Applicative Notions in ML-like Programs

Budi Halim Ling

Master of Philosophy
Department of Computer Science
University of Edinburgh
1998

Abstract

Pure functional languages are expressive tools for writing modular and reliable code. State in programming languages is a useful tool for programming dynamic systems. However, their combination yields programming languages that are difficult to model and to reason about.

There have been ongoing attempts to find subsets of the whole languages which have good properties; in particular subsets where the programs are more modular and the side effects are controlled. The existing studies are: interference control, typing with side-effects information, and linear logic based languages. This thesis presents a new classification for a paradigm called constant program throughout a computational invariant. A program is called constant throughout an invariant R if its input-output behaviour is constant over any variations of state that satisfy the invariant R. Hence such a program behaves in an applicative way when it is executed in a context that satisfies the invariant R.

The language of discussion is a pure ML fragment augmented with `ref`, `:=`, and `!`. Programs with side effects are modelled in terms of sets, functions, and the side effect monad. Computational invariants are modelled in terms of transition systems. The notion of being constant throughout an invariant requires the notion of indistinguishability throughout an invariant and we define the latter using logical relation technique. We give two definitions of each of them: the first one can be used for reasoning about programs with flat stores adequately. The second one is more sensitive to the behaviour of `ref` and gives a better account of constant programs with dynamic allocations.

Our results are: indistinguishability throughout an invariant R is an equivalence relation over elements that are constant throughout R, and the notion of being constant throughout an invariant is preserved under function application. On the practical side we present some substantial ML examples which use references and side effects but externally behave in a constant way, together with the proofs that they are classified as being constant. These are evidences that our notions are useful concepts in the practise of writing modular programs.

Acknowledgements

I am indebted to my first supervisor, Michael Fourman, for providing me with valuable support and advise. He also encouraged me to study imperative aspects of ML.

Thanks to my second supervisor, Stuart Anderson, for giving a lot of encouragement, particularly in the final phase of writing up.

Thanks to people in LFCS in general, particularly Stephen Gilmore, Stefan Kahrs, John Longley, Dave Matthews, Alex Simpson, and Paul Steckler. They are keen to answer my questions about ML and general technical issues.

Thanks to people who have helped me in understanding the issues in my thesis; they are are: Andrzej Filinski, Guy McCusker, Peter O'Hearn, and Ian Stark. Ian's works on *names* significantly contribute to my understanding of local references in ML. I got many inspirations from his works.

Thanks to fellow students in LFCS, particularly Juliusz Chroboczek, Ewen Denney, Luis Dominguez, Masahito Hasegawa, Tim Heap, Alvaro Moreira, Nikos Mylonakis, and Jitka Stribrna. It is a pleasure to chat with them.

Thanks to the the computer support and maintenance people.

The administrative people have helped me in one way or another in providing information and administrative support. In particular I like to thank Margaret Davis and Bill Orrok.

To people who influenced me in my undergraduate years: Wesley Phoa, for encouraging me to come to Edinburgh to pursue a postgraduate study; Ken Robinson, for introducing me to the theory of programming language; and Arun Sharma, for teaching me recursion theory.

To Faraz Khan, Eira Williams, Anne Denniss, JanPieter Hoogma, and Jane Schonveld for their positive influence to my life.

To my parents and my family for everything. And I pray for their safety in Indonesia.

The first three years of the study was supported by the Overseas Research Students Studentships and the Faculty of Science and Engineering Scholarships.

This thesis uses Paul Taylor's diagram package.

Declaration

I declare that this thesis was composed by myself and that the work contained therein is my own, except where explicitly stated otherwise in the text.

(*Budi Halim Ling*)

Table of Contents

Chapter 1 Introduction **4**

1.1 Survey . . . 7

1.1.1 Syntactic control of interference . . . 7

1.1.2 Effects system . . . 9

1.1.3 Linear Logic based type system . . . 10

1.1.4 Single threaded lambda calculus . . . 11

1.2 Aim . . . 12

1.3 Method . . . 13

1.4 Results . . . 13

1.5 Synopsis . . . 14

Chapter 2 Notion of being constant **16**

2.1 Being constant in general . . . 16

2.1.1 Constant behaviour with respect to an observation . . . 16

2.1.2 Constant behaviour throughout an interaction . . . 18

2.1.3 Constant behaviour with a limited observation within an interaction . . . 21

2.2 Being constant in computation . . . 21

2.2.1 Notion of observation in computation . . . 21

2.2.2 Reachable sets . . . 23

2.2.3 Observing reference-type values . . . 24

2.2.4 Observing reference-type computations . . . 25

2.2.5 Observing int-type values and computations . . . 27

2.2.6 Observing functions of ground types . . . 28

2.2.7 Dynamic allocations . . . 28

2.2.8 Some examples . . . 30

Chapter 3 iML and operational semantics **33**

3.1 iML . . . 33

3.1.1 Language . . . 33

3.1.2 Operational Semantics 34
3.2 Operational characterisation 37
3.3 Discussion and example 38

Chapter 4 Semantics **41**
4.1 A semantics for iML 41
4.1.1 Issues in imperative functional languages 42
4.1.2 Preliminaries in giving semantics of ML-like languages . . 43
4.1.3 A denotational semantics for iML 45
4.2 Computational invariant 52
4.3 Being constant throughout a transition system 57
4.3.1 Background 57
4.3.2 Definition 59
4.3.3 Examples 61
4.4 Being constant within a transition system 64
4.4.1 Background 64
4.4.2 Definition 65
4.4.3 Example 65
4.4.4 Discussion 68

Chapter 5 Case study: Queue module **69**
5.1 Background 69
5.2 Modules in ML 72
5.2.1 Signatures and structures 72
5.2.2 `ref` and optimising an applicative module 75
5.3 Queue Module 76
5.3.1 Queue implementations 76
5.3.2 Complexity analysis 78
5.3.3 QueueOne equivalent to QueueTwo 79
5.3.4 Method for proving QueueOne equivalent to QueueThree . 84
5.3.5 Discussions 88
5.4 QueueOne equivalent to QueueThree 90
5.4.1 Definitions 91
5.4.2 Proof of being constant 93
5.4.3 Proof of equivalence 97

Chapter 6 Conclusions and directions for further research **102**

Appendix A Ingredients of monadic semantics **105**

Appendix B Proofs **109**
B.1 $Queue_1$ const-within R 109
B.2 Proof of Proposition 5.4.6 110
B.3 Proof of Proposition 5.4.7 111
B.4 Proof of Proposition 5.4.8 112
B.5 Proof of Proposition 5.4.15 115
B.6 Proof of Proposition 5.4.16 115
B.7 Proof of Proposition 5.4.17 116

Bibliography **119**

Chapter 1

Introduction

Over the last thirty years or so, there have been shifts of interest in programming language paradigm. From procedural languages that eliminate `goto`, to structured programming (e.g. Pascal, Modula-2), and to strongly-typed functional languages (e.g. Haskell, Miranda, and ML). Recently there has been growing interest in studying strongly-typed functional languages with state (e.g. Idealized Algol and ML). Note that Idealized Algol [Rey81b] is different from the classical Algol [And64] since the former has higher order types whereas the latter does not.

The popularity of pure functional languages is mainly attributed to their expressiveness: twenty lines of code written in a procedural language can often be rewritten in two or three lines in a functional language (a sorting algorithm is such an example[Tur86]). Pure functional languages are suitable for prototyping and easy to reason about. Programs can be regarded as mathematical functions. With their strong type systems, they guide programmers to avoid bugs at run time.

On the other hand, there are still places where the notion of state is useful in programming. State gives a natural view of some computational phenomena (e.g. reading/writing files and name generation) and it also can be used for code optimisation. Introducing state to functional languages gives programmers more flexibility in implementing their solutions.

We are not claiming that functional languages with states are the best among programming languages. Nor are we claiming that it is a new trend. In fact, the invention of such a language is not new (Lisp is an untyped version of such a language). What we have now is a growing interest in studying such languages. This is driven by theoretical and practical motivations.

On the theoretical side, there has been work towards fully abstract models for languages with local variables[MS88, Sie96b] and languages with names[Sta94].

The interaction between local variables or private names and higher order functions are not yet fully understood. Other work takes the design of reasoning systems for such languages[MT92, HMST95] an important areas of study.

On the practical side, functional languages with states share features similar to those exist in object-oriented languages. The following are the examples.

- We can encode the notion of object method and object identity. A private variable in an object is coded as a local reference in a function. The following code shows two implementations of an updatable integer class, one is written in Java and another in ML. The Java code is adapted from [CW96].

```
class AnIntegerNamedX {
  private int x;
  public int lookupX() {
    return x;
  }
  public void setX(int newX) {
    x = newX;
  }
}

...
AnIntegerNamedX myX = new AnIntegerNamedX();
myX.setX(1);
.... myX.lookupX() ...
...

fun class_AnIntegerNamedX () =
  let
    val class_x = ref 0
    fun class_lookupX () = !class_x
    fun class_setX newX = (class_x := newX)
  in
    {lookupX=class_lookupX, setX=class_setX}
  end

...
```

```
val myX = class_AnIntegerNamedX();
(#setX myX) (1);
... (#lookupX myX)() ...
...
```

- Conventional object-oriented languages do not have higher-order features. Thus, implementing a function such as `maplist` is cumbersome if not impossible.
- It is possible to add subtyping to functional languages. Such resulting languages would have a feature that resemble inheritance.

In relation to pure functional languages, functional languages with state allow programmers to optimise their programs at the programming language level. This may be desirable when we do not know how the compiler optimises our code.

The advantages of functional languages with state do not come without a price. The combination of pure functional language and state yields programming languages that are difficult to model and to reason about.

It is still possible to view programs as functions, but now we have states as implicit argument and output. A program `M` of type `int`$\to$`int` may be viewed as a function $f \in S \times N \to S \times N$ (where S is a space of stores and N the space of natural numbers). Thus giving a uniform denotational interpretation is more involved.

It is difficult to have a good reasoning technique for such languages. The usefulness of a reasoning technique is judged from its simplicity, its naturalness, and its completeness. The simple equational reasoning of lambda calculus is invalid in functional languages with side effects (eg. the η rule is violated). This is arguably one of the strongest objections using imperative functional languages in place of pure functional languages. If we had reasoning techniques that are simple enough for practising programmers to use, then the confidence in writing correct code in imperative functional languages would return.

The notion of type also changes in such languages. This is very much related to the issue of giving an interpretation of such a language, in particular the relationship between the syntactical notion of type (ie. type checking and type inference) and the semantical notion of type. At one extreme, types convey minimum information about the typable programs. This is the case in ML type inference. A program of type $\sigma \to \tau$ says that when it takes a value of type

σ the application may or may not produce side effects. At the other extreme types capture precisely the effects produced by typable programs. We are not interested in such type systems since they are undecidable. In the middle of the spectrum lie type systems that convey partial information about effects produced by typable programs. Effects system [LG88] is such an example. It can tell whether a particular global variable might be modified by a program. Since a type system enforces a particular discipline in writing programs, a good type system with effects information would guide programmers in the process of checking, transforming, and documenting code.

There have been some works on trying to deal with these issues. One line of work motivates the topic of this thesis. It is about finding subsets of the languages in which the good properties of pure functional languages are recovered. The next section gives a survey of recent works concerning this issue.

1.1 Survey

There have been ongoing attempts to find subsets of imperative functional languages which have good properties; in particular subsets where the programs are more modular and the side effects are controlled. These include: interference control, typing with side-effects information, type systems which are based on Linear Logic, and single threaded lambda calculus. The following sections describe recent papers on each of them. The survey is presented in the form of summaries of each of the papers. Each summary is divided into four parts: background, aim, method, and results.

1.1.1 Syntactic control of interference

The issues in this work concern variables and higher-order aliasing. Two variables are aliased when they refer to the same cell, whereas two functions are aliased when they refer to the same variable and they use the variable in incompatible ways (eg. both try to write or one tries to read while the other tries to write). Interference free languages give a modular extension to languages which include concurrency. This section summarises articles about syntactically eliminating interferences for Algol-like languages.

Syntactic control of interference [Rey78].
Background: interference contributes to errors in programs.
Aim: to eliminate interference by syntactical means.

Method: introduces a notion of passive types. These are for procedures that can read variables but not modify them.
Results: unfortunately subject reduction is not preserved.

Syntactic control of interference, part II [Rey89].
Background: same as in [Rey78]
Aim: to improve the syntactic control of interference defined in [Rey78].
Method: develops a type system using passive types (see [Rey78]) and conjunctive types (see [CDCV81]).
Results: a decidable type system that only accepts noninterfering programs. Subject reduction is preserved.

The semantics of non-interference: a natural approach [O'H90].
Background: to study inteference or aliasing in Algol-like languages.
Aim: to give a semantic characterisation of noninterference, in particular noninterference among higher-order functions.
Method: uses functor category for the characterisation of noninterference.
Results: the semantics is sound with respect to Reynolds Specification Logic. It gives a syntactical control of inteference which is sound with respect to the semantics.

Passivity and Independence [Red94].
Background: to use models of linear logic for understanding functional languages with states.
Aim: to give a semantical account of Reynolds' syntactic control of interference [Rey78] and to give a better model for explaining local variables.
Method: uses coherent spaces (with dependency relations) for characterising notions of historicity, absence of change, independent change, and passivity.
Results: a denotational account of passivity. An accurate model of local variables which can verify all of [MS88]'s examples.

Syntactic control of interference revisited [OTTP95].
Background: to give a syntactical and semantical account of interference control.
Aim: to define a type system for SCI (called SCIR) and provide its semantical framework.
Method: defines a typing judgement of the form $\Pi \mid \Gamma \vdash M : \theta$ where Π denotes passive zone and Γ active zone. Gives a categorical account of the type system

using bireflexive models.
Results: the type system satisfies subject reduction. The bireflexive model can interpret $\Pi \mid \Gamma \vdash M : \theta$ and the reduction relation preserves equality in any bireflexive model.

Type Reconstruction for SCI [HR95].
Background: controlling interference in Algol-like languages syntactically.
Aim: to give a type inference for SCIR defined in [OTTP95].
Method: extends SCIR with a new constraint information so that principal types exist.
Results: an algorithm for inferring types and interference information on Algol-like languages. This is an improvement over [Rey89]'s type system where SCI analysis is defined only for explicitly typed terms.

1.1.2 Effects system

Effects systems are concerned with developing more robust type systems for ML-like languages. The target type system should be able to give information about the effects produced by typable programs. A good effects system is one that gives information about global side effects and masks unobservable local side effects. The soundness of the type systems are measured with respect to the operational semantics.

Polymorphic Effect Systems [LG88].
Background: a type system for reasoning and implementing compilers for functional languages with state and polymorphism.
Aim: to encode information about side effects and regions in types.
Method: a type conveys three pieces of information: the original notion of type for describing the value that an expression may return, an effect for describing side effect, and a region for describing the area of the state in which side effects may occur. It has two kinds of typing judgment: $A, B \vdash e : \tau$ and $A, B \vdash e : \epsilon$ where ϵ denotes an effect.
Results: the effect system is sound. Formally, if $\{\}, \{\} \vdash e : \epsilon$ and $(e, s) \stackrel{red}{\Rightarrow} (v, s')$, then the reduction process $\stackrel{red}{\Rightarrow}$ does not violate the constraint imposed by ϵ.

The type and effect discipline [TJ92].
Background: a more robust type system for an ML-like language.
Aim: to reconstruct types and effects information from ML-like programs.

Method: defines a notion of type-expression parameterised by effects and regions. Defines a notion of inclusion between effects.
Results: The type system is sound with respect to the operational semantics. It provides an algorithm for inferring types and effects information.

1.1.3 Linear Logic based type system

Linear Logic can be viewed as a logic of resource; in particular, copying has to be done explicitly. This methodology can be transferred to pure functional languages by developing a type system which requires explicit annotation for resources which are duplicated or discarded. Such a type system is finer than the usual type system in the sense that it may distinguish two equivalent programs where one program uses resources in a different way than another.

Linear types can change the world! [Wad90].
Background: We can introduce an imperative aspect to a language by extending a pure functional language with destructive array update mechanism. Arrays can be used to model states efficiently. On the other hand, we would like the type system to tell us how the arrays are used.
Aim: to use ideas from Linear Logic for designing a type system for programs that neither duplicate nor discard an array.
Method: Has two families of types: values of linear types and values of nonlinear types. The rules for linear types are reflections of Girard's Linear Logic.
Results: a type system that can distinguish linear and nonlinear usage of arrays. It shows an example of implementing an interpreter for a simple imperative programming language under a functional language with destructive array update.

Once upon a type [TWM95].
Background: a type system that can determine how many times a resouce is used. This is useful for compiler optimisation.
Aim: a type inference which detects when values are accessed at most once.
Method: extends the Hindley-Milner type system with *uses* information. A typing judgement takes the form $\Gamma \vdash_{\Theta} e : \tau$ where Θ is a set of constraints. The type system uses ideas from Linear Logic [Gir87] and Logic of Unity [Gir93].
Results: satisfies subject reduction with respect to call-by-need reduction. If a term is typable, then the principal type exists.

1.1.4 Single threaded lambda calculus

Single threaded lambda calculus is a language based on lambda calculus extended with an mutation operator and a mechanism for sequencing computation. The type system is designed to ensure single-threadedness. A single threaded program is a program that never duplicates its mutable datatype (e.g. array) in an unsafe way. This property ensures the referential transparency of such language and at the same time is expressive enough to model store and store updates. When we use states in functional languages, we normally do not duplicate the whole states; rather we change or modify the old state by updating a small fraction of its cells. The mutation operator in the single threaded lambda calculus provides just this mechanism.

Assignments for Applicative Languages [SRI91].
Background: trying to find a functional language with states which satisfies referential transparency.
Aim: extend lambda calculus with assignments while maintaining the static views of functions.
Method: designs a type system that ensures that expressions have no side effects. Type-expressions are divided into three layers: applicative types, mutable types, and observer types.
Results: the type system satisfies subject reduction. It has an operational semantics which is confluent.

Single-threaded Polymorphic Lambda Calculus [GH90].
Background: this extends a pure functional language like Haskell with a mutator operator while still maintaining referential transparency for the extended language.
Aim: To design a language extension and a type system that is able to restrict the language into a subset which satisfies referential transparency.
Method: Defines a language called *single-threaded-lambda-calculus*. It is basically a call-by-name functional language extended with a construct for applying function application in a single threaded way. The language is then extended with a mutator operation and a type system for rejecting programs that may not be confluent.
Results: Typable terms are confluent. It has a type reconstruction algorithm.

1.2 Aim

The work in this thesis is driven by the observation that some programs that use references internally still behave in an applicative way. To get a better understanding of the issue, we need to explain the notion of 'use references internally' and 'behave in an applicative way'.

In a programming language like ML, there are at least two ways a reference can be used internally. The first way is to allocate a fresh location `x` and to return a term `f` that does not explicitly reveal `x` to the surrounding context. The following is an example.

```
val M = let
          val x = ref 0
          fun f n = (x:= !x+4; if isEven(!x) then n else !x)
        in f
        end
```

The surrounding context cannot have arbitrary access to `x`. It can only affect `x` by invoking the identifier `M`.

The second way is to return a pointer `x`, but the pointer is encapsulated by abstract data type mechanism. Hence the surrounding context regards `x` as an abstract value and cannot invoke arbitrary operations on `x`. An example of this is an implementation of a Queue data structure (see `QueueThree` on page 77).

ML programs written without `ref`, `:=`, and `!` can be viewed as mathematical functions. They are fully described by the graphs of their input-output pairs. For programs that use `ref`, `:=`, or `!` the situation is more involved because the output of such a program depends on the current state as well as the input.

The identifier `M` defined above has the property that given a fixed input n, the outputs of (`M` n) are constant throughout all possible current states producable/reachable by its surrounding context. Notice that this set of possible states is a subset of the set of all states. The set is essentially determined by an invariant that the identifier `M` satisfies and how private `x` is. If `M` satisfies an invariant I and `x` is 'private enough', then the possible surrounding contexts would also satify I.

We do not study the notion of private references in depth, therefore we parameterise the notion of being constant over a class of relevant contexts. One of the aims of this thesis is to find a suitable mathematical structure for expressing a class of contexts.

We would like to give a semantic framework of being constant with respect to a class of contexts. Moreover, we want to build the framework in a setting that is easy to understand.

We should emphasise that our work is not primarily about finding a type system or a decision procedure that can detect constant programs. Rather, we aim to build a framework for understanding the notion of constant programs. Developing a type system is possible future work.

