossible to decide how often **showStringPairList** calls **ssElementAt** and therefore how much memory **showStringPairList** will need. This is the reason for LP being infeasible. Furthermore, instead of halting at this point, lfd_infer continues computing heap usage for the remaining functions, so we still

obtain an LFD type for **showStringPairList**, even though it is wrong. (The LFD type below says that **showStringPairList** doesn't consume any memory, but this is incorrect, since we need some space to store the input list.)

```
showStringPairList: 0, stringpairlist[0|string,string,#,0] → int → int
                    → string → string, 0;
```

The solution to this problem is to append "@_" to the pair constructor in **showStringPairList** as follows (since we don't need the pair returned by **ssElementAt** again, we can throw it away):

```
match ssElementAt l k with
                    Pair2 (key, value)@_  → ...
```

As a result, the program becomes feasible and corresponding correct LFD types are:

```
ssElementAt:  1, stringpairlist[0|string,string,#,0] → int
              → stringstringpair[1|string,string,0], 0;

showStringPairList: 1, stringpairlist[0|string,string,#,0] → int → int
                    → string → string, 1;
```

The type for **showStringPairList** says that it needs at least one free heap cell before it starts and leaves at least one free heap cell when it returns. This heap cell is used up in the call to **ssElementAt**, but is recovered when the destructive match is performed. And again this heap cell is used the next time that **ssElementAt** is invoked. When we eventually return from showStringPairList, this heap cell will be reclaimed to the freelist for later use by other functions, so we get the return type string, 1.

Problem 2: Memory Leak

Reason: lfd_infer

In the lfd_infer manual [28], memory leak problem is explained as the loss of references to heap cells. Actually, lfd_infer's judgement for this problem is not accurate. Our code is a proof of this. Following is a function in

objectlist.cmlt(Appendix F):

```
let clearObjectList (l : objectlist) =
                    match l with Nil -> Nil
                    Cons (h,t)@_ -> clearObjectList t
```

lfd_infer gives a LFD type:

clearObjectList: 0, objectlist[0|obj[u07|int,string,int,int,string,u08],#,0]
→ unit, 0;

and then prints a message warning of a the memory leak with an inequality:

clearObjectList_163_Ma2: -1×K1 +1×a69 -1×u06 -1×x03 <= 0 ;

which means that the memory leak occurs in the branch of a destructive pattern match on the second constructor of the matched type. We can see in this branch there is a reference for a heap cell (represented by -1×K1 in the inequality) released due to the destructive match. Since there is not a datatype constructor from that point, lfd_infer falsely assumes that the heap cell pointed to by the released reference has been lost, and therefore warns that a memory leak occurs. In fact, there is no leak. **clearObjectList** works properly. The heap cell released by the destructive pattern has been reclaimed and put in the free list (recall the definition of "@_"). If we give a list of length n as input, there will be n extra heap cells put on the free list when **clearObjectList** returns. The essential reason is that the LFD type system is not strong enough to express this. Since **clearObjectList** returns a unit type, the only possible return types for the function are things like:

unit, 0
unit, 1
⋮
unit, n $(n \geq 1)$

Values appearing in LFD types have to be constants, and the best the LFD type system can is to say that the return type of **clearObjectList** is unit, 0. The 0 will at least be a correct bound for the amount of heap space obtained when we call **clearObjectList**. To get a correct answer, we'd need a type like

unit, n (where n is the length of the input list)

But lfd_infer's type system currently cannot express this. Therefore, the "memory leak" in lfd_infer sometimes is not a memory leak. However, in order to pass the inference of lfd_infer, we modify the code a little:

```
type objectlist = !Nil | Cons of obj * objectlist
type obj = !NullObject | Object of int*string*int*int*string

let clearObjectList (l : objectlist) =
        match l with Nil -> Nil
                | Cons (h,t)@_ ->
                    Cons(NullObject, clearObjectList t)
```

The modification seems artificial. Since constructor NullObject takes no space, this function uses the same heap space as the original one. But the lfd_infer passes it, first because the second Cons could subsequently gain the reference released by the destructive pattern match, and second because the return type is **objectlist** rather than unit. We use the same trick to solve the memory leak reported when we analyse other programs.

Problem 3: Unbounded Heap Usage

Reason: lfd_infer;

When parsing stringpairlist.cmlt and stringintlist.cmlt, we get LFD types suck as the following, taking function **siElemenetAt** in **stringintlist.cmlt** for example:

```
siElementAt: 10000, stringintlist[0|string,int,#,0] → int
                → stringintpair[10000|string,int,9999], 0;
```

This means that the heap space required by function **siElemenetAt** is unbounded when it starts executing, as 10000 is the default highest value of the number of heap cells available. From the view of linear programming, we can use the following figure 3.6 to explain this unboundedness problem. As we can see, it is impossible for the objective function to find a point of intersection with either constraint inequality in the feasible region within a finite value. In this case, lfd_infer just

simply assumes that the consumption reaches the upper bound, and this is why we see 10000 or 9999 in the LFD type. In order to obtain a finite solution, the programmer has to use the options –olhs, -orhs, -odin, -odout provided by lfd_infer to change the objective function himself. Figure 3.7 shows how this change works. For our program, after using option **–olhs 4**, we can get a reasonable LFD type:

siElementAt: 1, stringintlist[0|string,int,#,0] → int
→ stringintpair[1|string,int,0], 0;

Although the problem is solved finally, the penalty for it is that programmers themselves have to manually change the objective function so as to get a reasonable solution. This obviously imposes a burden on the programmer.