1.3 Method

The language studied in this thesis is a pure ML fragment extended with `:=`, `!`, and `ref`, where the values storable by references are integers. This is essentially the language Reduced ML [Sta94]. Although the language does not have recursion, we can encode nontrivial programs such as counter and memoisation programs.

We give an operational and denotational semantics of the language, with more emphasis on the denotational semantics. We model types and programs in terms of sets and functions and side effects in terms of side effect constructor $TA = S \to S \times A$, where S is a set of states [Mog91]. One of the most delicate issues is how to define the notion of *being constant throughout a class of contexts*. We decide to express the notion of a class of contexts by an invariant that they must satisfy. This is justifyable since the only aspects of a context that is relevant to us is the invariant that it satisfies.

We study two structures for modelling invariants: one is a reachable set, and the more general one is a transition system. Intuitively, a reachable set consists of an initial state s_0 and the possible states that are reachable from s_0. A transition system is a binary relation on states. It is more general than a reachable set since we can extract only the possible future states from the current state and the set of all reachable states. Hence a transition system is a more accurate structure for expressing dynamic allocation.

The definition of *being constant throughout* a computational invariant R requires the notion of *indistinguishable throughout* R and we define the latter in a logical relation fashion [Mit90]. In effect, we define a class of relations indexed by type expressions.

1.4 Results

We give two pairs of definitions of indistinguishability and being constant with respect to a transition system R. The first pair is geared towards programs with flat store. Programs with flat store are ones that use only global variables. The definition is called:

$$\{indistinguishable\ throughout_\sigma\ R\},\ and$$
$$\{const\text{-}throughout_\sigma\ R\}.$$

We show that the notion of *indistinguishable throughout*$_\sigma$ R is an equivalence relation over elements of $[\![\sigma]\!]$ that are *constant throughout*$_\sigma$ R. The property of *constant throughout*$_\sigma$ R is preserved under function application.

The second pair of the definitions makes use of the 'future sensitive' property of transition systems. It can handle the freedom of of choosing fresh names when we allocate new locations and it only quantifies the behaviour of a computation over the current and future states and prevents quantifying the computation over previous states. The definition of *indistinguishable* is parameterised over pairs of transition relations instead of transition relations because we have to take the freedom of choosing fresh locations into account. The pair is called:

$$\{indistinguishable\ within_\sigma\ R_1, R_2\},\ and$$
$$\{const\text{-}within_\sigma\ R\}.$$

The first notation is used for comparing a denotation a living in a relation R_1 to another denotation b living in R_2. This framework is used in Chapter 5 for showing an imperative ML implementation of Queue data structure indistinguishable within R,R to its pure counterpart, where R is a transition system that both implementations satisfy. This is one of the main practical contributions of the thesis.

1.5 Synopsis

Chapter 2 aims to introduce an informal notion of being constant in computation. It emphasises the notion of observation, a computation which satisfies an invariant, a reachable set for capturing computations that satisfy an invariant, and a notion of indistingishability throughout an invariant.

Chapter 3 defines a language for our discussion. It is a pure fragment of ML extended with `ref`, `:=`, and `!`. We define its operational semantics and operationally characterise the notion of indistinguishability throughout a reachable set and being constant throughout a reachable set. The method is essentially the same as the one used in Chapter 2.

Chapter 4 extends the definitions in Chapter 2 by replacing reachable sets with transition systems. We define notions called *indistinguishable throughout R* and *const-throughout* R and show that these are generalisations of their counterparts in Chapter 2. We give another definition which can handle dynamic allocation and call them *indistinguishable within* R_1,R_2 and *const-within* R and show that *indistinguishable within* can equate an example which involves creating new locations.

Chapter 5 is the practical side of the thesis. It gives concrete ML implementations of Queue module, with one implementation written in code that internally uses references. It then shows that we can use the notion of *indistinguishable within* for showing the denotation of a pure implementation indistinguishable to the denotation of the imperative implementation with respect to a pair of identical transition systems R,R, where R is a transition system that both denotations satisfy. We can view Chapter 5 as the driving force of the thesis and the previous chapters as giving a suitable framework for analysing Queue module.

Chapter 6 concludes the thesis and gives some discussions towards further research.

Chapter 2

Notion of being constant

The purpose of this chapter is to introduce the concept of constant computation. Section 2.1 shows some examples of constant behaviour in general phenomena, and sum up their key elements in understanding constant behaviour. The key elements are: the levels of details an observer can pick up from using or interacting with the system and the constraint that an observer must obey when interacting with the system.

Section 2.2 shows that the key elements in understanding constant behaviour in general phenomena also apply in the case of computation, but the situation is more delicate. When a program has a higher order feature, it is not clear what being constant means. When programs cannot have access to arbitrary information about current store, it is not straightforward to determine the scope of information they can access from the store and the possible side effects they can inflict upon the store. When the store is dynamically changed and new references can be allocated to the store, determining the environment's possible action upon the store such that the program still behaves in a constant way requires a new insight into the behaviour of computations with dynamic allocations. These are what make analysis of constant computations subtle.

2.1 Being constant in general

2.1.1 Constant behaviour with respect to an observation

Constant functions are abundant in the real world.

When we consider just the space of functions, it is not clear whether constant functions are abundant. But when we are talking about the real world, then there are a lot of them. The reason is that when we model a phenomenon using a mathematical function, the accuracy of our observation has to be taken into account. Let us pick up an example from, say, electronics. To be precise, supposing we want to analyse the behaviour of an electric switch with respect to the voltage of a particular node in a circuit. Consider a circuit in Figure 2.1. This circuit consists of a series of a resistor, a switch, and a DC source. We are interested in determining the voltage at node n when the switch is closed. A typical description of the voltage is described in Figure 2.2.

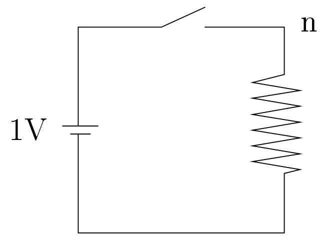

Figure 2.1: A switch connected in serial to a resistor and a DC source

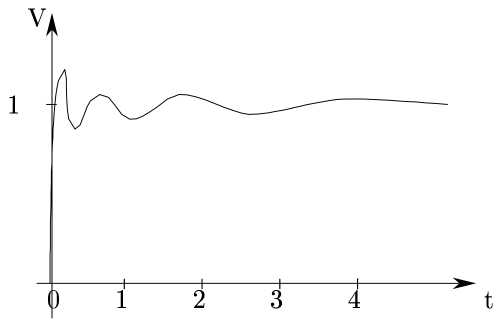

Figure 2.2: The voltage of node n after the switch is set to on

Initially we have the switch set to off. At time t equals zero, the switch is set to on, allowing the current to flow through the circuit.

Figure 2.2 shows a typical real life situation. Indeed, mathematically, it is quite a complex one. When it is working along with other millions of switches – as in the case of the internals of a processor – then the above model is too complex to comprehend. However, when we know that the switch can only be observed at a discrete interval of one second and the sampling resolution is 0.2 volts, then its behaviour, according to the observer is described in Figure 2.3

All the fluctuations are not observable. To the observer, the voltages (after $t = 1$ *second*) are constant. All that the observer knows is that before and at the time the switch is on the values are zero volts and at the first second the switch is on and afterwards the values are 1 volts.

The graph in Figure 2.3 suggests that we have a 'high-level' view of the behaviour of the switch, which is portrayed in Figure 2.4.

The graph in Figure 2.4 provides a nice and simple abstraction of the behaviour of the switch. To what extent this abstraction is accurate depends on the

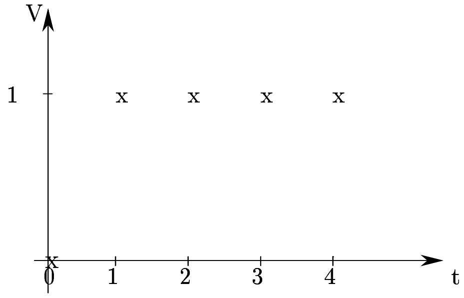

Figure 2.3: The voltage at node n sampled at 1 sample/s with resolution 0.2 volts

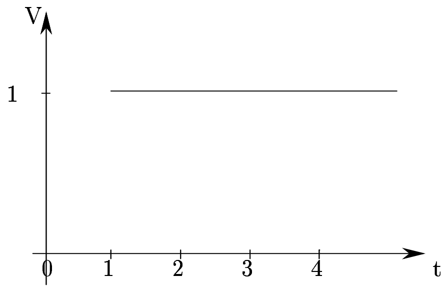

Figure 2.4: A simplified view of the voltage at n

applications, but in many cases this abstraction is adequate.

To sum up, the following are the essential points in understanding constant behaviour when we have a limited degree of observation:

- We describe a system in terms of a mathematical function. In other words, it is an information transformer.
- The 'resolution' of observing the system determines how the observer views the behaviour of the system. When the observer cannot detect the fluctuations of the output, then the observer can deduce that the output does not change.

2.1.2 Constant behaviour throughout an interaction

Another dimension of constant behaviour involves the issue of interaction. In an interaction, there are two protagonists: an agent and an environment. The usual scenario consists of an environment interacting with the agent and getting some information out of the interactions. Some interactions may change the behaviour of the agent as well as the environment. The 'states' of the environment and the agent before and after the interaction might differ — the difference reflects the effects of the interaction.

With the issue of constant behaviours, we are interested in classifying a set of interactions such that the behaviour of the agent is unaltered.

There are many constant behaviour patterns in the worlds of interactions, be it interactions between a human and other human being, a tree with the wind or weather, or the sun with the earth (for other examples, think of objects being used in a dynamic way). We study a simple example which consists of a spring and a mass (see Figure 2.5). Let us say that we are interested in hanging the mass on the spring and seeing how much the spring extends. Assuming the spring is ideal, we have a linear curve characterising the coefficient of the spring (see Figure 2.6).

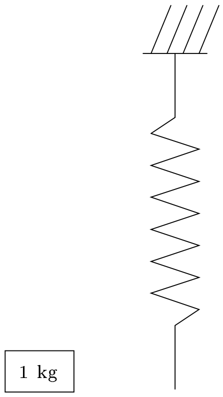

Figure 2.5: Agents of interactions: the spring and the mass

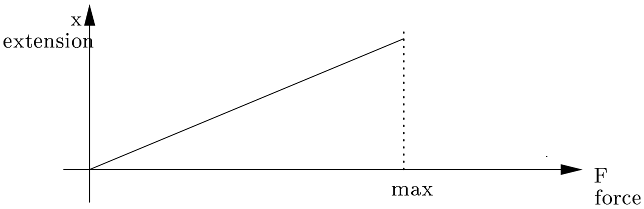

Figure 2.6: The graph of extension of the spring against the force applied to it

In this example, the spring is the agent and we are the environment; the interactions are the putting of masses to the spring and measuring its extensions. Figure 2.7 shows some possible 'interactions' we can perform on the spring.

Notice that in Figure 2.6 the curve does not go on forever; it is only defined up to force F equals max. Intuitively, it is obvious since the spring cannot stretch out indefinitely. When the weight is too heavy, it will break (see Figure 2.8).

The thing is that the graph of Figure 2.6 describes the behaviour of the spring as long as it is not broken. Once it is broken, its 'intended behaviour' is altered.

Computer minded readers could see that there is an element of 'state' here. The state of a spring is either "in good form" or "broken". When the spring is in good form, it can serve us as a weight indicator (whose characteristics is specified in Figure 2.6). When it is broken, it is useless.

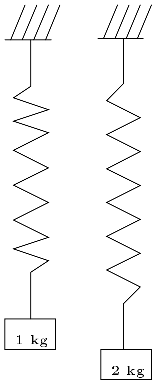

Figure 2.7: Some possible scenarios of interaction

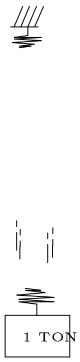

Figure 2.8: When the force is beyond the constraint, the spring breaks

Pragmatically we do not want to break the spring. The way to do it is to restrict the amount of weight applied to it. Figure 2.6 gives a guidance that the force F applicable to the spring must be less than or equal to max. In computing terminology, we say that the *invariant* that must be preserved during the interaction is :

$$0 \leq F \leq max. \tag{2.1}$$

Strictly speaking, Figure 2.6 in itself is not a constant function. However, what we are interested in is whether the coefficient of the spring is unaltered throughout its use. The notion of constancy in this setting is different to the notion of constancy in the circuit example.

To sum up, the following are the essential points in understanding constant behaviour throughout an interaction.

1. Some objects are best described in terms of their interactions with the environment which uses them. From the environment point of view, an object can be considered as providing a function.

2. This function is not altered as long as the environment satisfies the restrictions on how to use the object.

2.1.3 Constant behaviour with a limited observation within an interaction

In general the phenomena of constant behaviours involve both elements of the 'resolution' of observation and the constraint in the interaction. For example, in the spring example above, normally we assume that our measurement of the spring extension is accurate only up to the nearest millimetre, say. It may be possible that the spring is vibrating with amplitude of less than a millimetre; or that given the same mass experimented twice on the spring, the first experiment differs from the second one by a deviation of less than a millimetre.

2.2 Being constant in computation

2.2.1 Notion of observation in computation

This subsection describes the standard notion of context and the notion of observation in terms of context, where a context is a program fragment with a hole. In a functional language with state, the notion of observation is more involved because in general a context may modify the behaviour of a program it observes.

In the previous section we described two important notions in understanding constant behaviours. They are the notion of resolution of an observation and the notion of preserving a certain condition throughout an interaction. The combination of the two definitions gives an accurate account of the notion of constant behaviour within an interaction. The two physical examples use physical devices (a voltmeter or oscilloscope in the former example and a ruler in the latter) for measuring the phenomena. The accuracy of the observations depend on the resolutions of the measuring devices. Physicists have standards of the level of accuracy that they need for a certain observation. Do we have an accuracy standard in observing a program? How do we define the resolution of a method for observing a program?

There are various answers to the last question. From a low level languages (such as assembly languages, C, and C++) programmer point of view, a widely accepted method to observe a program is by using a debugger. The bare essential features of a debugger includes an interpreter that can evaluates a program in a step-by-step fashion and a window that displays the current states or variables. When the hardware has a debugging facility, a debugger can display complete information on how a program runs from one step to another.

In a single machine not connected to a network, this is the lowest level and the most fine grain analysis we can perform.

From a program correctness point of view, the above analysis is too low level. It requires the knowledge on how the actual program is executed in a hardware (ie. the operational semantics) and — in many cases — how the compiler transforms a program into a machine code (or a lower level code). These low level details are useful in optimising and finding better internal representations of programs, but they are too complex for analysis of the logical behaviours of programs. We need a different notion of observation which is at the higher level of abstraction.

A widely accepted notion for observing a program at the logical level can be explained in terms of a notion of *context*. A context is a program with a hole. It is denoted by the symbol $C[\,.\,]$. C is the program part of the context and $[\,.\,]$ is where we plug in the program we want to observe. We can view a context as a program parameterised by programs. A context represents a possible environment for observing programs. In a typed programming language, we need to match the types before plugging in a program to a hole since programs, holes, and contexts have types. In the following discussion, we assume that we only plug a type-compatible program to a context.

We observe a program by using other programs to observe it. In our setting we apply the program to contexts. There are important relationships between contexts, programs, and indistinguishability. Two programs are equal when they are indistinguishable throughout all possible contexts. This is a notion of *observational equivalence* in [Plo77] and is a characterisation of equality at the operational level.

The notions of observation and indistinguishability in pure functional languages can be explained in terms of contexts. However, in functional languages with state the above notions are more involved, because the way a context observes a program can also affect the programs behaviour. This is because a context can change the current state and the output of a program may depend on the current state as well as its input. This is the major source of difficulties in reasoning about functional languages with state.

This thesis studies a notion called invariant for understanding the behaviour of functional languages with state. The introduction of invariant is natural for capturing the dynamic aspects of computation.

We decide to express context not in terms of a syntactical construct, but in terms of a semantical structure. The reason is that we want to characterise a class of contexts that satisfy an invariant. If we define a context as a syntactical construct, then we have to mix syntactical constructs with semantical structures, and there is no clear way on how to do this. Thus we decide to shift everything on to a semantical level.

In the next section we describe a structure for modelling invariants. It is called *reachable set*. Intuitively, a reachable set consists of an initial state s_0 and the possible states that are reachable from s_0. In Chapter 4, we give a more general structure called transition relation.

2.2.2 Reachable sets

We are interested in functional languages with references (such as ML). In particular, our focus will be on functional languages with natural number references only. We need to define states which can handle unused and allocated references.

Definition 2.2.1. *L is a countable set.*

$AllStates = L \to N \cup \{Unused\}$,	N *the set of natural numbers.*
l *defined-in* s *iff* $s(l) \neq Unused$,	*for* $l \in L$ *and* $s \in AllStates$.
$dom\ s = \{l \in L \mid\ l \text{ defined-in } s\}$,	*for* $s \in AllStates$.
$S = \{s \in AllStates \mid\ dom\ s\ is\ finite\}$.	

In this chapter, we would model a class of contexts with the notion of reachable set. Intuitively, a reachable set is a set of states reachable from an initial state s_0. Formally, the definition is given below

Definition 2.2.2. *A reachable set $\mathcal{Q}$ is a tuple $< Q, s_0 >$ where $Q \subseteq S$ and $s_0 \in Q$.*

The state s_0 denotes an initial state whereas the nonempty set Q denotes the possible states that are reachable from s_0. We would abbreviate $\mathcal{Q}$ with its underlying set Q.

The motivation behind this definition is that we are interested in a notion of observation over a reachable set. The use of a reachable set provides a window where we can restrict our observation into the set of computations that are relevant.

A reachable set represents the possible states that could be computed by a class of contexts. In other words, the abstraction of a class of contexts is that they are a reachable set, and observation with respect to a class of contexts is observation with respect to a reachable set.

In this chapter we deal with computations with flat stores (ie. no dynamic allocation). In this setting, the relevant locations are the ones that are definable in every reachable state. The following defines what it means for a location to be definable in a reachable set.

Definition 2.2.3. *Let Q be a reachable set.*
l defined-in Q iff forall $s \in Q$: l defined-in s.

2.2.3 Observing reference-type values

Natural numbers and locations are both 'ground types', but the former are pure objects whereas the latter are imperative objects. We do not need the current state to observe numbers, but we need it to observe references. This is because the behaviour of a reference is determined relative to the current store.

If we know the reachable set of a class of computations, and we know that a particular reference points to the same value in every state s in the reachable set, then computationally this means the 'lookup part' of the reference behaves in a constant way throughout this class of computations. This is the basis of our idea of a constant location relative to a reachable set.

Definition 2.2.4. *Let $l \in L$.*
l V-const-throughout Q
iff
forall $s_i, s_j \in Q$:
$s_i(l) = s_j(l)$

The 'V' in the prediate *being-constant* shows that the predicate is defined over values instead of computations. When it is clear from context we would omit the 'V' letter.

Another issue in the difference between natural numbers and locations is the notion of indistinguishability. Two natural numbers are indistinguishable when they are equal. The same characterisation does not apply for locations, because it is possible to have two fresh locations which at the programming language level are indistinguishable. Therefore we have to resort to a more abstract view of equality.

We are interested in defining locations indistinguishability with respect to a class of context. The following is the formal definition.

Definition 2.2.5. *Let $l, k \in L$.*
l,k V-indistinguishable throughout Q
iff
forall $s_i, s_j \in Q$:
$s_i(l) = s_j(k)$

It says that two locations are indistinguishable throughout a reachable set Q if their lookup parts agree over Q. It follows that if l,k *V-indistinguishable throughout* Q then l *V-const-throughout* Q. Similarly for k.

Our definition of location indistinguishability is driven by practical motivation: we would like to use our definition in analysing ML modules which use references but still behave in an applicative way. Such a module is a Queue module[1] where we have an efficient implementation by using a pointer to represent a queue. We are allowed to pass the pointers around and they can be manipulated in a restricted way, but we are not allowed to compare them.

The following are some properties of locations indistinguishability.

Proposition 2.2.6 (Reflexivity on indistinguishability). *Let* $l \in L$.
l *V-const-throughout* Q
iff
l,l *V-indistinguishable throughout* Q

This suggests that we can define location constancy in terms of location indistinguishability.

Proposition 2.2.7. *The relation V-indistinguishable throughout* Q *is a partial equivalence relation over* L.

Corollary 2.2.8. *Let* $\overline{L} = \{\ l \in L \mid\ l$ *V-const-throughout* $Q\}$. *The relation V-indistinguishable throughout* Q *is an equivalence relation over* $\overline{L}$.

2.2.4 Observing reference-type computations

In an ML-like language, its call-by-value nature enforces the computations to only allow canonical values to be passed around. When we want to pass an expression to a function, the expression has to be evaluated first to a canonical value. Taking side effects into account, the evaluation may change the current state.

The notions of canonical value and expression at the operational level can be viewed as notions of value and computation at the denotational level [Mog89]. An expression M of type `int ref` is modelled as a function $m \in S \to S \times L$. The process of reducing M to a canonical value corresponds to the process of applying M to the current state s and obtaining an after-state s' and a location l.

We would like to define the notion of indistinguishability throughout $< Q, s_0 >$ for computations of arbitrary types. For an expression, the process of reducing the expression into a canonical value may create side effects. One condition of

[1] see page 77 of this thesis

two expressions being indistinguishable is to require the side effects satisfy the invariant Q. The second condition is to require that all possible canonical values to which the expressions reduce are *V-indistinguishable throughout* Q when the expressions are reduced under Q. The following is the formal definition. The variable A denotes an arbitrary set.

Definition 2.2.9 (Some Conventions). *Given* $s \in S; m \in S \to S \times A$, *we define* $(m\ s)_1 = \pi_1(m\ s)$ *and* $(m\ s)_2 = \pi_2(m\ s)$.

Definition 2.2.10. *Let* $m, n\ \in S \to S \times A$.
m, n *C-indistinguishable throughout* Q
iff
1. forall $s \in Q$ *:* $(m\ s)_1 \in Q \wedge (n\ s)_1 \in Q$
2. <u>let</u>
$\quad\overline{A}{=}\{(m\ s)_2 \mid\ s \in Q\} \cup \{(n\ s)_2 \mid\ s \in Q\}$
<u>in assert</u>
$\quad$*forall* $a, b \in \overline{A}$ *: a,b V-indistinguishable throughout* Q
<u>end</u>

The first clause says that Q is a state invariant that the functions m and n must satisfy. The set $\overline{A}$ denotes the collection of values computed by m and n over the reachable set Q. The 'C' in the prediate *indistinguishable throughout* shows that the predicate is defined over computations instead of values. When it is clear from context we would omit the 'C' letter.

The notion of constancy throughout Q for computations of arbitrary types can be defined in terms of *C-indistinguishable throughout* Q.

Definition 2.2.11. *Let* $m \in S \to S \times A$.
m *C-const-throughout* Q *iff* m, m *C-indistinguishable throughout* Q

The above definitions specialises to location computations when we substitue A with the location space L. The following are some properties of *indistinguishable throughout* Q and *const-throughout* Q for location computations.

Proposition 2.2.12. *Let* $m \in S \to S \times L$.
if m *C-const-throughout* Q
then forall $s\ \in Q$ *:* $(m\ s)_2$ *V-const-throughout* Q.

Proposition 2.2.13. *The relation C-indistinguishable throughout* Q *is a partial equivalence relation over* $S \to S \times L$.

Corollary 2.2.14. *Let $m, n \in S \to S \times L$.*
If m,n C-indistinguishable throughout Q,
then m C-const-throughout Q and
n C-const-throughout Q.

2.2.5 Observing int-type values and computations

We need definitions of being constant and indistinguishability for the ground type `int`. For integer values, the definitions are simple since natural number values are pure objects.

Definition 2.2.15. *Let $n \in N$*
n V-const-throughout Q
iff
true

Definition 2.2.16. *Let $a, b \in N$*
a, b V-indistinguishable throughout Q
iff
$a = b$

The definitions of indistinguishability and constancy for computations of natural numbers type are special cases of Definition 2.2.10 and 2.2.11 with the variable A substituted by the space of natural numbers N.

Definition 2.2.17. *Let $m, n \in S \to S \times N$.*
m, n C-indistinguishable throughout Q
iff
1. forall $s \in Q : (m\ s)_1 \in Q \wedge (n\ s)_1 \in Q$
2. <u>let</u>
$\overline{N}$=$\{(m\ s)_2 \mid s \in Q\} \cup \{(n\ s)_2 \mid s \in Q\}$
<u>in assert</u>
forall $a, b \in \overline{N}$: a,b V-indistinguishable throughout Q
<u>end</u>

Definition 2.2.18. *Let $m \in S \to S \times N$.*
m C-const-throughout Q iff m, m C-indistinguishable throughout Q

Similarly, Propositions 2.2.12 and 2.2.13 apply to natural numbers computations.

Proposition 2.2.19. *The relation C-indistinguishable throughout Q is a partial equivalence relation over $S \to S \times N$.*

2.2.6 Observing functions of ground types

Our method of defining *being-constant* and *indistinguishability* for function types contains two elements: the first element is that we define them in logical relation fashion. The second is that a canonical value of function type is interpreted as a function that takes values and produces computations. An expression of type σ is interpreted as an element of type $T[\![\sigma]\!]$ where $TA = S \to S \times A$ (see [Mog91]).

This subsection gives general and uniform definitions of *being-constant* and *indistinguishability* for function types. In principle the definition applies to arbitrary type expressions; but since we have not developed the formal setting[2] of the semantics, we only define them for functions of ground types.

For the rest of this subsection, the variables A, B are quantified over $\{N, L\}$.

Definition 2.2.20. *Let* $f, g \in A \to (S \to S \times B)$
f, g *V-indistinguishable throughout* Q
iff
forall $a, b \in A$ *:*
    a,b V-indistinguishable throughout $Q \Rightarrow$
    (f a),(g b) C-indistinguishable throughout Q

Definition 2.2.21. *Let* $f \in N \to (S \to S \times L)$
f *V-const-throughout* Q *iff* f, f *V-indistinguishable throughout* Q

Proposition 2.2.22. *The relation V-indistinguishable throughout Q is a partial equivalence relation over $A \to TB$.*

2.2.7 Dynamic allocations

The structure reachable set is good enough for modelling computations that deal only with global variables, but it is not sufficient for computations that involve dynamic allocations. For a computation that allocates fresh locations, the reachable set would contain the initial state s_0 and a larger state s_1 where s_1 has more defined locations than s_0. Let l_x be a location defined in s_1 but not in s_0. Then l_x is not *const-throughout* Q, since $s_0(l_x) \neq s_1(l_x)$. Such a decision is premature, since if the rest of the computation maintains the lookup value of l_x, then we would like to say that l_x is *const-throughout* Q.

[2] The formal setting is defined in Chapter 4

This suggests that we cannot force $s_i(l_x) = s_j(l_x)$ for all $s_i, s_j \in Q$ because the status of an allocated location differs from the status of an unallocated one. To make the distinction on their status, we need to filter out the undefined locations before comparing the values they point to. The alternative definition of *being-constant* would be:

l_x *V-const-throughout* Q
iff
<u>*let*</u> $\overline{Q} = \{s \in Q \mid l_x \textit{ defined-in } s\}$
<u>*in assert*</u>
  forall $s_i, s_j \in \overline{Q} \; : s_i(l_x) = s_j(l_x)$
<u>*end*</u>

However, this solution still cannot reason about names accurately. Consider the

```
val M = let
          val x = ref 7
        in fn y => y = x
        end

val N = fn (y:int ref) => false
```

$m = [\![\texttt{M}]\!] \; \in S \to (S \times (L \to TB))$
$n = \; [\![\texttt{N}]\!] \; \in S \to (S \times (L \to TB))$
where
  $TB = S \to S \times B$
  $B = \{\mathit{true}, \mathit{false}\}$

Figure 2.9: Two equivalent programs with program `M` written using a local name.

programs `M` and `N` in Figure 2.9. When we evaluate M, we create a new reference `x` and output a function that takes a reference and compares it with `x`. This function is bound to an identifier `M`. Since `x` is a new reference and it cannot escape the function `M`, any input to `M` will be different to `x`. Operationally `M` behaves like a constant expression and is observationally equivalent to `N`. However we cannot find a suitable reachable set Q such that m,n *C-indistinguishable throughout* Q. Here is the sketch of the proof: reducing `M` corresponds to applying m to the current state (say s_0) and yielding an after state s_1 and a function $f \in L \to TB$. Similarly, we apply n to s_0 and get an after-state s_0 and a function $g \in L \to TB$. The after state s_1 is larger than s_0 since it has a new location (say l_x) allocated. Since l_x is hidden and cannot be modified, we expect the values pointed by l_x

are either *Unused* or 7 for all states in Q. Hence l_x is *constant throughout* Q. However, $(f\ l_x)$ and $(g\ l_x)$ are not *C-indistinguishable throughout* Q. Hence m and n are not *C-indistinguishable throughout* Q.

The above argument shows that the structure of reachable set essentially describes a state invariant for global variables. For a more accurate analysis of computations that involve private location and dynamic allocation, we have to use structures that are parameterised over stores. This framework is used in functor category technique for reasoning about names [Sta94] and also apparent in operational technique for reasoning about names using state relations [Sta97, PS98].

We need to reemphasise that our main aim is not to find structures for reasoning about names since we consider that is beyond the scope of our topic which is defining the notion of being-constant. Our aim is driven by practical issues of implementing applicative programs in terms of programs that internally use references. Such programs do not have references in their input or output types, and when they do, the references are encapsulated by abstract data types (for an example, see the Queue implementation using references on page 77).

Reachable sets have some practical uses in reasoning about programs that use dynamic allocations. Although at the formal setting it cannot handle dynamic allocations, at the semi-formal setting it is applicable for reasoning about programs with dynamic allocations. The following is the technique:

> Given an expression M and the current state s_0, reduce it to a canonical value C and an after-state s_1.
>
> $$s_0{,}M \Downarrow C{,}s_1$$
>
> If C does not have the potential to allocate new locations, then we can reason at the flat store level by restricting the relevant reachable set into states that are reachable from s_1.

2.2.8 Some examples

Example 2.2.23. Consider two imperative ML programs.

```
val M = (x := !x + 4; 7)
val N = (x := !x + 2; 7)
```

Where `x` is a global reference already allocated before the two programs are declared.

Let l_x be the location meaning of `x`. Then the meanings of `M` and `N` are defined as the following.

$m \in S \rightarrow S \times N$
$m\ s = \underline{let}$
$\quad v = s(l_x) + 4$
$\quad \underline{in}$
$\quad (s[l_x \mapsto v], 7)$
$\quad \underline{end}$

$n \in S \rightarrow S \times N$
$n\ s = \underline{let}$
$\quad v = s(l_x) + 2$
$\quad \underline{in}$
$\quad (s[l_x \mapsto v], 7)$
$\quad \underline{end}$

Consider an fixed state s_0 where $s_0(l_x) = 0$ and a reachable set $< Q, s_0 >$ where

$$Q = \{s \in S \mid s(l_x)\ is\ even\} \tag{2.2}$$

It is easy to show that *m,n C-indistinguishable throughout Q*. They both satisfy the invariant Q since they both preserve the evenness of $s(l_x)$ for $s \in Q$. The second condition of Definition 2.2.10 is satisfied since for all $s \in Q$, $(m\ s)_2 = 7 = (n\ s)_2$.

Example 2.2.24. Consider the following ML programs.

```
val M = let val x = ref 0
            fun f n = if (isEven (!x))
                      then (x := !x+2; n)
                      else (!x)
        in f
        end


fun id (n:int) = n
```

`M` is an expression whereas `id` is a canonical value. We would use our semi-formal technique for proving `M` equivalent to `id`. Let s_0 be the initial state when we reduce `M`. The reduction process is described as:

$$s_0,\mathtt{M} \Downarrow C, s_0[l_x \mapsto 0]$$

which says that the result of the reduction is a canonical value C and an after-state $s_0[l_x \mapsto 0]$. We are interested in comparing C with `id` at the denotational level. Their meanings are:

$$
\begin{array}{l}
f = [\![C]\!] \ \in N \rightarrow (S \rightarrow S \times N) \\
f\ n\ s = \mathit{if}\ s(l_x)\ \mathit{is\ even} \\
\qquad\qquad \mathit{then}\ \underline{\mathit{let}} \\
\qquad\qquad\qquad\quad v = s(l_x) + 2 \\
\qquad\qquad\qquad \underline{\mathit{in}} \\
\qquad\qquad\qquad\quad (s[l_x \mapsto v], n) \\
\qquad\qquad\qquad \underline{\mathit{end}} \\
\qquad\qquad \mathit{else}\ (s, s(l_x))
\end{array}
$$

$$
\begin{array}{l}
id = [\![\mathtt{id}]\!] \in N \rightarrow (S \rightarrow S \times N) \\
id\ n\ s = (s, n)
\end{array}
$$

Consider a reachable set $< Q, s_0[l_x \mapsto 0] >$ where

$$Q = \{s \in S \mid s(l_x)\ \mathit{is\ even}\} \tag{2.3}$$

Then we have f, id *V-indistinguishable throughout* Q. This is because for any $a \in N$, we have $(f\ \ a), (id\ a)$ *C-indistinguishable throughout* Q.

Chapter 3

iML and operational semantics

Section 3.1 defines a functional language with states and its operational semantics. Section 3.2 gives an operational characterisation of indistinguishability throughout a reachable set and being constant throughout a reachable set. These are the operational counterparts of the definitions in Chapter 2.

3.1 iML

This section defines the language iML (imperative ML) and its operational semantics. It is a pure fragment of ML extended with `ref`, `!`, and `:=`. Essentially the language is the same as Reduced ML (RML) [Sta94] minus the reference equality $=_{\texttt{int ref}}$. Since $=_{\texttt{int ref}}$ is definable in iML (see later), they are essentially the same language.

3.1.1 Language

The type expressions contain `int ref` for typing programs of type integer references. Sometimes we would abbreviate `int ref` to `ref`. The type expressions and the terms are the following.

$$\tau ::= \texttt{int} \mid \texttt{unit} \mid \texttt{bool} \mid \texttt{int ref} \mid \sigma \to \tau.$$

$M ::= x$	variable
$\mid\ l$	locations
$\mid\ \texttt{()} \mid n \mid \texttt{true} \mid \texttt{false}$	basic constants
$\mid\ \texttt{if}\ M_1\ \texttt{then}\ M_2\ \texttt{else}\ M_3$	conditional
$\mid\ M_1 \oplus M_2$	operations on integers
$\mid\ M_1 \otimes M_2$	tests on integers
$\mid\ \texttt{ref}\ M$	new reference

$\mid$	$!M$	lookup
$\mid$	M := N	assignment
$\mid$	$\lambda x{:}\sigma.M$	function abstraction
$\mid$	MN	function application

where $\oplus \in \{$+,-,*$\}$ and $\otimes \in \{$<,>,=,<=,>=,<>$\}$.

Notice that the language does not have recursion. However, the language is rich enough to express nontrivial programs such as counter, memoisation, and profiling programs.

We would use the following syntactic abbreviations.

- `let val x` $= M$ `in` $N \equiv (\lambda x{:}\sigma.N)M$
- `let fun f x` $= M$ `in` $N \equiv$ `let val f` $= (\lambda x{:}\sigma.M)$ `in` N `end` where `f` is not free in M.
- $M;N \equiv (\lambda x{:}\sigma.N)M$, $\qquad$ x not free in N.

We define $loc(M)$ as the free locations that occur in M. The type system is a simple extension of the type system of simply typed lambda calculus. The extensions are rules for handling constants, conditional, integer operations and comparisons, and reference operations. The type system is shown in Figure 3.1. We use the following notation for the type system. The variable b ranges over boolean literals and a ranges over integer literals. The variable $\oplus$ in $M_1 \oplus M_2$ ranges over $\{$+,-,*$\}$ and the variable $\otimes$ in $M_1 \otimes M_2$ ranges over $\{$<,>,=,<=,>=,<>$\}$. Note that the symbol $\oplus$ is also used for patching type environments (see the typing rules for function abstraction).

The type system of iML inherits properties satisfied by the type system of RML. In particular, we have the following from [Sta94].

Fact 3.1.1.
1. If $\Gamma \vdash M : \sigma$ then the type σ is unique.
2. If $\Gamma \vdash M : \sigma$ then $\Gamma \oplus \Gamma' \vdash M : \sigma$.

3.1.2 Operational Semantics

At the operational level, we need to distinguish between expressions and canonical values.

$$\frac{}{\Gamma \vdash x{:}\sigma} x : \sigma \in \Gamma \qquad \frac{}{\Gamma \vdash l \ : \texttt{int ref}}$$

$$\frac{}{\Gamma \vdash \texttt{()} \ : \texttt{unit}} \qquad \frac{}{\Gamma \vdash b \ : \texttt{bool}} \qquad \frac{}{\Gamma \vdash a \ : \texttt{int}}$$

$$\frac{\Gamma \vdash M_1 \ : \texttt{bool} \quad \Gamma \vdash M_2 \ : \sigma \quad \Gamma \vdash M_3 \ : \sigma}{\Gamma \vdash \texttt{if } M_1 \texttt{ then } M_2 \texttt{ else } M_3 : \sigma} \qquad \frac{\Gamma \vdash M_1 \ : \texttt{int} \quad \Gamma \vdash M_2 \ : \texttt{int}}{\Gamma \vdash M_1 \oplus M_2 \ : \texttt{int}}$$

$$\frac{\Gamma \vdash M_1 \ : \texttt{int} \quad \Gamma \vdash M_2 \ : \texttt{int}}{\Gamma \vdash M_1 \otimes M_2 \ : \texttt{bool}} \qquad \frac{\Gamma \vdash M \ : \texttt{int}}{\Gamma \vdash \texttt{ref } M \ : \texttt{int ref}}$$

$$\frac{\Gamma \vdash M \ : \texttt{int ref}}{\Gamma \vdash \texttt{!}M \ : \texttt{int}} \qquad \frac{\Gamma \vdash M \ : \texttt{int ref} \quad \Gamma \vdash N \ : \texttt{int}}{\Gamma \vdash M \texttt{ := } N \ : \texttt{unit}}$$

$$\frac{\Gamma \oplus \{x : \sigma\} \vdash M \ : \tau}{\Gamma \vdash \lambda x : \sigma . M \ : \sigma \to \tau} \qquad \frac{\Gamma \vdash M \ : \sigma \to \tau \quad \Gamma \vdash N \ : \sigma}{\Gamma \vdash MN \ : \tau}$$

Figure 3.1: Type system of iML

Definition 3.1.2.
M is in canonical form iff M is a variable, a location, a ground constant, or a function abstraction.

$\texttt{Exp}_\sigma(\Gamma) = \{M \mid \Gamma \vdash M : \sigma\}$
$\texttt{Can}_\sigma(\Gamma) = \{C \in \texttt{Exp}_\sigma(\Gamma) \mid C \ \textit{canonical}\}$
$\texttt{Exp}_\sigma = \texttt{Exp}_\sigma(\{\})$
$\texttt{Can}_\sigma = \texttt{Can}_\sigma(\{\})$
$\texttt{Exp} \ = \bigcup\{\texttt{Exp}_\sigma \mid \sigma \ \textit{a type}\}$
$\texttt{Can} \ = \bigcup\{\texttt{Can}_\sigma \mid \sigma \ \textit{a type}\}$

We define a big-step reduction semantics of the form

$$s{,}M \Downarrow C{,}s'$$

where $M \in \texttt{Exp}_\sigma$ and $C \in \texttt{Can}_\sigma$. It says that starting from a state s, the term M reduces to a canonical value C with the final state s'. We only consider well formed judgements, where $loc(M) \subseteq (dom\ s)$ and $loc(C) \subseteq (dom\ s')$.

The operational semantics is shown in Figure 3.2. The definition is essentially taken from the operational semantics for RML[Sta94]. For binary operations on integers we give a definition for +. The others (including tests on integers) are similarly defined.

$$s,C \Downarrow C,s$$

$$\frac{s,M_1 \Downarrow \texttt{true},s_1 \quad s_1,M_2 \Downarrow C,s_2}{s,\texttt{if } M_1 \texttt{ then } M_2 \texttt{ else } M_3 \Downarrow C,s_2} \qquad \frac{s,M_1 \Downarrow \texttt{false},s_1 \quad s_1,M_3 \Downarrow C,s_2}{s,\texttt{if } M_1 \texttt{ then } M_2 \texttt{ else } M_3 \Downarrow C,s_2}$$

$$\frac{s,M_1 \Downarrow a,s_1 \quad s_1,M_2 \Downarrow a',s_2}{s,M_1 \texttt{ + } M_2 \Downarrow a+a',s_2} \qquad \frac{s,M \Downarrow n,s' \quad l \notin dom\ s'}{s,\texttt{ref } M \Downarrow l,s'[l \mapsto n]}$$

$$\frac{s,M \Downarrow l,s'}{s,\texttt{!}M \Downarrow s'(l),s'} \qquad \frac{s,M \Downarrow l,s_1 \quad s_1,N \Downarrow n,s_2}{s,M \texttt{:=} N \Downarrow \texttt{()},s_2[l \mapsto n]}$$

$$\frac{s,M \Downarrow \lambda x{:}\sigma.M',s_1 \quad s_1,N \Downarrow C,s_2 \quad s_2,M'[C/x] \Downarrow C',s_3}{s,MN \Downarrow C',s_3}$$

Figure 3.2: Operational semantics of iML

Notice that in function application rule, N has to be reduced to a canonical value before it can be applied.