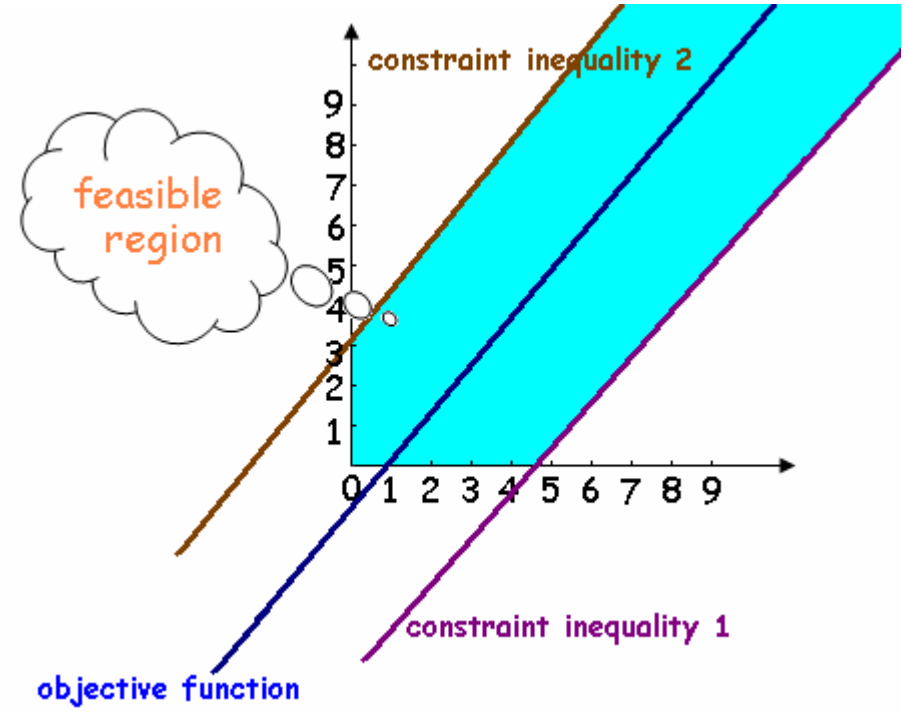


Figure 3.6: An Unbounded Case in Linear Programming

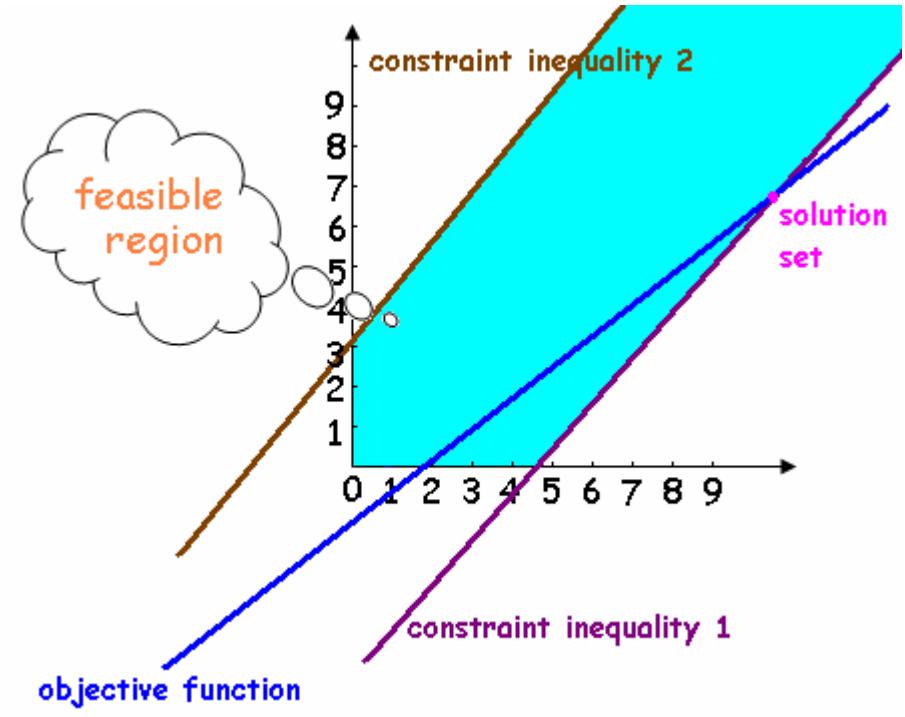


Figure 3.7: Changed Linear Problem

3.4.3 Heap Consumption of This Project

Although our project implements a multi-threaded game system, constrained by the capability of the inference tool, our experiment results are oriented to a single thread. On the other hand, no thread in our program uses those pure Camelot functions (without operations on objects) we mentioned here. So for this project, heap space usage of pure Camelot codes for one thread is equal to it for multi-threads. Next I will list the inference results of all the pure Camelot codes in this project, which are separated by datatypes codes operate on.

stringpairlist:

$$clearStringPairList:\ 0,\ stringpairlist[0 \mid string, string, \#, 0] \rightarrow stringpairlist[0 \mid string, string, \#, 0],\ 0;$$

$$copyStringPairList:\ 0,\ stringpairlist[0 \mid string, string, \#, 1] \rightarrow stringpairlist[0 \mid string, string, \#, 0] \rightarrow stringpairlist[0 \mid string, string, \#, 0],\ 0;$$

$$findNumber\ :\ 0,\ stringpairlist[0 \mid string, string, \#, 0] \rightarrow string \rightarrow string,\ 0;$$

$$getValue\ :\ 0,\ stringpairlist[0 \mid string, string, \#, 0] \rightarrow string \rightarrow string,\ 0;$$

$$rmPlayer\ :\ 0,\ stringpairlist[0 \mid string, string, \#, 0] \rightarrow string \rightarrow stringpairlist[0 \mid string, string, \#, 0],\ 0;$$

$$same\_string:\ 0,\ string \rightarrow string \rightarrow bool,\ 0;$$

$$showStringPairList:\ 1,\ stringpairlist[0 \mid string, string, \#, 0] \rightarrow \text{int} \rightarrow \text{int} \rightarrow string \rightarrow string,\ 1;$$

$$sizeof\ :\ 0,\ stringpairlist[0 \mid string, string, \#, 0] \rightarrow \text{int},\ 0;$$

$$ssElementAt:\ 1,\ stringpairlist[0 \mid string, string, \#, 0] \rightarrow \text{int} \rightarrow stringstringpair[1 \mid string, string, 0],\ 0;$$

$$stringAt\ :\ 0,\ stringlist[0 \mid string, \#, 0] \rightarrow \text{int} \rightarrow string,\ 0;$$

stringintlist:

$clearStringIntList:\ 0,\ string\ \text{int}\ list[0 \mid string, \text{int}, \#, 0] \rightarrow string\ \text{int}\ list[0 \mid string, \text{int}, \#, 0],\ 0;$

$concatStringIntList:\ 1,\ string\ \text{int}\ list[0 \mid string, \text{int}, \#, 0] \rightarrow \text{int} \rightarrow \text{int} \rightarrow string \rightarrow string,\ 1;$

$copyStringIntList:\ 0,\ string\ \text{int}\ list[0 \mid string, \text{int}, \#, 1] \rightarrow string\ \text{int}\ list[0 \mid string, \text{int}, \#, 0] \rightarrow string\ \text{int}\ list[0 \mid string, \text{int}, \#, 0],\ 0;$

$findRoom\ :\ 0,\ string\ \text{int}\ list[0 \mid string, \text{int}, \#, 0] \rightarrow string \rightarrow \text{int},\ 0;$

$lengthof\ :\ 0,\ string\ \text{int}\ list[0 \mid string, \text{int}, \#, 0] \rightarrow \text{int},\ 0;$

$removeFirst:\ 0,\ string\ \text{int}\ list[0 \mid string, \text{int}, \#, 0] \rightarrow string\ \text{int}\ list[0 \mid string, \text{int}, \#, 0],\ 0;$

$same\_string:\ 0,\ string \rightarrow string \rightarrow bool,\ 0;$

$showStringIntList:\ 1,\ string\ \text{int}\ list[0 \mid string, \text{int}, \#, 0] \rightarrow \text{int} \rightarrow \text{int} \rightarrow string \rightarrow string,\ 1;$

$siElementAt:\ 1,\ string\ \text{int}\ list[0 \mid string, \text{int}, \#, 0] \rightarrow \text{int} \rightarrow string\ \text{int}\ pair[1 \mid string, \text{int}, 0],\ 0;$

$string\_of\_\text{int}:\ 0,\ \text{int} \rightarrow string,\ 0;$

objectlist:

$ability : 0,\ obj[0 \mid \text{int}, string, \text{int}, \text{int}, string, 0] \rightarrow \text{int},\ 0;$

$category : 0,\ obj[0 \mid \text{int}, string, \text{int}, \text{int}, string, 0] \rightarrow \text{int},\ 0;$

$clearObjectList : 0,\ objectlist[0 \mid obj[0 \mid \text{int}, string, \text{int}, \text{int}, string, 0], \#, 0] \rightarrow objectlist[0 \mid obj[0 \mid \text{int}, string, \text{int}, \text{int}, string, 0], \#, 0],\ 0;$

$concatObjectList : 1,\ objectlist[0 \mid obj[0 \mid \text{int}, string, \text{int}, \text{int}, string, 0], \#, 0] \rightarrow \text{int} \rightarrow \text{int} \rightarrow string \rightarrow string,\ 1;$

$copyObjectList : 0,\ objectlist[0 \mid obj[0 \mid \text{int}, string, \text{int}, \text{int}, string, 0], \#, 1] \rightarrow objectlist[0 \mid obj[0 \mid \text{int}, string, \text{int}, \text{int}, string, 0], \#, 0] \rightarrow objectlist[0 \mid obj[0 \mid \text{int}, string, \text{int}, \text{int}, string, 0], \#, 0],\ 0;$

$findObject : 0,\ objectlist[0 \mid obj[0 \mid \text{int}, string, \text{int}, \text{int}, string, 0], \#, 0] \rightarrow \text{int} \rightarrow \text{int} \rightarrow obj[0 \mid \text{int}, string, \text{int}, \text{int}, string, 0],\ 0;$

$objElementAt : 1,\ objectlist[0 \mid obj[0 \mid \text{int}, string, \text{int}, \text{int}, string, 0], \#, 0] \rightarrow \text{int} \rightarrow obj[0 \mid \text{int}, string, \text{int}, \text{int}, string, 0],\ 0;$