Proposition 3.1.3. *If $s,M \Downarrow C,s'$ then $dom\ s \subseteq dom\ s'$.*

Proof: By induction on the structure of the derivation of the evaluation judgement. ∎

Although iML lacks a primitive function $=_{\texttt{int ref}}$ for comparing locations, such function is definable in iML (this is pointed to by McCusker[McC97a]). The following is the definition.

```
fun eqRef x y =
  let
   val tmp = !x
   val _ = x := !y + 1
   val isAliased = if ((!x)=(!y)) then true
                   else false
  in
   (x := tmp; isAliased)
  end
```

The trick is that we use aliasing property for detecting whether two pointers are the same. If they are aliased, then a value change in one of them will be reflected in the other. This method does not work for `unit ref`.

Proposition 3.1.4. `eqRef` *satisfies the following rules.*

$$\frac{s{,}M_1 \Downarrow l{,}s_1 \quad s_1{,}M_2 \Downarrow l{,}s_2}{s{,}\texttt{eqRef}\ M_1\ M_2 \Downarrow \texttt{true}{,}s_2}$$

$$\frac{s{,}M_1 \Downarrow l{,}s_1 \quad s_1{,}M_2 \Downarrow k{,}s_2}{s{,}\texttt{eqRef}\ M_1\ M_2 \Downarrow \texttt{false}{,}s_2}\ l \neq k$$

Hence iML and RML are interdefinable.

iML inherits the following two properties from RML. The first property says that unreachable stores do not affect the reduction. The second property says that evaluations of terms always terminate.

Fact 3.1.5.

1. *For all* $s \in S$ *and* $M \in \texttt{Exp}_\sigma$ *s.t.* $loc(M) \subseteq (dom\ s)$ *:*
   $s{,}M \Downarrow C{,}s'$ *iff* $s{\oplus}s''{,}M \Downarrow C{,}s'{\oplus}s''$
2. *For all* $s \in S$ *and* $M \in \texttt{Exp}_\sigma$ *s.t.* $loc(M) \subseteq (dom\ s)$ *:*
   there exist $s' \in S$, $C \in \texttt{Can}_\sigma$ *s.t.* $(dom\ s) \subseteq (dom\ s') \wedge s{,}M \Downarrow C{,}s'$.

3.2 Operational characterisation

This section gives an operational characterisation of indistinguishable throughout a reachable set Q and being constant throughout Q. The definitions are very much similar to the ones in Chapter 2, but here we skip the motivational part. Moreover, the operational definitions are defined over all type expressions.

We define two families of binary logical relations parameterised over type expressions and reachable sets.

$$\textit{V-op-indistinguishable throughout}_\sigma\ Q \subseteq \texttt{Can}_\sigma \times \texttt{Can}_\sigma$$

$$\textit{E-op-indistinguishable throughout}_\sigma\ Q \subseteq \texttt{Exp}_\sigma \times \texttt{Exp}_\sigma$$

Figure 3.3 shows their definitions. It is instructive to compare the definitions with their denotational counterparts in Chapter 2. The definitions for values are the same. The definition of *E-op-indistinguishable throughout* is a rewriting of the definition of *C-indistinguishable throughout* at the operational setting.

Figure 3.4 shows the operational characterisation of being constant throughout Q. We have the following properties.

Proposition 3.2.1.
1. The relation V-op-indistinguishable throughout Q is a partial equivalence relation over $\texttt{Can}_\sigma$.
2. The relation E-op-indistinguishable throughout Q is a partial equivalence relation over $\texttt{Exp}_\sigma$.

$$
\begin{array}{|l|}
\hline
\texttt{()},\texttt{()}\ \textit{V-op-indistinguishable throughout}_{\texttt{unit}}\ Q \quad \textit{iff} \quad \textit{true} \\
a,b\ \textit{V-op-indistinguishable throughout}_{o}\ Q \quad \textit{iff} \quad a=b \qquad\qquad o\in\{\texttt{int},\texttt{bool}\} \\
l,k\ \textit{V-op-indistinguishable throughout}_{\texttt{ref}}\ Q \quad \textit{iff} \\
\quad \textit{forall } s_i,s_j\in Q : s_i(l)=s_j(k) \\
\\
V_1,V_2\ \textit{V-op-indistinguishable throughout}_{\sigma\to\tau}\ Q \quad \textit{iff} \\
\quad \textit{forall } W_1,W_2\in Can_\sigma : \\
\qquad W_1,W_2\ \textit{V-op-indistinguishable throughout}_{\sigma}\ Q \\
\qquad\quad \textit{implies } (V_1\ W_1),(V_2\ W_2)\ \textit{E-op-indistinguishable throughout}_{\tau}\ Q \\
\\
M_1,M_2\ \textit{E-op-indistinguishable throughout}_{\sigma}\ Q \quad \textit{iff} \\
\quad \textit{forall } s_i,s_j\in Q,\ V_1,V_2\in Can_\sigma,\ s_i',s_j'\in S : \\
\quad \left.\begin{array}{l} s_i,M_1\Downarrow V_1,s_i' \\ s_j,M_2\Downarrow V_2,s_j' \end{array}\right\}\textit{implies} \\
\qquad s_i',s_j'\in Q \wedge V_1,V_2\ \textit{V-op-indistinguishable throughout}_{\sigma}\ Q \\
\hline
\end{array}
$$

Figure 3.3: Operational definition of $\{$*V-op-indistinguishable throughout*$_\sigma$ $Q\}$ and $\{$*E-op-indistinguishable throughout*$_\sigma$ $Q\}$.

$$
\begin{array}{|l|}
\hline
C\ \textit{V-op-const-throughout}_{\sigma}\ Q \quad \textit{iff} \quad C,C\ \textit{V-op-indistinguishable throughout}_{\sigma}\ Q \\
M\ \textit{E-op-const-throughout}_{\sigma}\ Q \quad \textit{iff} \quad M,M\ \textit{E-op-indistinguishable throughout}_{\sigma}\ Q \\
\hline
\end{array}
$$

Figure 3.4: Operational definition of $\{$*V-op-const throughout*$_\sigma$ $Q\}$ and $\{$*E-op-const throughout*$_\sigma$ $Q\}$.

3.3 Discussion and example

The definition of *V-op-indistinguishable throughout Q* in Figure 3.3 is similar in spirit to the operational logical relation method of reasoning about names in nu-calculus [Sta94]. The latter method uses the structure partial bijection $R : s_1 \rightleftharpoons s_2$ (where s_1 and s_2 are collections of names) for relating the local names of two terms. There are two essential differences. Firstly, [Sta94] uses partial bijection $R : s_1 \rightleftharpoons s_2$ whereas we use reachable set. R is used to relate the names in s_1

with names in s_2. This structure is used for comparing two names, where name equality is defined as:

$$n_1, n_2 \ \textit{V-equal}_{\texttt{ref}} \ R \iff R(n_1, n_2)$$

Secondly, the difference lies in comparing expressions. The definition of expression equality in nu-calculus is defined as:

$$\begin{array}{l}
M_1, M_2 \ \textit{E-equal}_\sigma \ R \iff \\
\quad \exists R' : s'_1 \rightleftharpoons s'_2, C_1 \in \texttt{Can}_\sigma(s_1 \oplus s'_1), C_2 \in \texttt{Can}_\sigma(s_2 \oplus s'_2). \\
\qquad s_1, M_1 \Downarrow C_1, (s_1 \oplus s'_1) \wedge \\
\qquad s_2, M_2 \Downarrow C_2, (s_2 \oplus s'_2) \wedge \\
\qquad C_1, C_2 \ \textit{V-equal}_\sigma \ (R \oplus R')
\end{array}$$

The definition does not assert that the after states need to satisfy an invariant, whereas our definition explicitly mentions this.

It should be emphasised that the above comparison is sketchy since we are comparing a language with states (of integer values) with a language with names.

It is also instructive to compare Definition in Figure 3.3 with its denotational counterpart in Chapter 2. However, since Chapter 2 does not give a denotational definition of iML, we can only compare them at the example level.

The semi formal technique outlined on page 30 is also applicable at the operational setting. We illustrate this by redoing the analysis in Example 2.2.24 on page 31. Let `M` be a term defined in Example 2.2.24, s_0 be the initial state when we reduce `M`. The reduction process is:

$$s_0, \texttt{M} \Downarrow C, s_0[l_x \mapsto 0]$$

where

$l_x \notin \textit{dom } s_0$

$C = \lambda \texttt{n.if (isEven(!}l_x\texttt{)) then (}l_x \texttt{ := !}l_x\texttt{+2; n) else (!}l_x\texttt{)}$.

Consider a reachable set $< Q, s_0[l_x \mapsto 0] >$ defined in 2.3 on page 32. We want to show C,`id` *V-op-indistinguishable throughout*$_{\texttt{int}\to\texttt{int}}$ Q. Consider an arbitrary integer literal `n`, $s_i, s_j \in Q$. We have

$$s_i, C \ \texttt{n} \Downarrow \texttt{n}, s_i[l_x \mapsto s_i(l_x) + 2]$$
$$s_j, \texttt{id n} \Downarrow \texttt{n}, s_j$$

The after state $s_i[l_x \mapsto s_i(l_x){+}2]$ is in Q; therefore $(C \ \texttt{n})$,(`id n`) *E-op-indistinguishable throughout*$_{\texttt{int}}$ Q. ∎

The next chapter gives a denotational setting for the notion of indistinguishability and being constant with respect to an invariant, where an invariant in modelled by a transition system.

Chapter 4

Semantics

Section 4.1 gives a background in giving a denotational semantics of functional language with states and defines a store-semantics model for iML. Section 4.2 defines the structure transition system for expressing the notion of computational invariant. The structure is a generalisation of the structure reachable set defined in Chapter 2. Section 4.3 gives a method and the formal definition of being constant throughout a transition system. This definition is good enough for analysing terms with flat stores. Section 4.4 gives another definition which can handle the freedom of allocating fresh locations in *new*. The definition of indistinguishability is parameterised over pairs of transition systems because we need to take the freedom of choosing fresh locations into accounts. It gives a method, the formal definition, and an example that the meanings of `(ref 8)` and `((ref 7);(ref 8))` are indistinguishable within R_1, R_2, for a suitable pair of (R_1, R_2).

4.1 A semantics for iML

The beauty of strongly typed, pure functional languages is that they have a simple and clear semantics: types are viewed as sets and programs are viewed as functions. The notion of program composition can be explained in terms of the usual mathematical notion of function applications. This high level view of programs is what makes functional languages effective tools in programming [Hug89, Hen86]. The idea that programs can be viewed statically as mathematical functions implies that we can use the abundant mathematical tools for reasoning about programs. In particular, we can replace equals with equals. Referential transparency guides programmers to avoid bugs in the coding process.

Adding novel features to the language still retains this 'programs-as-functions' view. With the addition of recursion, there is an adequate model that interprets types as complete partial orders and programs as continuous functions (see [Plo83] for a more detailed exposition of this issue.) These continuous functions are still functions in the set theoretic sense, and program compositions are still the usual function compositions. At the syntactic level, reasoning about recursive programs can be done by viewing them as recursive equations [Tur82]. An additional technique for reasoning about recursive programs is by using induction. This can be done at the denotational level [Pau87] or syntactic level [BW88].

With the addition of references and assignments, new complexities are introduced that are difficult to resolve elegantly. It is more difficult to reason about programs, to teach the language to students, and to give a simple and elegant denotational model. The following are the explanations.

4.1.1 Issues in imperative functional languages

The equality of the expanded language is not a conservative extension of the equality of the pure fragment. Consider the following example by [RV95].

```
fun M1 x f g = ((f x);(g x))
fun M2 x f g = ((f x);(f x);(g x))
```

`M1` takes a value `x` and two functions `f` and `g` and applies `f` to `x` and `g` to `x`. `M2` is similar to `M1`, but it does `(f x)` twice before doing `(g x)`. In the pure fragment of ML, `M1` and `M2` are observationally equivalent. When we extend the language to include `ref`, `!`, and `:=`, there is a context that can distinguish them. The following is such a context expressed as an ML function.

```
fun myCtxt M =
  let
    val r = ref 0
    fun inc x = (r := !r + 1)
    fun lkUp x = !r
  in
    M () inc lkUp
  end
```

The application of `(myCtxt M1)` returns `1` whereas `(myCtxt M2)` returns `2`.

The above example suggests that now we have to take care of its intensional behaviour as well as its extensional one. The program `(x := !x + 1; 5)` is not the same as the program `5` although they always yield the same value.

It is more difficult to give a consistent presentation to students on the nature of programs in the context of an imperative functional language. In ML courses (for example, [Gil97]), students are introduced to the pure ML fragment and are told that programs are functions. But when we introduce references and assigments, we need to tell them that programs are actually functions that take input states and produce output states. This shift of perception may create ambiguity and confusion among students: which one is the actual meaning of a program?

Another complication is the interactions between higher order functions and imperative features. We have Hoare Logic [Hoa69] for reasoning about While Language (the logic can be extended to include first order functional languages with assigment) but threre is no widely accepted method for reasoning about higher order functions in terms of their pre-postcondition behaviours. The most substantial work on related to this is Reynolds' Specification Logic [Rey81a] for reasoning about Idealized Algol programs. No similar logics for ML have been developed.

The `ref` construct in ML can be used to code name generations and local variables (for some examples, see [Sta94], p.102-105). The price we have to pay is that it is difficult to give a simple and elegant semantics.

If we want to retain the 'programs-as-functions' view, then we have to view a program as a function which also takes an input state and produces an output state. The states are implicit arguments. The existence of implicit arguments makes it cumbersome to define program composition. At the denotational level, a composition of two programs is no longer just a function composition : we have to 'convert' the domain and codomain of the interpretations into an appropriate structure before composing them.

The following is our approach to giving a semantics of an ML-like language for studying the notion of being constant with respect to a computational invariant.

4.1.2 Preliminaries in giving semantics of ML-like languages

4.1.2.1 Store semantics

A simple language that deals with assignment is While Language [Hoa69]. With the While Language the central type in the language is `comm`. Its interpretation is simply a space of state transformers.

$S = L \rightarrow V$
$[\![\texttt{comm}]\!] = S \rightarrow S$

Note that L is a set of locations and V is a set of storable values.

In the case of a while language with higher order features, the essential idea does not change. We have a notion of an L-value, which is a location l in L and an R-value which is a storable value v in V. The meaning of a reference is a location l. Assignment on l changes the current store s in S by modifying the value pointed by l. Notice that an assignment does not change the location but the value pointed to by the location.

This idea is explained in [Str73]. The idea can be used in a number of variants of imperative functional languages including Idealized Algol and ML.

4.1.2.2 Monad

If we want to retain the programs-as-functions view of an imperative functional language, there is an issue in how to compose programs, since we have to take care of the implicit state arguments before composing their interpretations. This mismatch of the domain-codomain of the interpretations can be solved by having separate notions of values and computations [Mog89]. The idea is that a computation is a value with some other information (usually intensional information). In our case, a side effect is the additional information. A program in general is no longer a value, but rather a computation. A program of function type is a computation that accepts a value and produces a computation. It cannot accept a computation because only values can be passed.

The separation of values and computations is explicitly expressed in terms of monad structure [Man76, Mog89, Cen96, Fil96]. The relevant feature of a monad structure is that it has an endofunctor T which is used for expressing computation type and a special arrow $let_{A,B} : TB^A \times TA \to TB$ for composing a computation of type TA with a value of type $A \to TB$. TA is a space of computations which return elements of type A. TB^A is a function space that takes a value space A to a computation space TB. The behaviour of let is that it takes f in TB^A and m in TA, evaluates m to a value of type A and applies f to the result while taking care of the side effect produced in evaluating m.

The formal definition of the structure that we need can be described in terms of strong monad. The following definition is adopted from [Sta94].

Definition 4.1.1. *A strong monad over a cartesian closed category $\mathcal{C}$ consists of an endofunctor $T : C \to C$ together with a unit natural transformation $\eta : 1 \to T$ and a lift operating taking $f : A \times B \to TC$ to $f^* : A \times TB \to TC$ such that*

1. $(\eta_B \circ snd_{A,B})^* = snd_{A,TB}$

2. $f^*\circ(id_A \times \eta_B) = f$
3. $g^*\circ < fst_{A,TB}, f^* >= (g^*\circ < fst_{A,B}, f >)^*$

whenever $f : A \times B \rightarrow TC$ *and* $g : A \times C \rightarrow TD$.

The above structure is equivalent to Kleisli triple with tensorial strength.

This structure gives rise to a computational metalanguage λ_C. Basically, λ_C is a simply typed lambda calculus augmented with `let` construct for dealing with computations. The language λ_C provides a syntactic framework for the notion of values and computations. It is augmented with an equational logic which is sound with respect to the categorical structure [Mog89].

4.1.2.3 The appropriate structure

We will use the side effect functor over the category $\mathcal{S}et$ of sets for defining the semantics of iML. The reasons are twofold: firstly, our focus is on characterising applicativity and we want to put most of our efforts in this area. The best way is to use a simple semantics and explain the core notions (which include the notion of computational invariant and being constant) in terms of this simple notation. There are other more sophisticated structures such as functor categories [Sta94] and games models[1][AM97] which give full abstraction results at least for ground types. Our approach in using sets and functions for modelling types and programs does not score well in full abstraction issue, but it is simple to understand.

The second reason is that we are analysing the notion of being constant with respect to a computational invariant R. In our setting, R is a transition system used for expressing computational constraints. It is not clear how to construct such notions when we are working in functor category. In the game semantics model, the situation is less clear. For example, the notion of pre-postcondition in game semantics is more complicated than the store-semantics one [McC97b].

4.1.3 A denotational semantics for iML

This section describes the denotational semantics that we use for the purpose of studying the notion of being constant with respect to a computational invariant. The nature of the semantics is a store semantics [Str73, Sto77] using the category $\mathcal{S}et$ of sets and functions structured in a monadic way [Mog89].

[1]The language has a construct `makevar` : (`unit`$\rightarrow$`int`) $\rightarrow$(`int`$\rightarrow$`unit`) $\rightarrow$`int ref` for creating new 'variable objects'.

The method is a standard monadic method of giving an interpretation function of a programming language.

$$\text{iML} \xrightarrow{(\!|\ |\!)} \text{CML}_{\text{iML}} \xrightarrow{[\![\ ]\!]} (T, \eta, \_^*) \tag{4.1}$$

First we define a computational metalanguage CML_{iML} which is rich enough to interpret constructs in iML. Then we define a model $(T, \eta, \_^*)$ for CML_{iML}, where T is the side effect functor over $\mathcal{S}et$. Then we define the interpretation $[\![\ ]\!]$ from CML_{iML} to the model and the interpretation $(\!|\ |\!)$ from iML to CML_{iML}. We would also use $[\![\ ]\!]$ to denote the composition of the arrows in (4.1). An exposition of this method for general notions of computation is provided in Appendix A.

The following subsections describes the details of the steps.

4.1.3.1 Metalanguage CML_{iML}

The type expressions of the metalanguage are defined as:

$\tau ::= \texttt{int}\ \mid \texttt{unit}\ \mid \texttt{bool}\ \mid \texttt{int ref}\ \mid \sigma \to \tau \mid T\sigma.$

The definition is similar with the definition of type expressions for iML (see Chapter 3, page 33), but here we have an additional construct $T\sigma$ for expressiong computations of type σ. A computation not only produces a values but also may create side effects.

The terms of CML_{iML} is the terms of the computational metalanguage λ_C extended with the necessary constructs for interpreting iML. As well as conditional and arithmetical constructs, an important set of constructs are `new`, `lookup`, and `update`. These are constructs for intrepreting location allocations, dereferencings, and assignments in iML.