$objectListLength : 0,\ objectlist[0 \mid obj[0 \mid \text{int}, string, \text{int}, \text{int}, string, 0], \#, 0] \rightarrow \text{int},\ 0;$

$oid : 0,\ obj[0 \mid \text{int}, string, \text{int}, \text{int}, string, 0] \rightarrow \text{int},\ 0;$

$oname : 0,\ obj[0 \mid \text{int}, string, \text{int}, \text{int}, string, 0] \rightarrow string,\ 0;$

$owner : 0,\ obj[0 \mid \text{int}, string, \text{int}, \text{int}, string, 0] \rightarrow string,\ 0;$

$removeObject : 0,\ objectlist[0 \mid obj[0 \mid \text{int}, string, \text{int}, \text{int}, string, 0], \#, 0] \rightarrow \text{int} \rightarrow objectlist[0 \mid obj[0 \mid \text{int}, string, \text{int}, \text{int}, string, 0], \#, 0],\ 0;$

$showObjectList : 0,\ objectlist[0 \mid obj[0 \mid \text{int}, string, \text{int}, \text{int}, string, 0], \#, 0] \rightarrow \text{int} \rightarrow \text{int} \rightarrow string \rightarrow string,\ 0;$

$showSortedObject : 0,\ objectlist[0 \mid obj[0 \mid \text{int}, string, \text{int}, \text{int}, string, 0], \#, 0] \rightarrow \text{int} \rightarrow \text{int} \rightarrow string \rightarrow string,\ 0;$

$string\_of\_\text{int} : 0,\ \text{int} \rightarrow string,\ 0;$

cardlist:

$attAbility\ :\ 0,\ card[0 \mid \text{int}, string, string, \text{int}, \text{int}, \text{int}, string, 0] \rightarrow \text{int},\ 0;$

$cardElementAt:\ 0,\ cardlist[0 \mid card[0 \mid \text{int}, string, string, \text{int}, \text{int}, \text{int}, string, 0], \#, 0]$
$\rightarrow \text{int} \rightarrow card[0 \mid \text{int}, string, string, \text{int}, \text{int}, \text{int}, string, 0],\ 0;$

$cardListLength:\ 0,\ cardlist[0 \mid card[0 \mid \text{int}, string, string, \text{int}, \text{int}, \text{int}, string, 0], \#, 0]$
$\rightarrow \text{int},\ 0;$

$cardOwner\ :\ 0,\ card[0 \mid \text{int}, string, string, \text{int}, \text{int}, \text{int}, string, 0] \rightarrow string,\ 0;$

$cid\ :\ 0,\ card[0 \mid \text{int}, string, string, \text{int}, \text{int}, \text{int}, string, 0] \rightarrow \text{int},\ 0;$

$clearCardList:\ 0,\ cardlist[0 \mid card[0 \mid \text{int}, string, string, \text{int}, \text{int}, \text{int}, string, 0], \#, 0]$
$\rightarrow cardlist[0 \mid card[0 \mid \text{int}, string, string, \text{int}, \text{int}, \text{int}, string, 0], \#, 0],\ 0;$

$cname\ :\ 0,\ card[0 \mid \text{int}, string, string, \text{int}, \text{int}, \text{int}, string, 0] \rightarrow string,\ 0;$

$concatCardList:\ 0,\ cardlist[0 \mid card[0 \mid \text{int}, string, string, \text{int}, \text{int}, \text{int}, string, 0], \#, 0]$
$\rightarrow \text{int} \rightarrow \text{int} \rightarrow string \rightarrow string,\ 0;$

$condition\ :\ 0,\ card[0 \mid \text{int}, string, string, \text{int}, \text{int}, \text{int}, string, 0] \rightarrow \text{int},\ 0;$

$copyCardList:\ 0,\ cardlist[0 \mid card[0 \mid \text{int}, string, string, \text{int}, \text{int}, \text{int}, string, 0], \#, 1]$
$\rightarrow cardlist[0 \mid card[0 \mid \text{int}, string, string, \text{int}, \text{int}, \text{int}, string, 0], \#, 0]$
$\rightarrow cardlist[0 \mid card[0 \mid \text{int}, string, string, \text{int}, \text{int}, \text{int}, string, 0], \#, 0],\ 0;$

$defAbility\ :\ 0,\ card[0 \mid \text{int}, string, string, \text{int}, \text{int}, \text{int}, string, 0] \rightarrow \text{int},\ 0;$

$figure\ :\ 0,\ card[0 \mid \text{int}, string, string, \text{int}, \text{int}, \text{int}, string, 0] \rightarrow string,\ 0;$

$findCard\ :\ 0,\ cardlist[0 \mid card[0 \mid \text{int}, string, string, \text{int}, \text{int}, \text{int}, string, 0], \#, 0]$
$\rightarrow \text{int} \rightarrow \text{int} \rightarrow \text{int}$
$\rightarrow card[0 \mid \text{int}, string, string, \text{int}, \text{int}, \text{int}, string, 0],\ 0;$

$removeCard\ :\ 0,\ cardlist[0 \mid card[0 \mid \text{int}, string, string, \text{int}, \text{int}, \text{int}, string, 0], \#, 0] \rightarrow \text{int}$
$\rightarrow cardlist[0 \mid card[0 \mid \text{int}, string, string, \text{int}, \text{int}, \text{int}, string, 0], \#, 0],\ 0;$

$showCardList:\ 0,\ cardlist[0 \mid card[0 \mid \text{int}, string, string, \text{int}, \text{int}, \text{int}, string, 0], \#, 0]$
$\rightarrow \text{int} \rightarrow \text{int} \rightarrow string \rightarrow string,\ 0;$

$string\_of\_\text{int}:\ 0,\ \text{int} \rightarrow string,\ 0;$

We have noticed that most of these functions require no extra heap cell before executing, and do not return any heap cell after executing. So the heap space this kind of functions need relies on the size of inputs. For example, the **sizeof** function on **stringpairlist**, it doesn't need any more heap cells during its execution except for those allocated at the beginning to hold the input **stringpairlist**. So the number of heap cells this function needs is equal to the length of the input list. It is the same to other functions which have same annotation types as **sizeof**.

There is another case shown above:

$$\begin{aligned} &\mathit{showStringPairList}:\ 1,\ \mathit{stringpairlist}[0 \mid \mathit{string}, \mathit{string}, \#, 0] \\ &\qquad \to \text{int} \to \text{int} \to \mathit{string}\ \to \mathit{string},\ 1; \end{aligned} \tag{1}$$

$$\begin{aligned} &\mathit{ssElementAt}:\ 1,\ \mathit{stringpairlist}[0 \mid \mathit{string}, \mathit{string}, \#, 0] \to \text{int} \\ &\qquad \to \mathit{stringstringpair}[1 \mid \mathit{string}, \mathit{string}, 0],\ 0; \end{aligned} \tag{2}$$

The type for **showStringPairList** says that it needs at least one free heap cell before starting and leaves at least one free cell on the heap when it returns. This heap cell is used up in the call to **ssElementAt**, but is recovered when we do the destructive match, and can be used again the next time we call **ssElementAt**. When we eventually return from **showStringPairList**, this heap cell will be free for other functions to use, so we get the return type string, 1. Moreover, **showStringPairList** is the only function invoking **ssElementAt**. So the overall consumption of these two functions is also equal to the size of inputs.

The third LFD type shown above is:

$$\begin{aligned} &\mathit{copyStringPairList}:\ 0,\ \mathit{stringpairlist}[0 \mid \mathit{string}, \mathit{string}, \#, 1] \\ &\qquad \to \mathit{stringpairlist}[0 \mid \mathit{string}, \mathit{string}, \#, 0] \\ &\qquad \to \mathit{stringpairlist}[0 \mid \mathit{string}, \mathit{string}, \#, 0],\ 0; \end{aligned}$$

which means that executing **copyStringPairList** may allocate a number of extra heap cells equal to the length of the input list, one free heap cell per **Cons**-node of the input list. So in this case, the heap space consumption of the function is the double of the input list's length, which includes the space to hold the input list at the very beginning.

Besides for what we have discussed above, another heap space consumption is the heap space required for those codes involving objects. In section 3.3.2.1, we have presented several solutions to this problem. Here we point out again just because it also needs to be taken account of when we want to get an accurate result of the overall heap space consumption of the program.

Furthermore, since different functions might be used in different circumstances, which implies that their inputs might vary, so it's not possible for us, in current condition, to give an exact number of heap cells we require for our program. One of the solutions to it is that we can impose more constraints on the inputs of some functions to make possible that we can expect which function will have what kind of input and how large of the input. In this case, providing an accurate result of heap space consumption becomes possible. This idea also suggests a possibility of our future work.

Chapter 4 Conclusion

4.1 Project Summary

This is a case study project of developing an on-line game application in the Camelot programming language which has been compiled to run on an emulator for a Java-enabled mobile phone. Three major issues have been considered in the implementation:

- The system deploys a distributed architecture. Some services of the system are provided by the Network. So peers in a virtual wireless environment need to broadcast their queries to locate the service they require.
- The system is highly mobile and has a P2P nature. **Room** is the most important mobile object of the system. Copies of it move from one peer to another peer. This "move" is actually a **clone** action, which is achieved through the direct Peer-to-Peer communication.
- The system provides environment actions which are local and synchronous. Room is the critical section, each of which has a Token for exclusive actions on it. Only the player owning the Token has the right to access resources and perform the action he requested. Other players must wait and their requests will be enqueued.

Programming a mobile device is very different from programming a general-purpose workstation because of the resource limit on the former. For this reason, developers can benefit from using programming languages, like Camelot, to get precise guarantees of resource consumption which are inferred directly from the application code. Camelot is an OCaml-like functional programming language which compiles to a high-level analogue of Java byte code named Grail. Camelot is equipped with a mechanism which enables programmers have precise control on

memory especially heap space usage. Moreover, although Camelot has been used in a wide range, most implementations of it so far are small and aim for particular problems. The implementation of this project faces to a set of various problems. Thus, it is a good opportunity to help test and improve the compiler of Camelot. Some improvements have been made to the compiler during this period are:

- Added new datatypes;
- Fixed a parsing problem related to the class name in Camelot;
- Fixed a problem caused by class inheritance;
- Allowed datatypes contain objects;
- Added the **isnull** construct;
- Released the constraint on variable names;
- Enabled the compiler to work on Windows system.

The Goal of the Mobile Resource Guarantees (MRG) project is to endow mobile code with certificate of bounded resource consumption. These certifications contain a claim of resource usage together with a proof of the claim. The proof is expressed in the program logic we mentioned in section 3.2.2, and the claim of resource usage can be obtained by the use of lfd_infer, an implementation of static prediction of heap space usage for first-order functional programs. This project uses this tool to infer the heap consumption of pure Camelot codes in our game. However, lfd_infer has its shortcomings which partly make effects on the accuracy of the result. These shortcomings are:

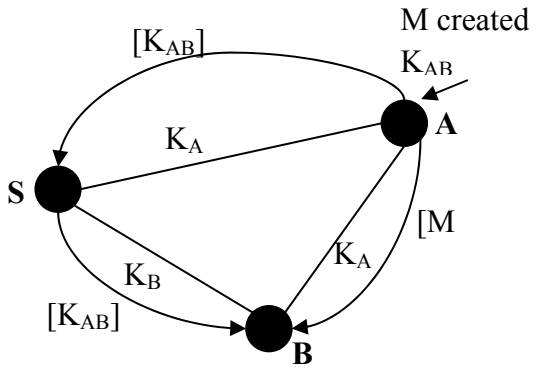
- Functional objects as allowed in Camelot are currently not accounted for. Therefore, we only perform inference on the Camelot code without objects.
- The term “memory leak” is not accurate in lfd_infer. We have to make some artificial things in order to pass the inference, and these introduce the accuracy of the result about the heap usage.
- Sometimes programmers are required to adjust objective functions created by lfd_infer themselves so as to get reasonable solutions to their functions.

4.2 Further Work

As for the further work, not only on the game itself, but also on the analysis on resource consumption of the game, major suggestions are:

(1) Game Interface. The user interface of the current game is quite simple. For a game which hopes to attract its players, the user interface is a critical part. J2ME provides a specific user interface package which can ease our work in the future.

(2) Game Security. [17] proposes a Wide Mouthed Frog algorithm for this problem based on the SSL protocol.



Protocol Narration:
1. A→S: $A,K_{AS}[B,K_{AB}]$
2. S→B: $K_{BS}[A,K_{AB}]$
3. A→B: $K_{AB}[M]$

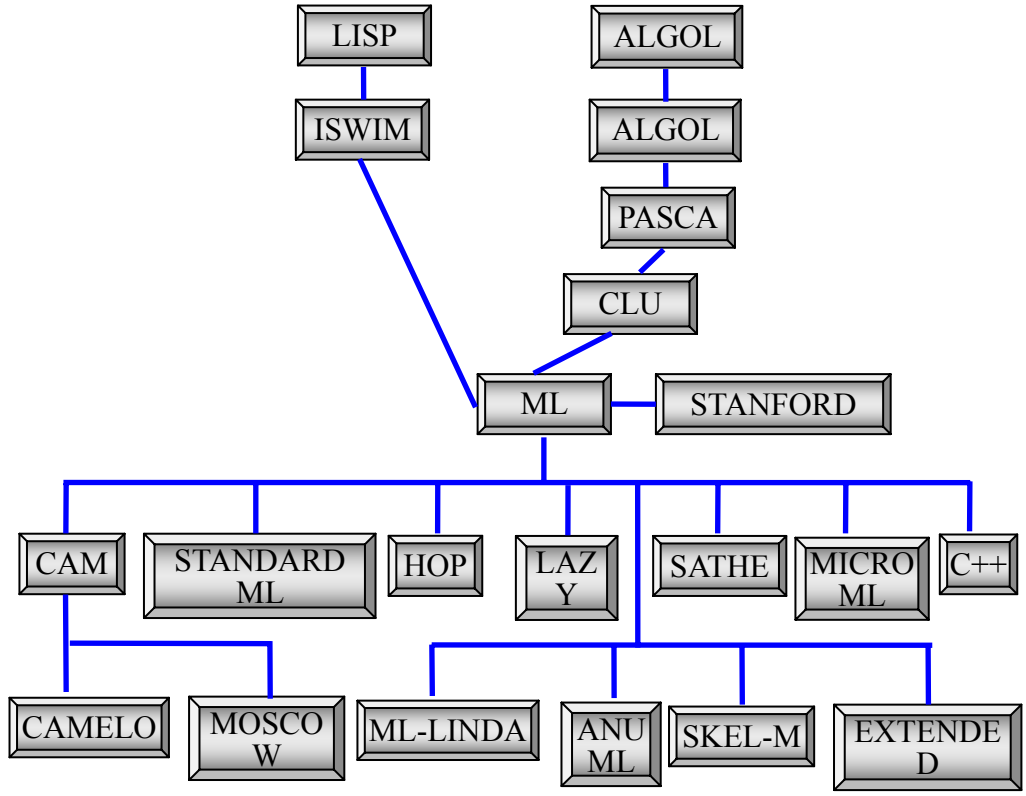
Figure 4.1 the Wide Mouthed Frog Algorithm [17]

The figure above shows the details of this algorithm to establish a secure connection between peer A and peer B. First peer A creates the key K_{AB} he wants to use to talk with B, then sends this information through a secure connection to the Server using his session key KA_S. The Server forwards the information directly to B using their session key K_{BS}. Now that B has the session key K_{AB}, A can encrypt message M using private session key K_{AB} and send it to B. Now that peer A and B have established their private session key they can have secure communication without relaying to the Server anymore.