$M ::= x$	variable
$\mid$ `()` $\mid n \mid$ `true` $\mid$ `false`	basic constants
$\mid$ $\texttt{cond}(M_1,M_2,M_3)$	conditional
$\mid$ $\texttt{plus}(M_1,M_2) \mid \ldots$	operations on integers
$\mid$ $\texttt{<}(M_1,M_2) \mid \ldots$	tests on integers
$\mid$ $\texttt{new}(M)$	new reference
$\mid$ $\texttt{lookup}(M)$	lookup
$\mid$ $\texttt{update}(M,N)$	assignment
$\mid$ $\lambda x{:}\sigma.M$	function abstraction
$\mid$ MN	function application
$\mid$ $[M]$	value as computation
$\mid$ $\texttt{let}\ x \Leftarrow M\ \texttt{in}\ N$	sequential computation

The intuitive meaning of $\texttt{let } x \Leftarrow M \texttt{ in } N$ is that we first evaluate M to a value v, take any side effects created into account, then evaluate N in a context where x has value v. The constructs `new`, `lookup`, and `update` behave like `ref`, `!`, and `:=` in iML.

The type system is presented in Figure 4.1. Notice that function application rule is the usual one as in pure functional languages. On the other hand, we have '`let`' application which, when applied, has a computation type.

$$\frac{}{\Gamma \vdash x : \sigma} x : \sigma \in \Gamma \qquad \frac{}{\Gamma \vdash \texttt{()} : \texttt{unit}}$$

$$\frac{}{\Gamma \vdash a : \texttt{int}} \qquad \frac{}{\Gamma \vdash b : \texttt{bool}}$$

$$\frac{\Gamma \vdash M_1 : \texttt{bool} \quad \Gamma \vdash M_2 : \sigma \quad \Gamma \vdash M_3 : \sigma}{\Gamma \vdash \texttt{cond}(M_1, M_2, M_3) : \sigma}$$

$$\frac{\Gamma \vdash M_1 : \texttt{int} \quad \Gamma \vdash M_2 : \texttt{int}}{\Gamma \vdash \texttt{plus}(M_1, M_2) : \texttt{int}} \qquad \frac{\Gamma \vdash M_1 : \texttt{int} \quad \Gamma \vdash M_2 : \texttt{int}}{\Gamma \vdash \texttt{<}(M_1, M_2) : \texttt{bool}}$$

$$\frac{\Gamma \vdash M : \texttt{int}}{\Gamma \vdash \texttt{new}(M) : T(\texttt{int ref})} \qquad \frac{\Gamma \vdash M : \texttt{int ref}}{\Gamma \vdash \texttt{lookup}(M) : T\texttt{int}}$$

$$\frac{\Gamma \vdash M : \texttt{int ref} \quad \Gamma \vdash N : \texttt{int}}{\Gamma \vdash \texttt{update}(M, N) : T\texttt{unit}}$$

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x : \sigma . M : \sigma \rightarrow \tau} \qquad \frac{\Gamma \vdash M \colon \sigma \rightarrow \tau \quad \Gamma \vdash N \colon \sigma}{\Gamma \vdash MN : \tau}$$

$$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash [M] : T\sigma} \qquad \frac{\Gamma \vdash M : T\sigma \quad \Gamma, x : \sigma \vdash N : T\tau}{\Gamma \vdash \texttt{let } x \Leftarrow M \texttt{ in } N : T\tau}$$

Figure 4.1: Type system for $\mathrm{CML}_{\mathrm{iML}}$

The computational metalanguage $\mathrm{CML}_{\mathrm{iML}}$ is not new. It is essentially the computational metalanguage defined for Reduced ML in [Sta94]. The only significant difference is here `new` takes an integer expression and yields $T(\texttt{int ref})$ whereas in [Sta94] `new` has the type $T(\texttt{int ref})$. We decide to use the former definition to avoid dealing with a fixed initial value.

4.1.3.2 Store model for $\mathrm{CML}_{\mathrm{iML}}$

Definition 4.1.1 on page 44 gives a general framework for modelling computations. This subsection presents a particular model based on sets and functions. It is essentially the traditional store semantics models in monadic setting. At the categorical level what we need is a structure that satisfies Definition 4.1.2.

Definition 4.1.2. *The categorical model consists of a category $\mathcal{C}$ such that:*
a. It is cartesian closed.
b. It has a strong monad $T : \mathcal{C} \to \mathcal{C}$.

Since $\mathrm{CML}_{\mathrm{iML}}$ has a mechanism for allocating a new reference (`new`), we would need a semantical counterpart *new* for interpreting it. We would define *new* in terms of a deterministic function $\mathit{Select} \in S \to L$ which selects a fresh location in a given store. The following is the formal definition of a selection function.

Definition 4.1.3. *A function $\mathit{Select} \in S \to L$ is a selection function iff $\forall s \in S.\mathit{Select}(s) \notin \mathit{dom}\ s$.*

Our set theoretical model $\mathcal{M}_{\mathit{Select}}$ is indexed over selection functions. Formally, the interpretation function $[\![\ ]\!]_{\mathit{Select}}$ is also indexed over selection functions. When the selection function is clear from context, we would omit the subscript.

Let *Select* be a selection function. A model $\mathcal{M}_{\mathit{Select}}$ is the following.

$N = \{0, 1, 2, ...\}$ $\mathbf{1} = \{*\}$ Also see Definition 2.2.1 on page 23.

Figure 4.2: Primitive sets for semantics of iML

- The category is the category of sets and functions $\mathcal{S}et$. In particular, we are interested in the set of natural numbers, the singleton set, a countable set of locations, and a set of stores (see Figure 4.2).

- The strong monad is the standard side effect monad defined in Figure 4.3. T is the side effect constructor. An element of TA is a computation that takes the current state and produces an updated state and a value of type A. The function η is indexed by the set A. It is a function that converts values into computations. The lifting $\_^*$ operation enables us to interpret compositions of iML terms.

$$
\boxed{\begin{array}{l}
TA = S \to S \times A \\
\\
\eta_A : A \to TA \\
\eta_A\ v\ s = (s, v) \\
\\
\dfrac{f : A \times B \to TC}{f^* : A \times TB \to TC} \\
f^*\ (a, m)\ s = \underline{let} \\
\qquad\qquad\qquad (s_1, v) = m\ s \\
\qquad\qquad\quad \underline{in} \\
\qquad\qquad\qquad f\ (a, v)\ s_1 \\
\qquad\qquad\quad \underline{end}
\end{array}}
$$

Figure 4.3: Kleisli structure for side effect monad

- We are interested in the following functions in the category.

  $cond_A : B \times A \times A \to A$
  $plus : N \times N \to N$
  $lt : N \times N \to N$
  $new : N \to TL$
  $lookup : L \to TN$
  $update : L \times N \to T\mathbf{1}$

  where their definitions are explained below.

  $cond(true,a,b) = a$
  $cond(false,a,b) = b$

  $plus$ = the usual addition operation
  lt = the usual less-than operation

  $$
  \begin{array}{l}
  new\ n\ s = \underline{let}\ l = Select(s) \\
  \qquad\qquad\quad \underline{in}\ (s[l \mapsto n], l) \\
  \qquad\qquad\quad \underline{end}
  \end{array}
  $$

  $lookup\ l\ s = (s, s(l))$
  $update\ (l,n)\ s = (s,[l \mapsto n], *)$

Remark: We can check that $\mathcal{M}_{Select}$ is a CCC with a strong monad. Throughout the rest of the thesis, we fix a selection function $Select$ for the interpretation function $[\![\ ]\!]$.

[Sta94] has a categorical model that captures many essential aspects of names in iML. The model gives rise to the same computational metalanguage $\mathrm{CML}_{\mathrm{iML}}$

(see Chapter 4), but it induces a stronger equational logic which can equate some additional programs which use references.

4.1.3.3 Interpreting CML$_{\mathrm{iML}}$

The interpretation of CML$_{\mathrm{iML}}$ into the store model is quite standard. The following is the interpretation of the type expressions.

$$
\begin{array}{rcl}
[\![\texttt{int}]\!] & = & N \\
[\![\texttt{unit}]\!] & = & \mathbf{1} \\
[\![\texttt{bool}]\!] & = & B \\
[\![\texttt{int ref}]\!] & = & L \\
[\![\sigma\to\tau]\!] & = & [\![\sigma]\!]\Rightarrow[\![\tau]\!] \\
[\![T\sigma]\!] & = & T[\![\sigma]\!]
\end{array}
$$

Where $B = \{true, false\}$. A context $\Gamma = \{x_1 : \sigma_1, ...x_n : \sigma_n\}$ is interpreted as the product $[\![\sigma_1]\!] \times ... \times [\![\sigma_n]\!]$. A term in context $\Gamma \vdash M{:}\sigma$ is interpreted as a morphism $[\![\Gamma]\!]\xrightarrow{m}[\![\sigma]\!]$. The rules for interpreting CML$_{\mathrm{iML}}$ are given below.

$$\frac{}{\Gamma \vdash x{:}\sigma} \mapsto \frac{}{[\![\Gamma]\!]\xrightarrow{\pi_x}[\![\sigma]\!]}$$

$$\frac{}{\Gamma \vdash \texttt{()} \ : \texttt{unit}} \mapsto \frac{}{[\![\Gamma]\!]\xrightarrow{!}\mathbf{1}}$$

$$\frac{\Gamma \vdash M_1 \ : \texttt{bool} \quad \Gamma \vdash M_2 \ : \sigma \quad \Gamma \vdash M_3 \ : \sigma}{\Gamma \vdash \texttt{cond}(M_1,M_2,M_3) \ {:}\sigma} \mapsto \frac{[\![\Gamma]\!]\xrightarrow{b}B \quad [\![\Gamma]\!]\xrightarrow{m}[\![\sigma]\!] \quad [\![\Gamma]\!]\xrightarrow{n}[\![\sigma]\!]}{[\![\Gamma]\!]\xrightarrow{<b,m,n>}B \times [\![\sigma]\!] \times [\![\sigma]\!]\xrightarrow{cond_{[\![\sigma]\!]}}[\![\sigma]\!]}$$

$$\frac{\Gamma \vdash M_1 \ : \texttt{int} \quad \Gamma \vdash M_2 \ : \texttt{int}}{\Gamma \vdash \texttt{plus}(M_1, M_2) \ : \texttt{int}} \mapsto \frac{[\![\Gamma]\!]\xrightarrow{a}N \quad [\![\Gamma]\!]\xrightarrow{b}N}{[\![\Gamma]\!]\xrightarrow{<a,b>}N \times N\xrightarrow{plus}N}$$

$$\frac{\Gamma \vdash M_1 \ : \texttt{int} \quad \Gamma \vdash M_2 \ : \texttt{int}}{\Gamma \vdash \texttt{<}(M_1, M_2) \ : \texttt{bool}} \mapsto \frac{[\![\Gamma]\!]\xrightarrow{a}N \quad [\![\Gamma]\!]\xrightarrow{b}N}{[\![\Gamma]\!]\xrightarrow{<a,b>}N \times N\xrightarrow{lt}B}$$

$$\frac{\Gamma \vdash M \ : \texttt{int}}{\Gamma \vdash \texttt{new}(M) \ {:}T(\texttt{int ref})} \mapsto \frac{[\![\Gamma]\!]\xrightarrow{m}N}{[\![\Gamma]\!]\xrightarrow{m}N\xrightarrow{new}TL}$$

$$\frac{\Gamma \vdash M \ : \texttt{int ref}}{\Gamma \vdash \texttt{lookup}(M) \ : T\texttt{int}} \mapsto \frac{\Gamma\xrightarrow{l}L}{[\![\Gamma]\!]\xrightarrow{l}L\xrightarrow{lookup}TN}$$

$$\frac{\Gamma \vdash M \ : \texttt{int ref} \quad \Gamma \vdash N \ : \texttt{int}}{\Gamma \vdash \texttt{update}(M, N) \ : T\texttt{unit}} \mapsto \frac{[\![\Gamma]\!]\xrightarrow{l}L \quad [\![\Gamma]\!]\xrightarrow{a}N}{[\![\Gamma]\!]\xrightarrow{<l,a>}L \times N\xrightarrow{update}T\mathbf{1}}$$

$$\frac{\Gamma, x : \sigma \vdash M \; : \tau}{\Gamma \vdash \lambda x : \sigma.M \; : \sigma \to \tau} \mapsto \frac{\Gamma \times [\![\sigma]\!] \xrightarrow{f} [\![\tau]\!]}{[\![\Gamma]\!] \xrightarrow{curry(f)} [\![\sigma]\!] \Rightarrow [\![\tau]\!]}$$

$$\frac{\Gamma \vdash M \; : \sigma \to \tau \quad \Gamma \vdash N \; : \sigma}{\Gamma \vdash MN \; : \tau} \mapsto \frac{[\![\Gamma]\!] \xrightarrow{f} [\![\sigma]\!] \Rightarrow [\![\tau]\!] \quad [\![\Gamma]\!] \xrightarrow{x} [\![\sigma]\!]}{[\![\Gamma]\!] \xrightarrow{fx} [\![\tau]\!]}$$

$$\frac{\Gamma \vdash M \; : \sigma}{\Gamma \vdash [M] \; : T\sigma} \mapsto \frac{[\![\Gamma]\!] \xrightarrow{x} [\![\sigma]\!]}{[\![\Gamma]\!] \xrightarrow{x} [\![\sigma]\!] \xrightarrow{\eta_{[\![\sigma]\!]}} T[\![\sigma]\!]}$$

$$\frac{\Gamma \vdash M \; : T\sigma \quad \Gamma, x : \sigma \vdash N \; : T\tau}{\Gamma \vdash \texttt{let } x \Leftarrow M \texttt{ in } N \; : \tau} \mapsto \frac{[\![\Gamma]\!] \xrightarrow{m} T[\![\sigma]\!] \quad [\![\Gamma]\!] \xrightarrow{n} T[\![\tau]\!]}{[\![\Gamma]\!] \xrightarrow{<1,m>} [\![\Gamma]\!] \times T[\![\sigma]\!] \xrightarrow{n^*} T[\![\tau]\!]}$$

4.1.3.4 Interpreting iML

As in the previous section, there are three steps in interpreting the language: interpreting type expressions, interpreting type environments, and interpreting iML terms.

The interpretation of type expressions is almost identical except for function types, where the codomain is of computation type. This is the case because at the level of $\mathrm{CML}_{\mathrm{iML}}$ we want to make the notion of computation explicit.

$$\begin{aligned} (\![\texttt{int}]\!) &= \texttt{int} \\ (\![\texttt{unit}]\!) &= \texttt{unit} \\ (\![\texttt{bool}]\!) &= \texttt{bool} \\ (\![\texttt{int ref}]\!) &= \texttt{int ref} \\ (\![\sigma \to \tau]\!) &= (\![\sigma]\!) \to T(\![\tau]\!) \end{aligned}$$

The interpretation of the type environment $\Gamma = \{x_1 : \sigma_1, ..., x_n : \sigma_n\}$ is done in a standard way.

$(\![\Gamma]\!) = \{x_1 : (\![\sigma_1]\!), ..., x_n : (\![\sigma_n]\!)\}$

iML terms are interpreted directly into $\mathrm{CML}_{\mathrm{iML}}$ terms. The interpretation is divided into interpreting iML canonical forms (denoted by $|\;| : \texttt{Can} \to [\![\sigma]\!]$) and interpreting iML expressions (denoted by $(\![\;]\!) : \texttt{Exp} \to T[\![\sigma]\!]$).

Canonical forms:

$$
\begin{aligned}
\mid x \mid &= x \\
\mid a \mid &= a \\
\mid l \mid &= l \\
\mid \lambda x : \sigma . M \mid &= \lambda x : (\![\sigma]\!).(\![M]\!)
\end{aligned}
$$

where the variable l ranges over locations.
Expressions:

$$
\begin{array}{rcl}
(\![C]\!) & = & [\mid C \mid] \\
(\![\mathtt{if}\ M_1\ \mathtt{then}\ M_2\ \mathtt{else}\ M_3]\!) & = & \mathtt{let\ x} = (\![M_1]\!)\ \mathtt{in\ cond}(\mathtt{x},(\![M_2]\!),(\![M_3]\!)) \\
(\![M_1 + M_2]\!) & = & \mathtt{let\ x} = (\![M_1]\!)\ \mathtt{in\ let\ y} = (\![M_2]\!)\ \mathtt{in}\ [\mathtt{plus}(\mathtt{x},\mathtt{y})] \\
(\![M_1 < M_2]\!) & = & \mathtt{let\ x} = (\![M_1]\!)\ \mathtt{in\ let\ y} = (\![M_2]\!)\ \mathtt{in}\ [<(\mathtt{x},\mathtt{y})] \\
(\![\mathtt{ref}\ M]\!) & = & \mathtt{let\ x} = (\![M]\!)\ \mathtt{in\ new}(\mathtt{x}) \\
(\![\,!M]\!) & = & \mathtt{let\ l} = (\![M]\!)\ \mathtt{in\ lookup}(\mathtt{l}) \\
(\![M\ := N]\!) & = & \mathtt{let\ l} = (\![M]\!)\ \mathtt{in\ let\ x} = (\![N]\!)\ \mathtt{in\ update}(\mathtt{l},\mathtt{x}) \\
(\![MN]\!) & = & \mathtt{let\ f} = (\![M]\!)\ \mathtt{in\ let\ x} = (\![N]\!)\ \mathtt{in\ f\ x}
\end{array}
$$

Note that the [ ] in $[\mid C \mid]$ denotes a $\mathrm{CML}_{\mathrm{iML}}$ construct lifting a value to a computation (see page 46).

The translation $(\![\ ]\!)$ is adequate with respect to the type systems. Formally, it is expressed by Proposition 4.1.4

Proposition 4.1.4. *For all* iML *term M,*
$\Gamma \vdash M : \sigma$ *iff* $(\![\Gamma]\!) \vdash (\![M]\!) : T(\![\sigma]\!)$.

Proof: By induction over the structure of the type derivation of M. ∎

4.2 Computational invariant

We aim to study constant computations in denotational setting. Since our model defined in the previous section is a simple sets and functions model, it does not capture a lot of computational aspects of programs[2]. In terms of side effects, it does not give a structure for distinguishing constant programs and nonconstant programs. As an explanation, consider the following iML program:

```
val M = let
          val x = ref 0
        in
          λn.  x:=!x+2; if even(!x) then n
                        else !x
        end
```

[2]For example, it is not fully abstract at first order types.

We are interested in the interpretation m of `M`. Below we give the typing of m.

$$\begin{aligned} m = [\![\texttt{M}]\!] &\in T[\![\texttt{int}{\to}\texttt{int}]\!] \\ &\in S{\to}S{\times}(N{\to}TN) \end{aligned}$$

The question we like to ask is: in what sense `M` and m are constant.

From the computation point of view the program `M` behaves like an identity program `id`(where `id` is the usual definition of an identity program). Technically, we say that `M` $\cong$ `id` (read: `M` is observationally equivalent to `id`).

At the denotational level, the definition of being constant is more intricate. A naive definition which requires m to be equal to $[\![\texttt{id}]\!]$ (call it id) is too restrictive. In fact, m is not equal to id since they differ on how they process the current state.

This suggests that we need an additional structure; one that enables us to assert that m is constant and is equal to id in some sense.

Let us consider a general case of a function $f{\in}A{\to}TB$ and see in what sense f can be considered to have a constant computation behaviour. We can view f as its uncurried version

$f \in S{\times}A{\to}S{\times}B.$

This says that the behaviour of the input of f not only depends on the input (of type A), but also on the current state[3]. Moreover, the output of f is a pair of an after-state and an output value (of type B), but the value bindable by programmers are only the output value[4].

A series of operations will induce a transition system on states with the initial state of the transition system being the initial state of the series. We can also consider a transition system induced by all possible series of operations involving f.

Going back to our earlier example, if we let $(s', f) = m\ s$, and let l_x the local variable in f, then we know that f preserves the evenness of x in the sense that

$$\forall s. \textit{if}\ \ (s(l_x))\ \textit{is even then}\ ((f(s,n))_1\ l_x)\ \textit{is even.}$$

Moreover, since we know that l_x is initialised to 0, the relevant transition system R is one that starts with l_x mapped to 0 and preserves the evenness of l_x. We can express R as follows:

$$\begin{aligned} T(s,s') &\iff s(l_x) + 2 = s'(l_x) \\ R(s,s') &\iff T^*([l_x{\mapsto}0], s) \wedge T^*(s,s') \qquad\qquad (\text{Defn-}R) \end{aligned}$$

[3]Notice that programmers have a restricted access to modify the current state.

[4]In other words, programmers cannot directly observe the after-state.