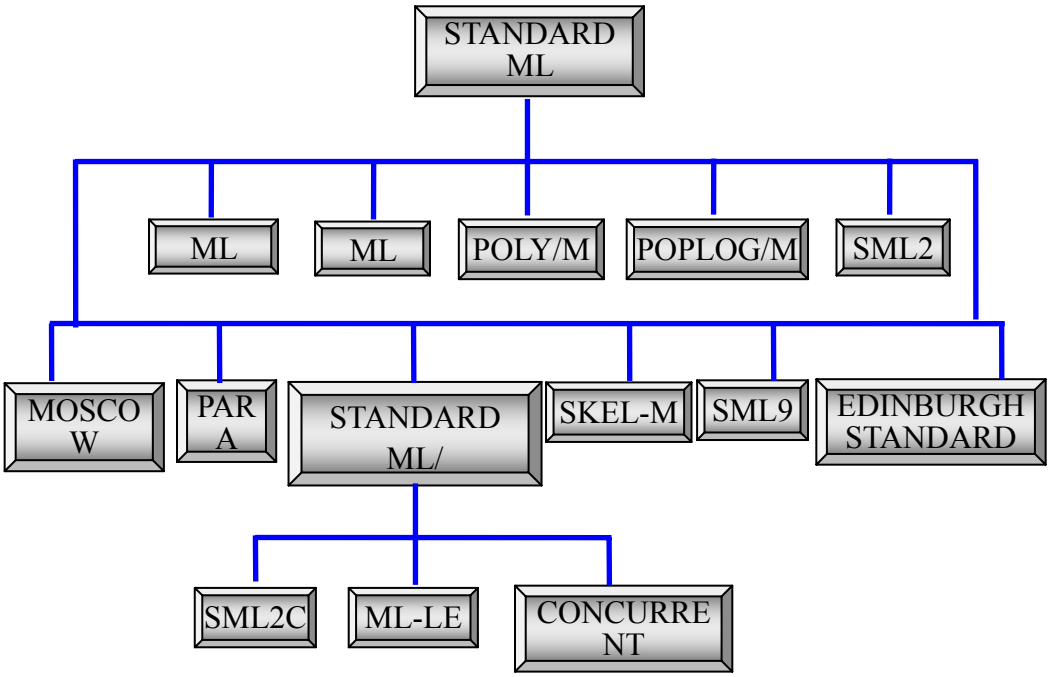
(3) The program logic. The program logic is used to assert resource-related properties of Java bytecode programs. Currently, we can get the inference of heap space usage for our program (those pure Camelot codes in the program), which forms part of the certification. It is nevertheless worthwhile to experiment with

another part of the certification, the proof, at the bytecode level. Moreover, using the ‘raw’ proof rules at the bytecode level will enable proofs to be given that are not possible via the high-level type system - for example, it may be possible to prove tighter resource bounds this way. This experimentation work can be carried out using the Isabelle interactive theorem prover.

Appendix A ML Family Evolution Tree



Standard ML has also influenced following languages



Appendix B The Syntax of Camelot

The terms **tycoon**, **con**, **fname**, and **var** refer to ***type constructors***, ***constructors***, ***function names*** and ***variable names*** respectively. The term **tyvar** refers to a ***type variable***, which is a name beginning with a single quote.

$$
\begin{array}{lcl}
program & :: = & \langle typedecseq\rangle\langle valdecseq\rangle\langle funimplseq\rangle \\
typedecseq & :: = & typedec\ \langle typedecseq\rangle \\
typesec & :: = & \text{type}\ \langle (ty\ \text{var}_1 \ldots ty\ \text{var}_n)\rangle\ tycon\ =\ conbind \\
conbind & :: = & con\ \langle of\ ty_1 * \ldots * ty_n\rangle\langle |\ conbind\rangle \\
 & & |!con\ \langle |\ conbind\rangle \\
ty & :: = & \text{int} \\
 & & \text{char} \\
 & & \text{bool} \\
 & & \text{float} \\
 & & \text{string} \\
 & & \text{unit} \\
 & & \text{byte} \\
 & & \text{long} \\
 & & tycar \\
 & & ty\ \text{array} \\
 & & tyseq\ tycon \\
Ty & :: = & ty \\
 & & Ty_n \to \ldots \to Ty_1 \to ty\ \ (higer-order\ type) \\
valdecseq & :: = & valdec\ \langle valdecseq\rangle \\
valdec & :: = & \text{val}\ var\ :\ ty \\
 & & \text{val}\ fname\ :\ Ty \\
funimplseq & :: = & funimpl\langle funimplseq\rangle \\
funimpl & :: = & \text{let}\ \langle \text{rec}\rangle\ fundecseq
\end{array}
$$

fundecseq	:: =	*fundec* ⟨and *fundecseq*⟩
fundec	:: =	*fname* var *seq* = exp *r*
exp *r*	:: =	*const*
		var
		fname
		fname exp $r_1 \ldots$ exp r_n
		if exp *r* then exp *r* else exp *r*
		let *pat* = exp *r* in exp *r*
		match exp *r* with *match*
		free var
		(exp *r*)
match	:: =	*mrule* ⟨| *match*⟩
mrule	:: =	*con*⟨($pat_1, \ldots, pat_n$)⟩ ⇒ exp *r*
		con ⟨($pat_1, \ldots, pat_n$)⟩ @ *pat* ⇒ exp *r*
pat	:: =	*vra*
		_

In some contexts the symbol _ can be used instead of a variable name. This feature can be used to discard unwanted values.

Appendix C Camelot's Built-in Funtions

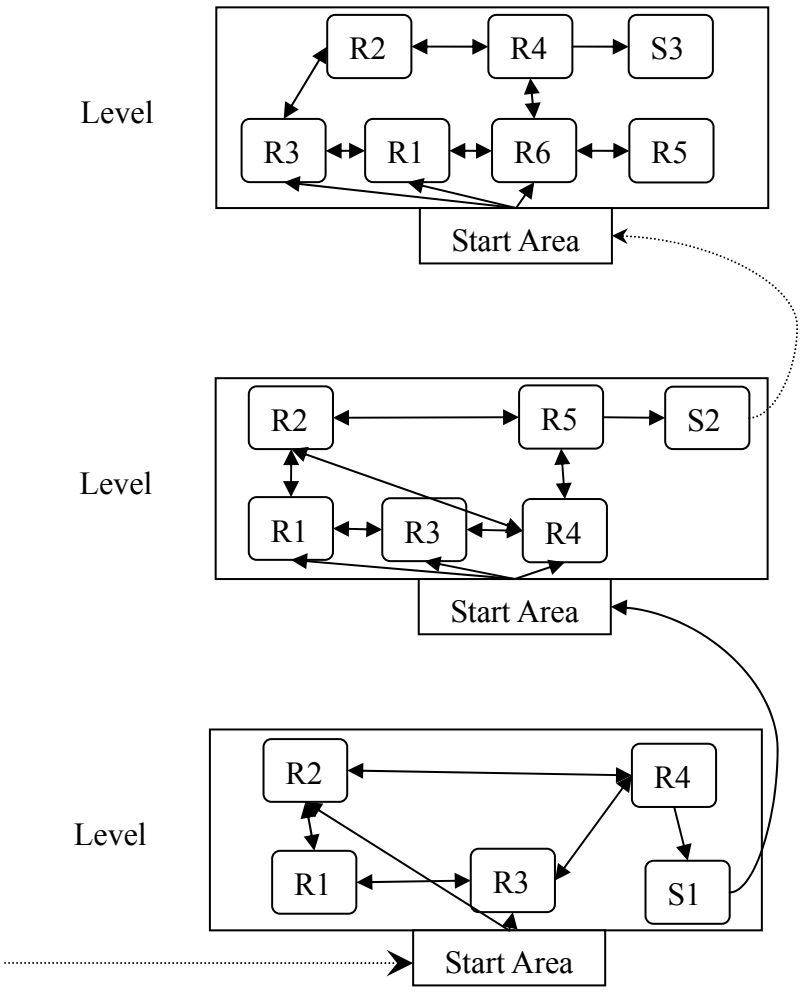
There are some built-in functions provide by Camelot. The names of most of these are self-explanatory.

int_of_float: float -> int
float_of_int : int -> float
int_of_string: string -> int
string_of_int: int -> string
float_of_string: string -> float
string_of_float: float $\rightarrow$ string
print_int: int $\rightarrow$ unit
print_int_newline: int $\rightarrow$ unit
print_float: float $\rightarrow$ unit
print_float_newline: float $\rightarrow$ unit
print_char: char $\rightarrow$ unit
print_char_newline: char $\rightarrow$ unit
print_string: string $\rightarrow$ unit
print_string_newline: string $\rightarrow$ unit
print_newline: unit $\rightarrow$ unit
same_string: string $\rightarrow$ string $\rightarrow$ bool

(* Array operations *)
empty: int $\rightarrow \alpha \rightarrow \alpha$ array
get: α array $\rightarrow$ int $\rightarrow \alpha$
set: α array $\rightarrow$ int $\rightarrow$ unit
arraylength: α array $\rightarrow$ int

Appendix D.1 The Map of The Game

The game consists of a series of game levels. Each level is composed of a start area, a certain number of standard rooms, and a special room which can hold only one player at a time.



Appendix D.2 Actions in The Game

This game allows actions:

- **Observe**: observe the current room's information;
- **Use**: use personal objects, points of some parameters can be increased;
- **Take Room Object**: take an object from the current room;
- **Take Room Weapon**: take an weapon from the current room;
- **Talk**: talk to a player inside the same room;
- **Fight**: fight with a player inside the same room;
- **Change Room**: leave the current room and enter another room;
- **Check My Weapons**, **Check My Objects**, **Check My Points**: check the player's information.

These actions can be divided into following three categories:

Self Actions: Local & Asynchronous

Environmental Action: Local & Synchronous

Interactive Action: P2P & Asynchronous

Action		Local	P2P	Synchronous	Asynchronous
To observe		X			X
To use	Personal obj.	X			X
To take	Room Objects	X		X	
	Room Weapons	X		X	
To Check	My Objects	X			X
	My Weapons	X			X
	My Points	X			X
To talk	Players		X		X
To fight	Players		X		X
To change room		X			X

Appendix D.3 The Categories of Objects

Objects of the game are divided into several categories based on the parameter they have effect on.

Categories	*Parameters affected*
Category 1	Health Points
Category 2	Strength
Category 3	Agility
Category 4	Cash
Category 5	Health Points & Strength
Category 6	Strength & Agility
Category 7	Health & Agility

Appendix D.4 Self-Defined Datatypes

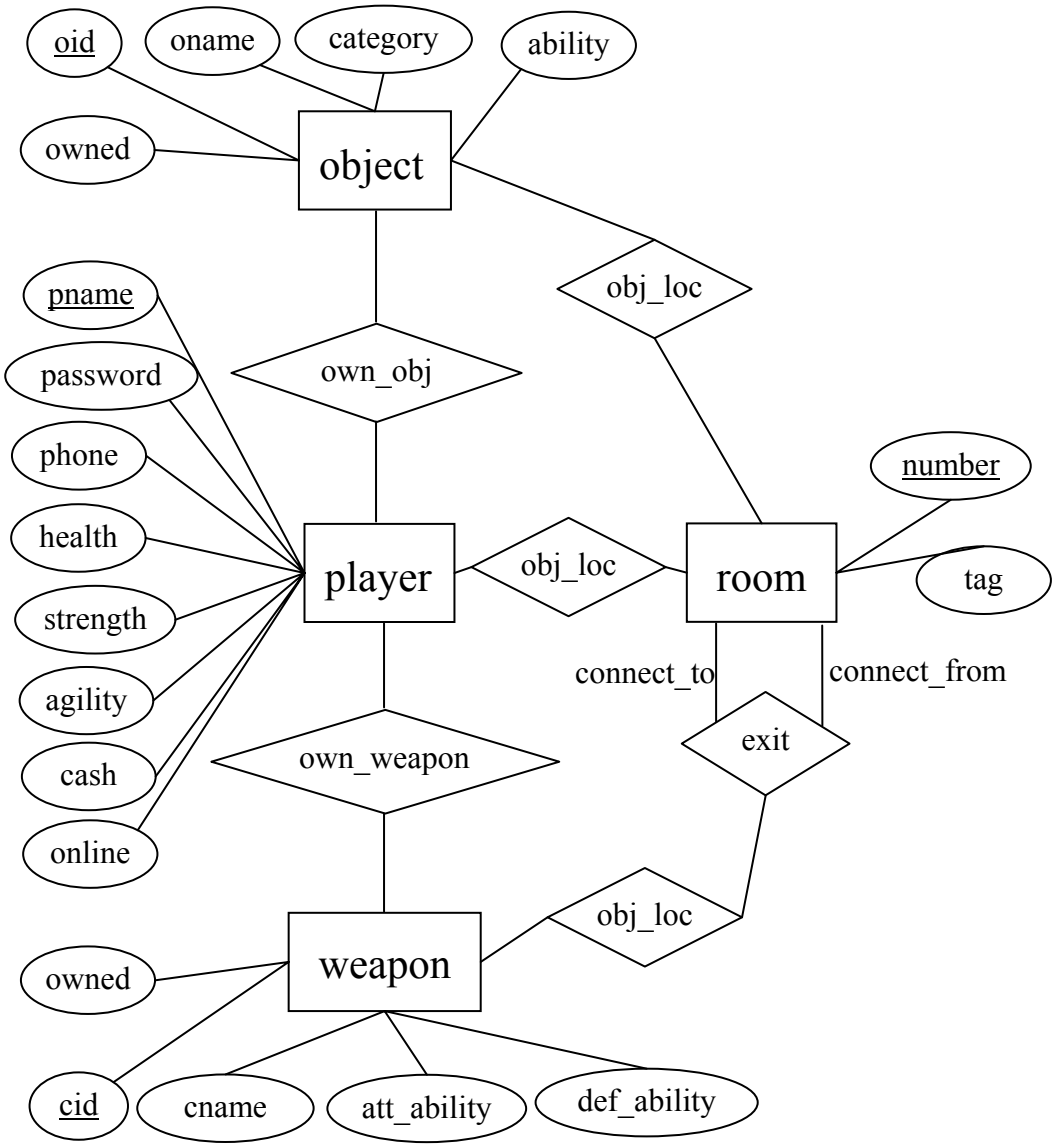
```
type objectlist = !Nil | Cons of obj * objectlist
type cardlist = !Nil2 | Cons2 of card * cardlist
type string int list = !Nil3 | Cons3 of string * int * string int list
type stringpairlist = !Nil4 | Cons4 of string * string * stringpairlist
type string int pair = Pair1 of string * int
type stringstringpair = Pair2 of string * string
type stringlist = !Nil5 | Cons5 of string * stringlist

type obj = Object of int* string * int* int* string
type card = Card of int* string * string * int* int* int* string
```

Note that the object element and the weapon element of the game are defined as datatypes, rather than classes.

Appendix D.5 E-R Diagram Of the Database

This is the E-R diagram for the database used on the Server side.



Appendix E Pure Camelot Code in the Game

Declarations of functions here are the very original ones, so they still have problems mentioned in chapter 3. Refer to chapter 3 to find how to modify them.