Where T^* is the transitive reflexive closure of T. The assertion $T^*([l_x \mapsto 0], s)$ means that s is reachable from the state $[l_x \mapsto 0]$ via T.[5]

Let $(s_2, \mathit{fid}) = id\ s$. We will use the uncurried version of *fid*. Now we can have a definition for asserting f and *fid* equivalent in the following sense:

a. for all $s \in dom\ R$*; for all* $n \in N$*:*

$R(s, (f\ (s,n))_1) \wedge R(s, (fid\ (s,n))_1)$

b. for all $s \in dom\ R$*; for all* $n \in N$*:*

$(f\ (s,n))_2 = (fid\ (s,n))_2$

Assertion (a) says that each of the input-output pairs of states of f and *fid* are related by R. Assertion (b) says that the bindable (or observable) outputs of f and *fid* always match. Notice that these assertions are quantified over 'possible states in computation' which is expressed by *dom R*.

The transition system R defined in (Defn-R) also has a way of expressing the fact that f is constant in the following sense:

Let $s \in dom\ R$. Let $(s_1, k) = f\ (s, n)$. Consider any s_2 such that $R(s_1, s_2)$. Let $(s_3, k_2) = f\ (s_2, n)$. Then $k_2 = k$.

Intuitively it says that after evaluating $f(s,n)$ we get (s_1, k). If f is part of a big computation, the computation may do some other operations (which may have side effects) before calling f the next time. Supposing the computation creates side effects by changing s_1 into s_2, and provided s_1 is related to s_2 via R, then the input-output behaviour of f the next time it is called is the same as the input-output behaviour of f the last time it is called.

In effect, the transition system R gives the context a scope for modifying the current states within the boundary of R. A series of computation which uses f will still maintain the property that f is constant as long as each state transition in the computation satisfies R. In this respect, we also call R *computational invariant.*

The purpose of a transition system serves as a context in which we execute an operation. In effect, what we are doing is we observe the behaviour of an operation only with respect to a particular or a class of contexts. There are different ways of choosing the structures of contexts, most of them are at the operational level (for example, see [Fel87, MT92, PS98] for definitions of evaluation contexts). We

[5] A bigger transition relation $R' \supseteq R$ is also applicable:

$$R'(s, s') \iff T^*([l_x \mapsto 0], s) \wedge T^*([l_x \mapsto 0], s')$$

choose to use transition systems because we are interested in the historical effects of a series of operations.

The following is a standard definition of transition system (also called transition relation) adopted from [Sif82b].

Definition 4.2.1. *A transition system R is a pair $(S, \rightarrow)$, where S is a set of states and $\rightarrow$ is a nonempty binary relation on S.*

We view the states of a transition system as memory stores and the transition $\rightarrow$ as the possible store changes that we are interested in. $s \rightarrow s'$ means it is possible for the state s to be updated into s'. Another terminology says that s' is *reachable* from s. We use $R(s, s')$ to denote $s \rightarrow s'$.

We use the following notations.

$$
\begin{aligned}
dom\ R\ &= \{s{\in}S \mid \exists s' \in S.R(s, s')\} \\
ran\ R\ &= \{s'{\in}S \mid \exists s \in S.R(s, s')\}.
\end{aligned}
$$

We call a transition sytem R *regular* if R is reflexive on its domain, transitive, and $ran\ R \subseteq dom\ R$. From now on we only consider regular transition systems.

One useful aspect of transition system is that they can capture the notion of reachable states. Given a transition system R and a state $s{\in}S$, we can define a notion of reachable states:

Definition 4.2.2. $reachable_s^R = \{s' \in S \mid R(s, s')\}$

When it is clear from context, we would omit the superscript R. Sometimes in the beginning of a computation we have little information about how the computation would behave — in particular how it would behave to the current store. As the computation progresses, we gain more information about the computation and could give a more determined characteristics about the computation. This is the case when a program creates a side effect by allocating and initialising a new reference. As the definition of *new* on page 49 shows, there is a choice in picking a new location. Such a computation needs to be analysed in a more general context in the beginning of the computation and gradually the context would narrow down to the possible relevant transitions as the computation goes along. This means before the allocation we do not know which new location would be allocated and we should start with a more general transition relation, and after the allocation we can narrow down our transition relation into the relevant possible paths.

We define the concept of narrowing down or restricting a transition relation R with respects to a set of states $A \subseteq S$.

Definition 4.2.3. $A \triangleleft R = \{(s, s') \in R \mid s \in A\}$

We have the following proposition.

Proposition 4.2.4. *dom* $(A \triangleleft R) \subseteq A$

Notice that the converse is not true. However, it is true when $A \subseteq$ *dom* R.

Proposition 4.2.5. *If* $A \subseteq$ *dom* R *then dom* $(A \triangleleft R) = A$.

The last two propositions are special cases of the following:

Proposition 4.2.6. *Let* $R' = A \triangleleft R$. *Then we have:*
dom $R' = A \cap$ *dom* R.

Proof: $\subseteq$ case: Let $x \in$ *dom* R'. ie. $x \in$ *dom* $A \triangleleft R$. By Proposition 4.2.4, we have $x \in A$. By the previous result and our assumption, there exists y such that $(x,y) \in A \subseteq R$. In other words, $x \in$ *dom* R.
$\supseteq$ case: Suppose $x \in A$ (*1) and $x \in$ *dom* R (*2). By (*2), there exists z such that $(x,z) \in$ R. By the previous result, by (*1), and by the definition of $\triangleleft$, we have $(x,z) \in A \triangleleft R$. In other words, $x \in$ *dom* $A \triangleleft R$. ∎

This gives us a useful corollary:

Corollary 4.2.7. *Let* $R' = (reachable_s) \triangleleft R$. *Then we have:*
dom $R' = (reachable_s) \cap$ *dom* R.

There are useful properties relating a reachable state with a transitive and reflexive relation.

Proposition 4.2.8. *If* R *is a transitive and reflexive relation and* $s \in$ *dom* R, *then:*
$reachable_s \triangleleft R$ *is transtive and reflexive.*

Proof: (reflexive:) Consider $x \in$ *dom* R'. By Proposition 4.2.7, $x \in (reachable_s) \cap$ *dom* R. By reflexivity of $R$and $x \in (reachable_s)$, we have $x \in (reachable_s) \triangleleft R$. Hence $(x,x) \in R'$.
(transitive:) Consider $(x,y) \in R'$ and $(y,z) \in R'$. By $R' \subseteq R$ and the transivity of R, we have $(x,z) \in R$. By $x \in reachable_s$ we have $(x,z) \in reachable_s \triangleleft R$. ∎

Proposition 4.2.9. *If R is regular and $s \in dom\ R$, then $reachable_s \lhd R$ is regular.*

4.3 Being constant throughout a transition system

4.3.1 Background

The purpose of this section is to give a uniform definition of being constant with respect to a transition system. The definition should be uniform in the sense that it is general enough to cover all interpretations of iML terms, but it should also be simple. Moreover, because our main motivation of the thesis is to find a subset of iML terms which behave in an applicative way, it is desirable to have a semantic definition which would easily induce a type system or a reasoning principle about being-constant. For higher-order functional languages, logical relation [Mit90] is such a method.

4.3.1.1 Logical relations

A logical relation is a family $\{R^\sigma\}$ of typed relations with the relation $R^{\sigma\to\tau}$ for type $\sigma \to \tau$ defined in terms of the relations R^σ and R^τ. It is defined over typed applicative structures whose definitions is given below.

Definition 4.3.1 ([Mit90]). *A typed applicative structure $\mathcal{A}$ for signature Σ is a tuple*

$$< \{A^\sigma\}, \{\mathbf{App}^{\sigma,\tau}\}, Const >$$

of families of sets and mappings indexed by type expressions over the type constants from Σ. It has the following characteristics:
a. A^σ is a set.
b. $\mathbf{App}^{\sigma,\tau}$ is a function $\mathbf{App}^{\sigma,\tau} : A^{\sigma\to\tau} \to A^\sigma \to A^\tau$
c. Const is a mapping from term constants of Σ to elements of the appropriate A^σ's.

The programming language iML can be viewed as a typed applicative structure with A^σ the set of closed terms of type σ and **App** the usual program application.

Formally, the definition of binary logical relation over two applicative structures is the following.

Definition 4.3.2 ([Mit90]). *Let $\mathcal{A} = < \{A^\sigma\}, \{\mathbf{App}_{\mathcal{A}}^{\sigma,\tau}\}, Const_{\mathcal{A}} >$ and $\mathcal{B} = < \{B^\sigma\}, \{\mathbf{App}_{\mathcal{B}}^{\sigma,\tau}\}, Const_{\mathcal{B}} >$ be applicative structures for some signature Σ. A logical relation $\mathcal{P} = \{P^\sigma\}$ over $\mathcal{A}$ and $\mathcal{B}$ is a family of relations indexed by the* *type expressions over* Σ *such that:*

- $P^{\sigma} \subseteq A^{\sigma} \times B^{\sigma}$ *for each type* σ,
- $P^{\sigma \to \tau}(f, g)$ *iff* $\forall x \in A^{\sigma}.\forall y \in B^{\sigma}.P^{\sigma}(x, y) \Rightarrow P^{\tau}(\mathbf{App}_{\mathcal{A}} f\ x, \mathbf{App}_{\mathcal{B}} g\ y)$,
- $P^{\sigma}(Const_{\mathcal{A}}(c), Const_{\mathcal{B}}(c))$ *for every typed constant* $c{:}\sigma$ *of* Σ.

The above definition can be adopted to unary predicates by rewriting the second point as

$P^{\sigma \to \tau}(f)$ iff $\forall x \in A^{\sigma}.P^{\sigma}(x) \Rightarrow P^{\tau}(\mathbf{App}_{\mathcal{A}} f\ x)$.

Notice that we will borrow the idea of the definition but will not strictly use the above definition since the predicate that we want to define (being-constant throughout R) does not satisfy the third point. For example, we do not want to have the assignment operator as being constant for any possible transition system R. However, the first two points are sufficient for guaranteeing closure under application when the function and the argument satisfy the predicates.

4.3.1.2 Logical relations on values and computations

The second issue in giving the definition of being constant throughout a transition system R is to adopt the logical relation method in values-computations setting. With a term in context $\Gamma \vdash M : \tau$ (where $\Gamma = \{x_1 : \sigma_1, ..., x_n : \sigma_n\}$) is interpreted as a function $[\![\sigma_1]\!] \times ... \times [\![\sigma_n]\!] \xrightarrow{m} T[\![\tau]\!]$, we need to have a family of relations $\{V^{\sigma_i}\}$ for handling values and a family of relations $\{C^{\tau}\}$ for handling computations.

To simplify our explanation, consider the case $n = 1$. Then a logical relation $P^{\sigma_1 \to \tau}$ which takes m would have the following format:

$P^{\sigma_1 \to \tau}(m)$ *iff for all* $a \in [\![\sigma_1]\!] : V^{\sigma_1}(a) \Rightarrow C^{\tau}(m\ a)$.

This is a natural modified definition of logical relation in monadic semantics. This method is used by [Cen96] for proving adequacy result on a tiny fragment of ML (TMLE) with exceptions. He defines two approximation relations:

$$\begin{aligned} \leq_{\sigma} &\subseteq [\![\sigma]\!] \times Can_{\sigma} \\ \preceq_{\sigma} &\subseteq T[\![\tau]\!] \times Exp_{\tau} \end{aligned}$$

where T is the exception monad, Can_{σ} is the strongly canonical TMLE terms of type σ, and Exp_{τ} is the closed TMLE terms of type τ. A relation $m \preceq_{\sigma} M$ means the program M terminates if the mathematical expression m denotes. Approximation at function type is defined as:

$$f \leq_{\sigma \to \tau} (\texttt{fn x} : \sigma \Rightarrow e) \text{ iff } f(d) \preceq_{\tau} [b/\texttt{x}]e \text{ for all } d \text{ and } b \text{ such that } d \leq_{\sigma} b$$

and the definition of $m \preceq_\tau e$ uses the following clause:

$$\text{if } m = val(d), \text{then } e \rightsquigarrow b, \text{ for some } b \text{ such that } d \leq_\tau b.$$

Notice that $\leq_{\sigma \to \tau}$ is defined in terms of $\leq_\sigma$ and $\preceq_\tau$, and $\preceq_\tau$ is defined in terms of $\leq_\tau$.

4.3.1.3 Indistinguishability relations

The third issue in giving the definition of being constant throughout a transition system R is about the need to have another logical relation. In analysing a computational phenomena, it is natural to define a binary relation for expressing indistinguishability. In our setting, we are interested in defining a notion of indistinguishability with respect to contexts that satisfy a transition system R. The purpose of the indistinguishability relation is to equate two denotations a and b which are not the same at the sets and functions level, but they are observationally indistinguishable within contexts that satisfy a transition system R. When this is the case, we say a and b are *indistinguishable throughout* R. We define a family of indistinguishability relations indexed by type expressions. They are defined in logical relation fashion.

In the above setting, a constant program with respect to R is one which is indistinguishable to itself with respect to R.

4.3.2 Definition

Figure 4.4 defines two families of binary relations

$$\{\textit{V-indistinguishable throughout}_\sigma\ R\} \tag{4.2}$$

$$\{\textit{C-indistinguishable throughout}_\sigma\ R\} \tag{4.3}$$

where R is a transition system. The definition is a standard logical relation definition in value-computation setting. The subtletly lies in the computation type.

The first condition in m, n *C-indistinguishable throughout*$_\sigma$ R says that their side effects have to satisfy R. The second condition says that their value parts are *V-indistinguishable throughout*$_\sigma$ R when their current states are in *dom* R.

Note that two locations l and k are *V-indistinguishable throughout*$_{\texttt{ref}}$ R if they are either never get allocated throughout R or they are both get allocated throughout R and they point to the same value.

Figure 4.5 defines the notion of being constant, which is a special case of the notion of indistinguishability.

a, b *V-indistinguishable throughout*$_{\texttt{int}}$ R
iff
$a = b$

l, k *V-indistinguishable throughout*$_{\texttt{ref}}$ R
iff
forall $s_i, s_j \in$ *dom* R : $s_i(l) = s_j(k)$

m, n *C-indistinguishable throughout*$_\sigma R$
iff
1. forall $s \in$ *dom* R : $R(s, (m\ s)_1) \wedge R(s, (n\ s)_1)$
2. <u>*let*</u> $\overline{A} = \{(m\ s)_2 \mid s \in dom\ R\} \cup \{(n\ s)_2 \mid s \in dom\ R\}$
<u>*in assert*</u>
forall $v, w \in \overline{A}$: v, w *V-indistinguishable throughout*$_\sigma$ R
<u>*end*</u>

f, g *V-indistinguishable throughout*$_{\sigma \to \tau} R$
iff
forall a, b *:*
a,b V-indistinguishable throughout$_\sigma R$
implies
(f a), (g b) C-indistinguishable throughout$_\tau R$*)*

Figure 4.4: $\{$*indistinguishable throughout*$_\sigma$ $R\}$. For brevity, we use `ref` to mean `int ref`.

x *V-const-throughout*$_\sigma R$
iff
x, x *V-indistinguishable throughout*$_\sigma R$

m *C-const-throughout*$_\sigma R$
iff
m, m *C-indistinguishable throughout*$_\sigma R$

Figure 4.5: $\{$*const-throughout*$_\sigma R\}$.

The definitions in Figure 4.4 and 4.5 are refinements of their counterparts in Chapter 2. A transition relation R can express a reachable set $< Q, s_0 >$ by setting $R = Q \times Q$. When $l, k \in L$ are both defined in Q, Definition 2.2.5 in Chapter 2 is the same as the definition of l, k *V-indistinguishable throughout*$_{\texttt{ref}}$ R in this chapter.

The following shows some properties of the definitions.

Proposition 4.3.3.
1. The relation V-indistinguishable throughout$_\sigma$ R is a partial equivalence relation over $[\![\sigma]\!]$
2. The relation C-indistinguishable throughout$_\sigma$ R is a partial equivalence relation over $T[\![\sigma]\!]$.

Corollary 4.3.4.
1. Let $\overline{A} = \{ a \in [\![\sigma]\!] \mid a$ V-const-throughout$_\sigma$ $R \}$. The relation V-indistinguishable throughout$_\sigma$ R is an equivalence relation over $\overline{A}$.
2. Let $\overline{A} = \{ a \in T[\![\sigma]\!] \mid a$ C-const-throughout$_\sigma$ $R \}$. The relation C-indistinguishable throughout$_\sigma$ R is an equivalence relation over $\overline{A}$.

The notion of *C-const-throughout*$_\sigma$ R is closed under function application. Formally,

Corollary 4.3.5.
If f V-const-throughout$_{\sigma \to \tau}$ R and
x V-const-throughout$_\sigma$ R,
then $(f\ x)$ C-const-throughout$_\tau$ R.

4.3.3 Examples

Example 4.3.6. Example 2.2.23 on page 30 can be easily expressed in our setting. We set $R = Q \times Q$, where Q is defined as (2.2) on page 31.

Example 4.3.7. The iML language can be easily extended to include pairing. Without loss of rigour, we can assume such construct and define the following ML program:

```
val M = let
          val x = ref 0
          fun f n = x := !x+2; if (even(x)) then n else (!x)
          fun g n = x := !x+4; if (even(x)) then (2*n) else (!x)
```

```
    in
      (f,g)
    end
```

We will use our semi-formal technique for reasoning about `M`. The expression `M` reduces to a canonical value of pair form.

$$s,\mathtt{M} \Downarrow (C_1, C_2), s[l_x \mapsto 0]$$

To say the meaning of (C_1, C_2) is *being-const throughout* R amounts to saying that the meanings of C_1 and C_2 are *being-const throughout* R. For $R = Q \times Q$ with Q defined as in the previous example, This is the case.

Example 4.3.8. This example analyses a memoised twice function[6]. Consider the following two programs.

```
fun twice n = 2*n

val mtwice =
let
  val x = ref 0
  val y = ref (twice 0)
  fun f n = if (n = !x) then (!y) else (x:=n; y:= (twice n); !y)
in
  f
end
```

We will use our semi-formal technique for showing `twice` indistinguishable to `mtwice`. Consider a current state $s \in S$. The expression `mtwice` reduces under s to the following.

$$s,\mathtt{mtwice} \Downarrow C, s[l_x \mapsto 0][l_y \mapsto 0]$$

The denotations of `twice` and C are the following.

$twice = [\![\mathtt{twice}]\!]$

[6] A standard example is a memoised factorial function. This is not possible in our framework since iML does not have recursion.

$mtwice = [\![C]\!]$

$twice\ n\ s = (s, 2 \times n)$

$$\begin{aligned}
mtwice\ n\ s = \ & \mathit{if}\ s(l_x) = n\ \mathit{then}\ (s, s(l_y)) \\
& \mathit{else}\ \underline{let}\ m = (twice\ n\ s[l_x \mapsto n]) \\
& \quad \underline{in} \\
& \quad\quad (s[l_x \mapsto n][l_y \mapsto m], m) \\
& \quad \underline{end}
\end{aligned}$$

Consider Q defined as

$$Q = \{s[l_x \mapsto n][l_y \mapsto 2 \times n] \mid s \in S, n \in N\}.$$

Let $R = Q \times Q$. Then we have *twice*,*mtwice* *V-indistinguishable throughout*$_{\texttt{int} \to \texttt{int}}$ R.

Example 4.3.9. This is a non-example. The definitions in Figure 4.4 and 4.5 still have limitations in dealing with dynamic allocations. Although the definition of location indistinguishability is an improvement of its counterpart in Chapter 2, it still cannot equate the following two programs.

```
val M = ref 8

val N = ((ref 7);(ref 8))
```

Let $m = [\![\texttt{M}]\!]$
$\quad n = [\![\texttt{N}]\!]$

We cannot find a regular transition system R such that they are *indistinguishable throughout* R. For the proof, consider $s \in dom\ R$. It is possible to have $(m\ s)$ yielding $(s[l \mapsto 8], l)$ and $(n\ s)$ yielding $(s[l \mapsto 7][k \mapsto 8], k)$. Let $s_2 = s[l \mapsto 7][k \mapsto 8]$. Then l and k are not *V-indistinguishable throughout* R since they are both defined in s_2 and $s_2 \in dom\ R$.

The next section provides a framework towards solving this problem.

4.4 Being constant within a transition system

This subsection gives an alternative definitions of being-constant and indistinguishable which are defined in previous section (Section 4.3). The purpose of the alternative definitions is motivated by the need to find a better indistinguishability relation which can handle dynamic allocations; in particular it should be able to handle Example 4.3.9.

4.4.1 Background

The idea of the new definitions lies in utilising the structures in transition systems for encoding some aspects of dynamic allocations and relevant future states. We explain the idea in terms of three points below.