Stringpairlist.cmlt

```
type stringpairlist = !Nil4 | Cons4 of string * string * stringpairlist
type stringstringpair = !None2 | Pair2 of string*string
type stringlist = !Nil5 | Cons5 of string * stringlist

let ssElementAt (l : stringpairlist) k =
        match l with Nil4 -> None2
            | Cons4 (a,b,t) -> if k > 0
                               then ssElementAt t (k-1)
                               else Pair2 (a,b)

let sizeof (l : stringpairlist) =
        match l with Nil4 -> 0
            | Cons4 (a,b,t) -> 1 + (sizeof t)

let rmPlayer (l : stringpairlist) (pname : string) : stringpairlist =
    match l with Nil4 -> Nil4
        | Cons4(a,b,t) -> if a = pname
                          then t
                          else rmPlayer t pname

let findNumber (l : stringpairlist) (pname : string) : string =
    match l with Nil4 -> ""
        | Cons4 (a,b,t) -> if a = pname
                           then b
                           else findNumber t pname

let copyStringPairList (l1: stringpairlist) (l2 : stringpairlist) =
    match l1 with Nil4 -> l2
        | Cons4 (a,b,t) -> let l2 = Cons4 (a,b,l2)
                           in copyStringPairList t l2
```

```
let showStringPairList (l : stringpairlist) (k : int) (n : int) (sink : string) : string =
        if k < n
        then begin
             match ssElementAt l k with
              Pair2 (key, value) →
                 let sink = sink ^ "player: " ^ key ^ " phone: " ^ value ^ "\n"
                 in showStringPairList l (k+1) n sink
             | None2 → sink
        end
        else sink
let getValue (l : stringpairlist) (key : string) =
             match l with Nil4 → " "
                        | Cons4 (a, b, t) → if a = key
                                            then b
                                            else getValue t key

let stringAt (l : stringlist) (k : int) =
          match l with Nil5 -> " "
                     | Cons5(h,t) → if k > 0
                                    then stringAt t (k-1)
                                    else h

let clearStringPairList (l : stringpairlist) =
          match l with Nil4 → ()
                     | Cons4 (a,b,t)@_ → clearStringPairList t
```

stringintlist.cmlt

```
type string int list = !Nil3 | Cons3 of string * int * string int list
type string int pair = !None1 | Pair1 of string * int

let siElementAt (l : string int list) k =
             match l with Nil3 → None1
                        | Cons3 (a,b,t) → if k > 0
                                          then siElementAt t (k-1)
                                          else Pair1(a,b)

let lengthof (l : string int list) =
             match l with Nil3 → 0
                        | Cons3 (a,b,t) → 1 + lengthof t
```

```
let removeFirst (l : string int list) =
        match l with Nil3 → Nil3
                   | Cons3 (a,b,t) → t

let findRoom (l : string int list) (direction : string) : int =
    match l with Nil3 → 0
               | Cons3 (a,b,t) → if a = direction
                                 then b
                                 else findRoom t direction

let concatStringIntList (l : string int list) (k : int) (n : int) (sin k : string) : string =
    if k < n
    then match siElementAt l k
       with Pair1(key, value)@_ →
                  let sin k = sin k ^ key ^ "|" ^ (string _of _int value) ^ "|"
                  in concatStringIntList l (k+1) n sin k
       | None1 → sin k
    else sin k

let copyStringIntList (l1 : string int list) (l2 : string int list) =
    match l1 with Nil3 → l2
            | Cons3 (a,b,t) → let l2 = Cons3(a,b,l2)
                              in copyStringIntList t l2

let showStringIntList (l : string int list) (k : int) (n : int) (sin k : string) : string =
        if k < n
        then begin
             match siElementAt l k with
              Pair1 (key, value) →
                let sin k = sin k ^ key ^ ": " ^ (string _of _int value) ^ "\n"
                in showStringIntList l (k+1) n sin k
             | None1 → sin k
        end
        else sin k

let clearStringIntList (l : string int list) =
                match l with Nil3 → ()
                           | Cons3 (a,b,t) → clearStringIntList t
```

cardlist.cmlt (At the very beginning, we used “card” to name “weapon”)

```
type cardlist = !Nil2 | Cons2 of weapon * weaponlist
type card = !NullWeapon | Weapon of int*string*string*int*int*int*string

let cid w = match w with Card (id,_,_,_,_,_,_) → id
                       | NullCard → 0

let cname w = match w with Card (_,name,_,_,_,_,_) → name
                         | NullCard → " "

let figure w = match w with Card (_,_,fpath,_,_,_,_) → fpath
                          | NullCard → " "

let attAbility w = match w with Weapon (_,_,_,attak,_,_,_) → attak
                              | NullWeapon → 0

let defAbility w = match w with Card (_,_,_,_,defence,_,_) → defence
                              | NullCard → 0

let condition w = match w with Card (_,_,_,_,_,cond,_) → cond
                             | NullCard → 0

let cardOwner w = match w with Card (_,_,_,_,_,_,owner) → owner
                             | NullCard → " "

let cardElementAt (l : cardlist) k =
        match l with Nil2 → NullCard
                   | Cons2 (h,t) → if k > 0
                                   then cardElementAt t (k-1)
                                   else h

let cardListLength (l : cardlist) =
        match l with Nil2 → 0
                   | Cons2 (h,t) → 1 + cardListLength t
```

```
let findCard (l : cardlist) (id : int) (k : int) (n : int) : card =
        if k < n
        then begin
             let weapon = cardElementAt l k
           in
             if id = (cid weapon)
             then weapon
             else findCard l id (k+1) n
        end
        else let weapon = NullCard
           in weapon
let concatCardList (l : cardlist) (k : int) (n : int) (sink : string) =
        if k < n
        then let cardIns = cardElementAt l k
         in let sink = sink ^ (string_of_int (cid cardIns)) ^ "|" ^ (cname cardIns) ^ "|"
                          ^ (figure cardIns) ^ "|" ^ (string_of_int (attAbility cardIns)) ^ "|"
                          ^ (string_of_int (defAbility cardIns)) ^ "|"
                          ^ (string_of_int (condition cardIns)) ^ "|"
         in concatCardList l (k+1) n sink
        else sink
let copyCardList (l1 : cardlist) (l2 : cardlist) =
        match l1 with Nil2 -> l2
                | Cons2 (a,t) -> let l2 = Cons2(a,l2)
                                 in copyCardList t l2

let showCardList (l : cardlist) (k : int) (n : int) (sink : string) : string =
            if k < n
            then begin
                   let cardIns = cardElementAt l k
                 in let sink = sink ^ (cname cardIns) ^ ": Attack Ability -> "
                                ^ (string_of_int (attAbility cardIns))
                                ^ "; Defend Ability -> "
                                ^ (string_of_int (defAbility cardIns)) ^ "\n"
                 in showCardList l (k+1) n sink
               end
               else sink

let clearCardList (l : cardlist) = match l with Nil2 -> ()
                                 | Cons2 (h,t)@_ -> clearCardList t
```

```
let removeCard (l : cardlist) (id : int) : cardlist =
        match l with Nil2 -> Nil2
              | Cons2 (h,t)@_ →
                        if cid h = id
                        then Cons2(NullCard, t)
                                  else Cons2 (h, removeCard t id)
```