- Recall that a transition system R represents the possible state changes that could occur in a class of contexts. The definition of *C-indistinguishable throughout* R in Figure 4.4 is based on quantifying the relevant states over *dom* R. This is too strong. Quantifying the behaviour of a location value produced by a location computation over (*dom* R) amounts to quantifying its behaviour relative to *all* possible states in the computations. A more appropriate setting is to prevent observing the behaviour of the location relative to its 'previous' states. This means given a location computation $m \in S \to S \times L$ and a current state $s_0 \in S$,

  let $(s_1, l) = m\ s_0$

  then we are interested in the behaviour of l relative to states that are reachable from s_1. Notice that this avoids the issue of undefined locations since if m is the denotation of an iML program, then l is defined in s_1 (iML cannot have a program that generates 'dangling' references).

- A transition system can express 'future possible states reachable from the current state'[7]. Continuing the above example, we are only interested in the behaviour of l over states that are reachable from s_1. Hence we can cut down a transition relation R into a relevant subset.

- Given *new* has a choice in picking up fresh locations, in comparing two computations that might generate locations we have to take this issue into account[8]. In other words, given two location computations $m, n \in S \to$

[7] This idea is of standard use in concurrency for expressing safety and liveness properties [Sif82a].

[8] This is the main reason why the definition in previous section fails to cope with Example 4.3.9.

$S \times L$ and a current state s, the characterisation of indistinguishability for $(m\ s)_2$ and $(n\ s)_2$ have to be compatible with the freedom to choose fresh locations. We decide to solve this by parameterising the indistinguishability relation over *pairs* of transition relations instead of just transition relations. In this setting, comparing two denotations m and n is done with m 'living' in a transition relation R_1 and n in R_2.

The next section gives the formal definitions.

4.4.2 Definition

Figure 4.6 defines two families of binary relations

$$\{\textit{V-indistinguishable within}_\sigma\ R_1, R_2\} \tag{4.4}$$

$$\{\textit{C-indistinguishable within}_\sigma\ R_1, R_2\} \tag{4.5}$$

Where R_1, R_2 are transition systems. The definition is a standard logical relation definition in value-computation setting. The definition is similar to *indistinguishable throughout* defined in Figure 4.4.

We would use the notation

(a within R_1) indistinguishable$_\sigma$ (b within R_2)

to mean *a,b indistinguishable within$_\sigma$* R_1,R_2.

The general difference between *indistinguishable within* and *indistinguishable throughout* is that we parameterise *indistinguishable within* over two transition systems whereas we parameterise *indistinguishable throughout* only over one transition system. When we look them in more detail, the difference lies in the `ref` and computation cases. *l,k V-indistinguishable within*$_{\texttt{ref}}$ R_1,R_2 requires that l has to be defined in R_1 and k has to be defined in R_2. The second condition in *m,n C-indistinguishable within$_\sigma$* R_1,R_2 says that their value parts are *V-indistinguishable within*$_{\texttt{ref}}$ R_1',R_2', where R_1' is a subset of transition system R_1, whose domain are states that are reachable from the after state obtained from evaluating m. Similarly for R_2'.

Figure 4.7 defines the notion of being constant, which is a special case of the notion of indistinguishability.

4.4.3 Example

We would like to validate the indistinguishability of m and n defined in Example 4.3.9. Their meanings are

$(a \ within \ R_1) \ V\text{-}indistinguishable_{\texttt{int}} \ (b \ within \ R_2)$
iff
$a = b$

$(l \ within \ R_1) \ V\text{-}indistinguishable_{\texttt{ref}} \ (k \ within \ R_2)$
iff
1. l *defined-in* R_1
2. k *defined-in* R_2
3. forall $s_i \in dom \ R_1, s_j \in dom \ R_2 : s_i(l) = s_j(k)$

$(m \ within \ R_1) \ C\text{-}indistinguishable_{\sigma} \ (n \ within \ R_2)$
iff
forall $s_i \in dom \ R_1, s_j \in dom \ R_2$:
<u>*let*</u>
$(s'_i, a) = m \ s_i$
$(s'_j, b) = n \ s_j$
<u>*in assert*</u>
1. $R_1(s_i, s'_i) \wedge R_2(s_j, s'_j)$
2. $(a \ within \ (reachable^{R_1}_{s'_i} \lhd R_1)) \ V\text{-}indistinguishable_{\sigma} \ (b \ within \ (reachable^{R_2}_{s'_j} \lhd R_2))$
<u>*end*</u>

$(f \ within \ R_1) \ V\text{-}indistinguishable_{\sigma \to \tau} \ (g \ within \ R_2)$
iff
forall a, b :
$(a \ within \ R_1) \ V\text{-}indistinguishable_{\sigma} \ (b \ within \ R_2)$
implies
$((f \ a) \ within \ R_1) \ C\text{-}indistinguishable_{\tau} \ ((g \ b) \ within \ R_2)$

Figure 4.6: $\{$*indistinguishable within*$_\sigma$ $R\}$. For brevity, we use `ref` to mean `int ref`.

$x \ V\text{-}const\text{-}within_{\sigma} R$
iff
$(x \ within \ R) \ V\text{-}indistinguishable_{\sigma} \ (x \ within \ R)$

$m \ C\text{-}const\text{-}within_{\sigma} R$
iff
$(m \ within \ R) \ C\text{-}indistinguishable_{\sigma} \ (m \ within \ R)$

Figure 4.7: $\{$*const-within*$_\sigma R\}$.

$$
\begin{aligned}
m\ s = &\ \underline{let}\ \ l = \mathit{Select}(s) \\
&\ \underline{in} \\
&\quad (s[l{\mapsto}8], l) \\
&\ \underline{end}
\end{aligned}
$$

$$
\begin{aligned}
n\ s = &\ \underline{let} \\
&\quad l = \mathit{Select}(s) \\
&\quad k = \mathit{Select}(s[l{\mapsto}7]) \\
&\ \underline{in} \\
&\quad (s[l{\mapsto}7][k{\mapsto}8], k) \\
&\ \underline{end}
\end{aligned}
$$

We need a transition system R for accomodating m. The specification is the following.

$$
\begin{aligned}
& R(s, s') \quad \mathit{iff} \\
& \exists l.s(l) = \mathit{Unused} \wedge s'(l) = 8 \wedge \\
& \quad \forall s'' s.t. R(s', s'') : s''(l) = 8.
\end{aligned}
$$

Notice that the definition is defined in terms of itself. We would construct such a transition system below.

$$
\begin{aligned}
& s_0 \text{ is the state where } \mathit{dom}\ s_0 = \{\} \\
& R_1 = \{(s_0, s_0[(\mathit{Select}\ s_0){\mapsto}8])\} \\
& R_{n+1} = \{(s, s[(\mathit{Select}\ s){\mapsto}8]) \mid s \in \mathit{ran}\ R_n\} \\
& R_\omega = \bigcup_{n=1}^{\infty} R_n \\
& R = \text{ the reflexive and transitive closure of } R_\omega.
\end{aligned}
$$

We need a transition system R' for accomodating n. The specification is the following.

$$
\begin{aligned}
& R'(s, s') \quad \mathit{iff} \\
& \exists l, k.s(l) = s(k) = \mathit{Unused} \wedge \\
& \quad s'(l) = 7 \wedge s'(k) = 8 \wedge \\
& \quad \forall s'' s.t. R'(s', s'') : s''(k) = 8.
\end{aligned}
$$

Such R' can be defined using previous technique, with the inductive part defined as:

$$
\begin{aligned}
& R_1 = \{(s_0, s_0[l{\mapsto}7][k{\mapsto}8]) \mid l = \mathit{Select}(s_0) \wedge k = \mathit{Select}(s_0[l \mapsto 7])\} \\
& R_{n+1} = \{(s, s[l{\mapsto}7][k{\mapsto}8]) \mid s \in \mathit{ran}\ R_n \wedge l = \mathit{Select}(s) \wedge k = \mathit{Select}(s[l \mapsto 7])\}.
\end{aligned}
$$

It is straightforward to verify that

$(m\ \mathit{within}\ R)\ C\text{-}\mathit{indistinguishable}_{\texttt{ref}}\ (n\ \mathit{within}\ R')$.

Since $(\mathit{dom}\ R) \cap (\mathit{dom}\ R') = \{s_0\}$, we also have

$(m\ \mathit{within}\ R_t)\ C\text{-}\mathit{indistinguishable}_{\texttt{ref}}\ (n\ \mathit{within}\ R_t)$

where $R_t = R \cup R'$.

4.4.4 Discussion

Notice that l,k *V-indistinguishable throughout*$_{\texttt{ref}}$ R does not imply $(l\ \mathit{within}\ R)$ *V-indistinguishable*$_{\texttt{ref}}$ $(k\ \mathit{within}\ R)$. The following is the sketch of the proof: consider $l, k \in L$ such that

$\forall s \in \mathit{dom}\ R. s(l) = s(k) = \mathit{Unused}.$

Then

$l, k\ V\text{-}\mathit{indistinguishable}\ \mathit{throughout}_{\texttt{ref}}\ R$

but not

$l, k\ V\text{-}\mathit{indistinguishable}\ \mathit{within}_{\texttt{ref}}\ R, R.$

Chapter 5

Case study: Queue module

Section 5.1 introduces general concept of abstract data types and modules as a way for structuring large programs. Section 5.2 describes the basic mechanism of modules in ML and the use of references in optimising an applicative module. It describes the issues of giving interpretations to ML modules and characterising equivalence between two interpretations. Section 5.3 shows three ML implementations of Queue module, two written without references and one written with references. section 5.4 provides a framework for showing equivalence between a pure implementation and an imperative implementation of Queue module. The idea is essentially based on the definition of *indistinguishable within* defined in Chapter 4. Section 5.5 gives the formalisation of the framework.

5.1 Background

When our job is to write small and simple programs, a simple programming language like iML defined in Chapter 3 is sufficient. It is a subset of the core part of Standard ML [MTH90]; the essence of the language is constructing functions, composing and applying them. A program is written as a list of functions, readily to be executed on the terminal.

When we shift from programming small codes into designing and integrating large systems, a bare typed lambda calculus programming language cannot support the process well. This is because we need abstractions in building and understanding large system. The main tool of abstraction is encapsulation — hiding the unnecessary details of a subsystem and providing an interface which provides information on the high level behaviour of the subsystem. This interface is what a programmer needs to know in order to use the system. One can treat it as a black box, and can pick other black boxes and combine them as one desires.

Enscapsulation is present in many programming language designed for building large, modular systems. For example:

- Modula 2 [Wir77, Wir88] which provides abstract data type (ADT) mechanism with its separate interface and implementation files,

- Common Lisp [Ste85, Gab89] which combines functional programming and object oriented features,

- Object-oriented languages (C++ [Str87], Eiffel [Mey88], Java [GJS97], and Smalltalk [PW88]) which use attributes, classes, and packages for hiding local functions and states, and

- ML families (Standard ML [MTH90], New-Jersey ML [AM91], and CAML [Ler93]) which have sophisticated module mechanisms which include signatures (the interface part), structures (the implementation part), functor (parameterised modules), and type sharing. On top of these they also handle polymorphism.

One use of encapsulation is in designing packages as independent entities to be included in libraries ready for use by programs that need them. The purpose of building libraries is to achieve a high degree of software reusability which implies cutting down duplicated work.

Modules and encapsulations are used extensively in real life projects. The following are some case studies on justifying the importance of modules in software system:

- Multics [CSC72] was a multi-user operating system developed in the 60s. The high level programming language PL/I was used in implementing the system which were carefully structured into modules that provided carefully controlled interfaces to the users. Out of the 1500 system modules, 250 were written in machine language for implementing the hardware dependent routines. Only a few modules were rewritten from PL/I to machine language for optimisation purposes. Some codes which were originally written in machine language were rewritten in PL/I for increasing their maintainability. The project was claimed to be a success and the choice of the programming language was the right one. Recalling their expericence, [CSC72] writes,

  > Not only has the cost of using a higher level language been acceptable, but increased maintainability of the software has permitted more rapid evolution of the system in response to development ideas as well as user needs.

- Ada vs C. [Zei97] compares the development costs of using C and Ada in developing commercial products. The products (called VADS) were software development tools for Ada, C, C++, and Ada95. The project started in March 1983 using C, then used Ada from 1986, and by mid 1991 the size of the Ada code had equalled the size of the C code. [Zei97] lists key elements in measuring the the costs of the development. Some of them were the bug/feature ratio, the training efforts to new programmers, the efforts in manually controlling the codes to insure that the codes were well behaved, and the accumulative cost. The resuts showed that Ada performed better on all the above aspects.

Even designers of specification languages (for example, Extended ML [KST94] and B-Notation [Abr91]) are aware that they need to have module mechanisms and ways of providing the high level view of a specification. There has been significant interest in using type theory [ML86, Luo94] as a general foundational notation for studying specification notation, implementation notation, modules and encapsulation (also see [Rus98]), and disciplined programming; the details are beyond the scope of this thesis.

The idea of encapsulation as a way of defining and implementing abstract data types (ADTs) is the heart of this section. In an ADT, the implementation is hidden and invisible from the outside program (the caller). The caller can use the ADT, but does not know how the ADT is implemented. For example, a stack abstract data type has `emptyStack`, `push`, and `pop` operations which represent the usual operations of stacks. Let us assume that the implementation satisfies the algebraic equations of stack properties. The stack ADT provides these operations to the users and does not reveal how the stack is actually implemented. The algebraic properties of the implementation are what the users need. The stack could be implemented using an array, a list, a pair of lists, or a pointer to a pair of lists. Because the underlying implementation of the data structure is hidden from its users, the implementation can easily be changed without affecting the programs that use it.

The studies of ADTs combine theoretical results of data structures with practical uses of modular programming. We can partition programming community into software builders who code efficient algorithms from journals or textbooks (For example, Algorithm and Data Structure by [AHU87]) and package them as off-the-shelf utilities, and programmers or analysts who work out the requirements of system and design the solutions in terms of the existing modules, thus avoiding thinking about the complexity of the actual implementation of the modules.

An abstract data type can provide applicative functions (as in the case of Stack, Queue, Hash, and finiteSet data types) or state-sensitive functions (as in the case of Counter datatype — it increments its values whenever it is called).

ADT research has concentrated on applicative packages since they are easily manipulated and reasoned. The main issues in this case is coding an efficient implementation of an abstract data type and showing that the new implementation is basically the same as the standard one. These issues are discussed in details in the following subsections. The abstract data type in question is Queue.

5.2 Modules in ML

This section is about an introduction to ML modules.

5.2.1 Signatures and structures

One main purpose of using ML module is for defining an abstract data type together with its various operations and concealing the type and the operations implementation details. The defining process is done with the signature construct and the implemetation process is done with structure construct. Consider the case where we want to define a set ADT with the operations `emptyset`, `addset`, and `memberset`. This example is taken from [Gil97]. In ML, we declare them with `signature` construct (see Figure 5.1). It says that we are interested in a module that contains the type `set` and operations `emptyset`, `addset`, and `memberset` (with their appropriate types). It only gives the information at the type level.

```
signature Set =
sig
  type ''a set
  val emptyset   :  ''a set
  val addset     :  ''a * ''a set → ''a set
  val memberset  :  ''a * ''a set → bool
end
```

Figure 5.1: The ML signature of Set data type

To implement the `Set` data type, we need the `structure` construct. An implementation involves deciding the representation of the stack and how the stack operations are implemented in terms of the manipulation of the representation. Figure 5.2 contains a simple implementation. We use a list as the representation

```
structure SetOne :  Set =
struct
  type ''a set = ''a list
  val emptyset = []
  val addset = op ::
  val memberset = ListUtils.member
end
```

Figure 5.2: A first implementation of Set data type

```
structure SetTwo :  Set =
struct
  type ''a set = ''a → bool
  fun emptyset _ = false
  fun addset (x,s) = fn e => e = x orelse s e
  fun memberset (x,s) = s x
end
```

Figure 5.3: A second implementation of Set data type

of a set, the `::` operation for inserting an element to a set, and the list membership operation for checking the set membership.

There are many ways of implementing `Set` data type. Figure 5.2 is one way, Figure 5.3 is another. Figure 5.3 uses boolean functions on elements of type `''a` for representing `set`.

We have two implementations of `Set` data type. How to show they are the same? It begs a question of what we mean by "the same". One answer is that both satisfy some algebraic properties of sets; an example of a collection of algebraic properties of sets is shown in Figure 5.4. This approach is discussed by [GTW78, EM85, EM90] in the issues of specifying and implementing abstract data types.

```
member (x,emptyset) = ff
member (x,addset(x,s)) = tt
member (x,addset(y,s)) = (x=y) ∨ member(x,s)
```

Figure 5.4: Algebraic propertis of Set Data Type

A second answer is that there is a relation between the first representation of the `Set` data type and the second representation of the `Set` data type such that the first implementation of the operations is related (via this relation) to the second implementation of the operations. In the case of `Set`, this definition is formalised in Figure 5.5. The key idea in the figure is the relation T. The relationship is visualised as commuting diagrams in Figure 5.6 (the diagram for `emptyset` is trivial, therefore it is omitted). The definition is defined in a logical relation style (see [Plo80, Mit90, Mit96]) where two functions are related if they send related inputs to related outputs.

Let $V =$ *the meaning of the variable type* `''a`

$B = [\![\texttt{bool}]\!]$

$S_1 = [\![\texttt{''a list}]\!] = V\ \mathit{list}$

$S_2 = [\![\texttt{''a} \rightarrow \texttt{bool}]\!] = V \rightarrow B$

$\mathit{valOfSetOne} \in S_1 \rightarrow \mathbb{P}V$

$\mathit{valOfSetOne}\ [x_1, ..., x_n] = \{x_1, ..., x_n\}$

$\mathit{valOfSetTwo} \in S_2 \rightarrow \mathbb{P}V$

$\mathit{valOfSetTwo}\ f = \{x | f\ x = tt\}$

$T\colon\ S_1 \leftrightarrow S_2$

$T(s_1, s_2)$ *iff* $\mathit{valOfSetOne}(s_1) = \mathit{valOfSetTwo}(s_2)$

The relations between the two implementations are the following:

Let $x \in V$:

$T([\![\texttt{SetOne.emptyset}]\!]\ ,\ [\![\texttt{SetTwo.emptyset}]\!])$

$T(s_1, s_2)$ *implies* $T([\![\texttt{SetOne.addset}]\!](x, s_1)\ ,\ [\![\texttt{SetTwo.addset}]\!](x, s_2))$

$T(s_1, s_2)$ *implies* $[\![\texttt{SetOne.memberset}]\!](x, s_1) = [\![\texttt{SetTwo.memberset}]\!](x, s_2)$

Figure 5.5: A relation T that relates two Set implementations

$$
\begin{tikzcd}
V \times S_1 \arrow[r, "{[\![\texttt{SetOne.addset}]\!]}"] \arrow[d, leftrightarrow, "id \times T"] & S_1 \arrow[d, leftrightarrow, "T"] \\
V \times S_2 \arrow[r, "{[\![\texttt{SetTwo.addset}]\!]}"] & S_2
\end{tikzcd}
\qquad
\begin{tikzcd}
V \times S_1 \arrow[r, "{[\![\texttt{SetOne.memberset}]\!]}"] \arrow[d, leftrightarrow, "id \times T"] & B \arrow[d, leftrightarrow, "id"] \\
V \times S_2 \arrow[r, "{[\![\texttt{SetTwo.memberset}]\!]}"] & B
\end{tikzcd}
$$

Figure 5.6: A diagramatic view of the relation R

A third answer, is that the two implementations are contextually equivalent. It means that for any ML programs using the `Set` signature, the behaviour of the program is the same whenever we use the first implementation or the second one. The essence of this definition is formalised for a language PCF (a typed lambda calculus with recursion) by Plotkin in [Plo77] and for nu-calculus (a typed lambda calculus with names) by Pitts and Stark [PS93, Sta94].

The second approach is usually studied at the denotational level; types are viewed as sets with some structures and programs are viewed as functions. The third approach is operational in nature; we define an operationa semantics of the language and test the equivalence on the operational behaviours of all possible program contexts.

There is a framework relating the second approach with the third one. If we have a denotational model which is adequate, then function equality of the meanings of two programs implies contextual equality. This framework exists for languages without module constructs such as PCF and nu-calculus. There is yet no frameworks for languages with modules since the studies of semantics of modules are still in reseach.

5.2.2 `ref` and optimising an applicative module

Most ADT research concentrates on applicative languages because because they have simple semantics. When an applicative library is heavily used, efforts are made to optimise the package; and one possible way is by using states that are visible within the package.

In the Standard ML New Jersey Library [AT 93], out of thirty-four structures or functors, nine use references(see Figure 5.7). Some of the references are used for implementing imperative data structures while others are used for optimising applicative data structures.