objectlist.cmlt

```
type objectlist = !Nil | Cons of obj * objectlist
type obj = !NullObject | Object of int*string*int*int*string

let oid o = match o with Object (id,_,_,_,_) → id
                         | NullObject → 0

let oname o = match o with Object (_,name,_,_,_) → name
                           | NullObject → " "

let category o = match o with Object (_,_,cat,_,_) → cat
                              | NullObject → 0

let ability o = match o with Object (_,_,_,abi,_) → abi
                             | NullObject → 0

let owner o = match o with Object (_,_,_,_,own) → own
                           | NullObject → " "

let objElementAt (l : objectlist) k =
              match l with Nil → NullObject
                           | Cons (h,t) → if k > 0
                                          then objElementAt t (k-1)
                                          else h

let objectListLength (l : objectlist) =
                    match l with Nil → 0
                                 | Cons (h,t) → 1 + objectListLength t

let clearObjectList (l : objectlist) =
                    match l with Nil -> Nil
                                 | Cons (h,t) → clearObjectList t
```

```
let removeObject (l : objectlist) (id : int) : objectlist =
    match l with Nil → Nil
        | Cons (h,t)@_ →
            if oid h = id
            then let _ = clearObject h in t
            else Cons(h, removeObject t id)

let findObject (l : objectlist) (id : int) (k : int) (n : int) : obj =
    if k < n
    then
        let objxt = objElementAt l k
      in
        if id = (oid objxt)
        then objxt
        else findObject l id (k+1) n
    else let objxt = NullObject
      in objxt

let concatObjectList (l : objectlist) (k : int) (n : int) (sin k : string) =
    if k < n
    then let objxtIns = objElementAt l k
     in let sin k = sin k ^ (string_of_int (oid objxtIns)) ^ "|"
                ^ (oname objxtIns) ^ "|" ^ (string_of_int (category objxtIns)) ^ "|"
                ^ (string_of_int (ability objxtIns)) ^ "|"
     in concatObjectList l (k+1) n sin k
    else sin k

let copyObjectList (l1 : objectlist) (l2 : objectlist) =
    match l1 with Nil → l2
                | Cons (a,t) → let l2 = Cons(a,l2)
                               in copyObjectList t l2
```

```
let showSortedObject (l : objectlist) (k : int) (n : int) (sink : string) =
        if k < n
        then let objxtIns = objElementAt l k
         in let sink = sink ^ "OBJECTS "
                    ^ (oname objxtIns) ^ ": category -> "
                    ^ (string_of_int (category objxtIns))
                    ^ " ability -> " ^ (string_of_int (ability objxtIns))
                    ^ "\n\n"
         in showSortedObject l (k+1) n sink
        else sink
let showObjectList (l : objectlist) (k : int) (n : int) (sink : string) : string =
        if k < n
        then begin
               let objxtIns = objElementAt l k
           in let sink = sink ^ (oname objxtIns) ^ ": category -> "
                       ^ (string_of_int (category objxtIns))
                       ^ "; Ability -> " ^ (string_of_int (ability objxtIns)) ^ "\n"
           in showObjectList l (k+1) n sink
          end
          else sink
```

Bibliography

[1] Surupa Biswas, Matthew Simpson, Rajeev Barua. Memory Overflow Protection for Embedded Systems using Run-time Checks, Reuse and Compression. *CASES"04*: 280-291, September 22-25, 2004

[2] Wind River. High Availability Design for Embedded Systems. Http://www.windriver.com/whitepapers/high_availability_design.html

[3] George V.Neville-Neil. Programming Without A Net. *ACM Queue: Tomorrow's Computing Today*, 1(2): 16-23, April 2003

[4] Motorola. M68000 User's Manual. *Prentice Hall, Englewood Cliffs, NJ*

[5] Intel i960Sx 32-bit Microprocessor. *Intel Corporation.* http://www.intel.com/design/i960/documentation/docs_sx.htm.

[6] MSP430 Ultra-Low-Power MSUs. *Texas Instruments*, 2004. http://focus.ti.com/lit/ml/slab034g/slab034g.pdf

[7] Michael Durrant. Running Linux on low cost, low power MMU-less processors. August 2000. http://www.linuxdevices.com/articles/AT6245686197.html

[8] George C. Necula, Peter Lee. Safe, Untrusted Agents using Proof-Carrying Code. In *LNCS 1419: Special Issue on Mobile Agent Security*. Springer, 1998.

[9] George C. Necula, Peter Lee. Safe Kernel extensions without run-time checking. In *Proc. 2nd USENIX Symp. On Operating System Design and Impl*., pages 229-243, 1996

[10] George C. Necula. Proof-Carrying Code. In *Proc. 24th ACM Symp. On Principles of Prog. Lang*., pages 106-119, New York, Jan. 1997. ACM Press.

[11] Andrew W. Appel. Foundational Proof-Carrying Code. *In LICS '01, 16th Annual IEEE Symposium Logic in Computer Science*, pages 1-10, June 16, 2001.

[12] David Aspinall, Stephen Gilmore, Martin Hofmann, Donald Sannella, and Ian Stark. Mobile Resource Guarantees for Smart Devices. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices: Proceedings of the International Workshop CASSIS 2004*, Springer-Verlag, No. 3362, pages 1-26, 2005.

[13] MRG Final Report. 30th June 2005. http://groups.inf.ed.ac.uk/mrg/project-info/final-report.pdf.

[14] Kenneth MacKenzie and Nicholas Wolverson. **Camelot and Grail: resource-aware functional programming on the JVM**. In *Trends in Functional Programing*, Intellect,Vol. 4, pages 29-46, 2004.

[15] Lennart Beringer, Kenneth MacKenzie and Ian Stark. **Grail: a Functional Form for Imperative Mobile Code**. In *Foundations of Global Computing: Proceedings of the 2nd EATCS Workshop*, Elsevier, No. 85.1, June 2003.

[16] MRG Report: Information Society Technologies (IST) Programme, Annex1 – "Description of Work". October 2004. http://groups.inf.ed.ac.uk/mrg/project-info/contract-new.pdf

[17] C. Caragiuli, D. Piazza, I.Mura, C. Chesta, G. Previti and M. Di Florio. **D24 Specification in UML of Case studies**. In *Design Environments for Global ApplicationS,* DEGAS IST-2001-32072, pages 6-12, 17-21, 34-61, 31 December 2002.

[18] Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/wiki/Main_Page

[19] Xavier Leroy. The Objective Caml system release 3.08 Document and User's manual. July 13, 2004. http://caml.inria.fr/pub/docs/manual-ocaml/index.html

[20] S. Gilmore, K.MacKenzie, and N. Wolverson. Extending Resource-Bounded Functional Programming Languages with mutable state and concurrency. In *Parallel and Distributed Computing Practices*, pages 1 – 19, 2005.

[21] Hans-Wolfgang Loidl and Kenneth MacKenzie. A Gentle Introduction to Camelot. LFCS, Univ of Edinburgh & Inst for Informatics, LMU Univ Munich, pages 1 – 36, September 2004.

[22] Kenneth MacKenzie. An Overview of Camelot. LFCS, Univ of Edinburgh, pages 1 – 11, 3rd July 2003.

[22] Lennart Beringer, Martin Hofmann, Alberto Momigliano and Olha Shkaravska. Towards certificate generation for linear heap consumption. In *Proceedings of ICALP/LICS Workshop on Logics for Resources, Processes, and Programs (LRPP2004)*, pages 1 – 12, July 2004.

[23] M. Hofmann and S. Jost. Static Prediction of Heap Space Usage for First-order Functional Programs. In *Proceedings of the 30th ACM Symposium on Principles of Programming Languages*, ACM Press, New York, Vol. 38, No. 1, pages 185-197, January 2003.

[24] Srinath, L. S. Linear Programming: Principles and Applications (2nd Edition) book

[25] David Aspinall, Lennart Beringer, Martin Hofmann, Hans-Wolfgang Loidl and Alberto Momigliano. A program Logic for Resource Verification. In *Proceedings of 17th International Conference on Theorem Proving in Higher Order Logics (TPHOLs2004)*, Springer-Verlag LNCS, Heidelberg, Vol. 3223, pages 34-49, September 2004.

[26] M. Hofmann and S. Jost. Static Prediction of Heap Space Usage for First-order Functional Programs. In *Proceedings of the 30th ACM Symposium on Principles of Programming Languages*, ACM Press, New York, Vol. 38, No. 1, pages 185-197, January 2003.

[27] Lennart Beringer, Martin Hofmann, Alberto Momigliano and Olha Shkaravska. Towards Certificate Generation For Linear Heap Consumption. In *Proceedings of ICALP/LICS Workshop on Logics for Resources, Processes, and Programs (LRPP2004)*, July 2004.

[28] Steffen Jost. lfd_infer: an Implementation of a Static Inference on Heap Space Usage. Inst for Informatics, LMU Univ Munich

[29] Nicholas Wolverson and Kenneth MacKenzie. O'Camelot: Adding Objects to

a Resource Aware Functional Language. In *Trends in Functional Programing*, Intellect, Vol. 4, pages 47-62, 2004.