`dynamic-array.sml`	`1 functor`
`finalizer.sml`	`1 functor`
`hash-table.sml`	`1 functor`
`name.sml`	`1 structure`
`poly-hash-table.sml`	`1 structure`
`queue.sml`	`1 structure`
`random.sml`	`1 structure`
`splay-dict.sml`	`1 functor`
`splay-set.sml`	`1 functor`

Figure 5.7: Libraries that use `ref`

There is a step in converting some imperative data structures into applicative ones. For example, in the Hash library, we have:

```
insert :  '2a hash_table → (Key.hash_key * '2a) → unit
```

We can convert such function into an applicative one by passing the data structure to its output:

```
insert :  '2a hash_table → (Key.hash_key * '2a) → '2a hash_table
```

It should be emphasised that the actual step is not this naive. We have to take into consideration in what sense the hash implementation is applicative. For example, if a hash is implemented as a pointer, and the insert operation returns the same pointer, then we need the condition of single-threadedness for ensuring the applicative behaviour of the Hash library.

The next section is a specific example of an optimisation. We define Queue data structure, two implementations without `ref`, and a more efficient implementation with `ref`. We then prove that the third implementation is the same as the first two in the logical relation sense (see page 74).

5.3 Queue Module

5.3.1 Queue implementations

Queue is a versatile data structure. They are used in job spooling, for temporarily storing keyboard or mouse events, or as buffers. There are also other variants of Queue, including priority queues (or heap) and bounded queues. The essential operations are `enqueue` — adding an element to the end of a queue and `dequeue` — withdrawing an element from the head of the queue.

In this section, we define Queue data structure and three implementations. This example is taken from [Fou95]. For simplicity we take the basic operations only; other operations can be added without introducing further complexity to the implementations and the semantics.

```
signature QSIG =
sig
   type Q
   exception Q
   val empty :  Q
   val enqueue :  int * Q → Q
   val dequeue :  Q → int * Q
end;
```

Figure 5.8: ML signature for Queue.ml

We define them using ML modules. The `signature` part (see Figure 5.8) shows the type of the operators. The `structure` part (see Figure 5.9) consists of three implementations: `QueueOne`, `QueueTwo`, and `QueueThree`. The first one

```
structure QueueOne : QSIG =
struct
exception Q
type Q = int list
val empty = []
fun enqueue (a, q) = (a :: q)
fun dequeue [] = raise Q
  | dequeue q = let
                   val r = rev q
                in
                   (head r, rev (tail r))
                end
end;


structure QueueTwo : QSIG =
struct
exception Q
type Q = (int list * int list)
val empty = ([],[])
fun enqueue (a, (qin, qout)) = (a :: qin, qout)
fun dequeue (qin, h :: qout) = (h, (qin, qout))
  | dequeue ([],[]) = raise Q
  | dequeue (qin,[]) = dequeue ([], rev qin)
end;


structure QueueThree :QSIG =
struct
exception Q
type Q = (int list * int list) ref
val empty = ref([],[]) : Q
fun enqueue (a, ref (qin, qout)) = ref((a :: qin), qout)
fun dequeue (ref (qin, (h :: qout))) = (h, ref(qin, qout))
  | dequeue (ref ([],[])) = raise Q
  | dequeue (q as ref (qin,[])) = (q := ([], rev qin); dequeue q)
end;
```

Figure 5.9: ML structures for Queue.ml

is a naive implementation using a list for representing a queue. The second one uses a pair of lists, and the third one uses a reference to a pair of lists. `QueueOne` and `QueueTwo` are written without using references, whereas `QueueThree` with references.

5.3.2 Complexity analysis

We are interested in comparing the efficiency of the three implementations. Starting from `QueueOne`, the `enqueue` operation takes a constant time because we only use concatenation (::) operation. The `dequeue` operation takes the complexity of the `rev` operation, which is proportional to the length of the list. The deficiency of this implementation is that it is expensive to `dequeue` an element. This is because there is no ways of accessing the head of the queue directly.

A better implementation is `QueueTwo` which allows us to access the head and the end of the queue at an amortized constant time (subject to no copying of the queue). It uses a pair of lists (`qin`,`qout`), where the `qin` represents the end of the queue up to a middle of the queue, and `qout` represents the front of the queue up to continuation of the middle of the queue. The relationship between the representations in `QueueOne` and `QueueTwo` is:

```
q = qin @ (rev qout)
```

`QueueTwo` is more efficient because the `dequeue` operation does not always take n steps (where n is the length of the queue); in fact, `dequeue` does more work only when `qout` is empty (see the third line of the definition of `dequeue`.) When `dequeue` sees that `qout` is empty, it rearranges the queue by flushing out `qin` , transfer the queue to `qout` , and try the `dequeue` again. This avoids using the reverse function on the successive dequeuing.

The amortized complexity [Oka96, Oka95] of `QueueTwo` is $O(1)$. An amortized bound $O(f)$ is defined to be: for any sequence of n operations, the *total* running time of the sequence is bounded by $n \times O(f)$. There is an additional implicit condition on the nature of the sequence of operations. It requires that each data structure is used at most once.

It is easily shown that the amoritzed complexity of `QueueOne` is $O(n)$. Just consider n operations of `enqueue`s followed by n operations of `dequeue`s.

In some cases `QueueThree` performs better than `QueueTwo`. Consider the case where we start with an empty queue and do `enqueue` operation n times. Then we pass the queue to a computation that dequeue the same queue m times. `QueueThree` will do the reverse operation on the first dequeue and a head operation on the second and subsequent dequeues, whereas `QueueTwo` will do the reverse operations m times.

5.3.3 QueueOne equivalent to QueueTwo

In Figure 5.9 we present three implementations of the `Queue` data structure and assume that the implementations are logically equivalent (ie. their external behaviours are the same). One way to justify this is by showing the structure `QueueOne` is equivalent to `QueueTwo` and `QueueOne` is equivalent to `QueueThree`. The former is straightforward; the latter is more difficult since it involves reasoning about the creation of references and passing them as abstract data. This subsection contains the proof of `QueueOne` equivalent to `QueueTwo`.

Recall that a characterisation of an equivalence of two abstract data types is done by finding a relation T between the two representations such that the first implementation is related — via T — to the second implementation. Figure 5.6 in page 74 illustrates two commuting diagrams of such relation for `Set` data type. We use this characterisation for showing `QueueOne` equivalent to `QueueTwo`.

First of all we define the interpretations of `QueueOne` and `QueueTwo` in terms of sets and functions. For simplicity, let us ignore the case where it raises an exception (the addition of the exception clause can be added without changing the proof).

$$Queue_1 \in \mathit{int\ list}\ \times (N \times \mathit{int\ list}\ \rightarrow \mathit{int\ list})\ \times (\mathit{int\ list}\ \rightarrow N \times \mathit{int\ list})$$

Figure 5.10: The type of $Queue_1 = \{\![\texttt{QueueOne}]\!\}$

$$\begin{array}{ll}
(Queue_1\ s) = & \underline{let} \\
& \quad emp_1 = [\,] \\
& \quad enq_1\ (n, l) = (n{::}l) \\
& \quad deq_1\ l = \underline{let}\ (h{::}l') = rev\ l \\
& \quad\qquad\qquad \underline{in}(h, rev\ l') \\
& \quad\qquad\qquad \underline{end} \\
& \underline{in}\ (emp_1, enq_1, deq_1) \\
& \underline{end}
\end{array}$$

Figure 5.11: The interpretation of `QueueOne`

Notice that the mathematical meanings $Queue_1$ and $Queue_2$ are very similar to their definitions (which are `QueueOne` and `QueueTwo`). The type `int list` is interpreted as the space *int list* of finite lists of natural numbers and the type $\sigma{*}\tau$ is interpreted as the cartesian product of the interpretations of σ and τ.

$$
\begin{aligned}
Queue_2 \in\ & (int\ list \times int\ list) \times \\
& ((N \times (int\ list \times int\ list)) \rightarrow (int\ list \times int\ list)) \times \\
& ((int\ list \times int\ list) \rightarrow (N \times (int\ list \times int\ list)))
\end{aligned}
$$

Figure 5.12: The type of $Queue_2 = \{[\texttt{QueueTwo}]\}$

$$
\begin{aligned}
(Queue_2\ s) = \ & \underline{let} \\
& \quad emp_2 = ([\,],[\,]) \\
& \quad enq_2 = \text{...see Figure 5.14 ...} \\
& \quad deq_2 = \text{...see Figure 5.15 ...} \\
& \underline{in} \\
& \quad (emp_2, enq_2, deq_2) \\
& \underline{end}
\end{aligned}
$$

Figure 5.13: The interpretation of `QueueTwo`

$$
\begin{aligned}
enq_2\ (n,q) = \ & \underline{let} \\
& \quad (qin, qout) = q \\
& \underline{in} \\
& \quad (n{::}qin, qout) \\
& \underline{end}
\end{aligned}
$$

Figure 5.14: The definition of enq_2

$$
\begin{aligned}
deq_2\ q = \ & \underline{let} \\
& \quad (qin, qout) = q \\
& \quad f\ (qin, n{::}qout') = (n, (qin, qout')) \\
& \quad flush\ q = \underline{let} \\
& \quad\qquad\qquad (qin, qout) = q \\
& \quad\qquad\quad \underline{in} \\
& \quad\qquad\qquad ([\,], qout@(reverse\ qin)) \\
& \quad\qquad\quad \underline{end} \\
& \underline{in} \\
& if\ (qout \neq [\,])\ then\ f\ (qin, qout) \\
& else\ if\ (qout = [\,] \wedge qin = [\,])\ then\ error \\
& \text{else } \underline{let}\ q' = flush\ q \\
& \qquad \underline{in}\ f\ q' \\
& \quad \underline{end} \\
& \underline{end}
\end{aligned}
$$

Figure 5.15: The definition of deq_2

For simplicity, assume we have *reverse* (or *rev*), *head*, *tail*, and *cons* (or ::) operations at the semantics level. Also assume that we can do pattern matching.

Now we define a relation T between the meaning of the representation of `Q` in `QueueOne` and the meaning of the representation of `Q` in `QueueTwo`.

Definition 5.3.1. *The relation T is defined in Figure 5.16*

$$
\begin{aligned}
[\![\texttt{QueueOne.Q}]\!] &= [\![\texttt{int list}]\!] \\
&= \mathit{int\ list} \qquad\qquad \textit{(finite lists of natural numbers)} \\
[\![\texttt{QueueTwo.Q}]\!] &= [\![\texttt{int list*int list}]\!] \\
&= \mathit{int\ list} \times \mathit{int\ list}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{valOf} \in \mathit{int\ list}\ \times\ \mathit{int\ list}\ \rightarrow \mathit{int\ list} \\
&\mathit{valOf}\,(\mathit{qin}, \mathit{qout}) = \mathit{qin}\ @\ (\mathit{rev\ qout}) \\
&T : [\![\texttt{QueueOne.Q}]\!] \leftrightarrow [\![\texttt{QueueTwo.Q}]\!] \\
&T : \mathit{int\ list} \leftrightarrow \mathit{int\ list} \times \mathit{int\ list} \\
&T(q, (\mathit{qin}, \mathit{qout}))\ \ \mathit{iff}\ \ q = \mathit{valOf}(\mathit{qin}, \mathit{qout})
\end{aligned}
$$

Figure 5.16: The relation T between the meanings of the representation of `QueueOne.Q` and `QueueTwo.Q`

Intuitively, the relation T acts as a "converter" between the representation of `Q` in the world of module `QueueOne` and the representation of `Q` in the world of module `QueueTwo`. When an abstract data q in $Queue_1$ is converted via T to an abstract data $(qin, qout)$ in $Queue_2$ (and vice versa), its logical behaviour is preserved.

Theorem 5.3.2. *The diagrams in Figure 5.17 commute.*

Proof:

<u>Case:</u> *enq*.

Let $a \in N$,

   $q \in$ *int list*, and

   $(qin, qout) \in$ *int list* $\times$ *int list*.

Assume that $T(q, (qin, qout))$. In other words,

  $q = qin\ @\ (rev\ qout)$.

Let $q' = enq_1(a, q)$

   $(qin', qout') = enq_2(a, (qin, qout))$.

$$
\begin{tikzcd}
N \times (int\ list) \arrow[r, "enq_1"] \arrow[d, leftrightarrow, "id \times T"] & (int\ list) \arrow[d, leftrightarrow, "T"] \\
N \times (int\ list \times int\ list) \arrow[r, "enq_2"] & (int\ list \times int\ list)
\end{tikzcd}
$$

$$
\begin{tikzcd}
(int\ list) \arrow[r, "deq_1"] \arrow[d, leftrightarrow, "T"] & N \times (int\ list) \arrow[d, leftrightarrow, "id \times T"] \\
(int\ list \times int\ list) \arrow[r, "deq_2"] & N \times (int\ list \times int\ list)
\end{tikzcd}
$$

Figure 5.17: The diagrams relating $Queue_1$ and $Queue_2$

We have

$qin' = a :: qin$

$qout' = qout.$

Proving the correspondence:

$q' = a :: q$ by enq_1

$= a :: (qin \mathbin{@} rev\ qout)$ by assumption

$= (a :: qin) \mathbin{@} rev\ qout$ rearrange bracket

$= qin' \mathbin{@} rev\ qout'$ by enq_2.

In other words, $T(q', (qin', qout'))$. ∎

<u>Case:</u> *deq*.

Let $q \in int\ list$ and

$(qin, qout) \in int\ list\ \times\ int\ list.$

Assume that $T(q, (qin, qout))$. In other words,

$q = qin \mathbin{@} (rev\ qout).$

We have two cases.

Consider the case $qout \neq [\,]$.

Let $(h :: out) = qout$.

Let $(a, q') = deq_1\ q$

$(b, (qin', qout')) = deq_2(qin, qout).$

By deq_2, we have

$b = h$
$qin' = qin$
$qout' = out.$

Proving the correspondence:

$$
\begin{aligned}
q' &= rev(tail(rev(q))) && \text{by } deq_1 \\
&= rev.tail.rev\ (qin\ @\ (rev\ qout)) && \text{by assumption} \\
&= rev.tail\ (qout\ @\ (rev\ qin)) && \text{by } rev \\
&= rev\ ((tail\ qout)\ @(rev\ qin)) && \text{rearranging bracket} \\
&= qin\ @\ (rev\ (tail\ qout)) && \text{by } rev \\
&= qin\ @\ (rev\ out)) && \text{by } qout = h :: out \\
&= qin'\ @\ (rev\ qout')) && \text{by } deq_2.
\end{aligned}
$$

To prove $a = b$, we have

$$
\begin{aligned}
a &= head.rev\ q && \text{by } deq_1 \\
&= head.rev\ (qin\ @\ (rev\ qout)) && \text{by assumption} \\
&= head\ (qout\ @\ (rev\ qin)) && \text{by } rev \\
&= head\ ((h :: out)\ @\ (rev\ qin)) && \text{by } qout = h :: out \\
&= h && \text{by } head \\
&= b && \text{by } deq_2.
\end{aligned}
$$

Consider the case $qin \neq [\,] \wedge qout\ = [\,]$.

Let $(a, q') = deq_1\ q$
$(b, (qin', qout')) = deq_2(qin, qout).$

By deq_2, we have
$b = head.rev\ qin$
$qin' = [\,]$
$qout' = tail(rev\ qin).$

Proving the correspondence:

$$
\begin{aligned}
q' &= rev(tail(rev(q))) && \text{by } deq_1 \\
&= rev.tail.rev\ (qin\ @\ (rev\ qout)) && \text{by assumption} \\
&= rev.tail.rev\ qin && \text{by } qout\ = [\,] \\
&= rev\ qout' && \text{by } deq_2 \\
&= qin'\ @\ (rev\ qout')) && \text{by } deq_2.
\end{aligned}
$$

To prove $a = b$, we have

$$
\begin{aligned}
a &= head.rev\ q && \text{by } deq_1 \\
&= head.rev\ (qin\ @\ (rev\ qout)) && \text{by assumption}
\end{aligned}
$$

$$= head\ (qout\ @\ (rev\ qin)) \qquad \text{by } rev$$
$$= head\ (rev\ qin) \qquad \text{by } qout\ = [\,]$$
$$= b \qquad \text{by } deq_2.$$

∎

5.3.4 Method for proving QueueOne equivalent to QueueThree

The justification of `QueueOne` equivalent to `QueueThree` is more difficult, since it involves comparing a pure module (`QueueOne`) with an imperative module (`QueueThree`). If we inspect the code of `QueueThree`, basically it is the same as the one in `QueueTwo`. One difference is that `QueueThree` uses a 'pointer' to a pair of list instead of a pair of list. The returning of a pair of lists in `QueueTwo` is replaced by the returning of a new pointer to a pair of lists.

Another difference is that `QueueThree` changes the state: the allocation of new pointers means that it always expands locations in the current state whenever it is called. More than that, in some cases `dequeue` can change the current store (see the presence of the `:=` operation in `dequeue`). There are two issues to consider: what can be externally observed with the references that are outputted from `QueueThree`? In what circumstances can we view `QueueThree` as being applicative? These are the key issues in characterising in what sense `QueueOne` is equivalent to `QueueThree`.

The answer can be informally explained as follows:

1. ML structures provides an encapsulation such that pointer representations of abstract data are invisible. This is done by declaring such data structure as opaque [Ull94, Gil97]. The references created by `QueueThree` are abstract data. By default they can be passed around. But since they are abstract data, they cannot be scrutinised directly by the users. For example, the user cannot compare the pointers, lookup the values pointed by the pointers, or modify the values pointed by the pointers.

2. We analyse how `QueueThree` expands and changes the current store. The expansion does not create any interference with the existing locations, so modulo the `:=` operation in `dequeue`, `QueueThree` is constant with respect to a computation relation that expands locations. To deal with the possibility of state changes by `dequeue`, notice that the assignment operation `:=` in `dequeue` has a particular pattern: if $s_0 = s[q \mapsto (qin, [\,])]$ is the state before the assignment (5.1), then $s_1 = s[q \mapsto ([\,], rev\ qin)]$ is the state after the assignment (5.1).

```
q := ([],rev qin)                                   (5.1)
```

The states s_0 and s_1 are related by the following.

$$valOf(s_0(q)) = qin::(rev\ [\,]) = [\,]::rev(rev\ qin) = valOf(s_1(q)).$$

This tells us that although the representations can change, the abstract meanings do not change.

Recall that in the last section we prove `QueueOne` equivalent to `QueueTwo` by defining a relation T between data type representations (at the semantics level) and showing that T relates the two implementations via the two commuting diagrams in page 81. This method is quite general and can be used for two implementations of abstract data types written in a pure functional language. One question we would like to ask is whether we can use the same method for proving `QueueOne` equivalent to `QueueThree`.

The answer to the above question is yes, but we have to do more work. In addition to the diagrams, we have to show that the behaviour of `QueueThree` is constant throught any possible state changes created by its implementation or the environment.

Here is the extra work that needs to be done.

1. First of all, it is instructive to define a computation invariant (call it R). This describes the possible changes of representation for queues that do not change the observable behaviour of the `QueueThree` functions. This is needed because we assume the environment does not make unrestricted changes to the state.

   The explanation on page 84 shows that `QueueThree` creates side effects by expanding locations or altering the internal representations of existing locations. However, `QueueThree` still behaves in a constant way in this context. Therefore, the computation invariant R that we need is the one that is formally defined in Figure 5.18.

$$
\begin{array}{l}
Qref = L \\
AllStates = L \rightarrow (int\ list\ \times\ int\ list) \cup \{Unused\}. \\
dom\ s\ = \{l \in L \mid\ l\ defined\text{-}in\ s\} \qquad\qquad ,for\ s \in AllStates. \\
S = \{s \in AllStates \mid\ dom\ s\ is\ finite\} \\
R \in S\ \leftrightarrow S \\
R(s, s') \\
\text{iff} \\
forall\ q \in dom\ s\ :\ valOf(s(q))\ = valOf(s'(q))
\end{array}
$$

Figure 5.18: The definition of R

$$
\begin{tikzcd}
\mathit{int\ list} \arrow[d, leftrightarrow, "T_0"'] & S \times \mathit{int\ list} \arrow[d, leftrightarrow, "T_1"] & S \times (N \times \mathit{int\ list}) \arrow[d, leftrightarrow, "T_2"] \\
\mathit{int\ list} \times \mathit{int\ list} & S \times \mathit{Qref} & S \times (N \times \mathit{Qref}) \\
\texttt{Q1} \leftrightarrow \texttt{Q2} & \texttt{Q1} \leftrightarrow \texttt{Q3} & \texttt{int*Q1} \leftrightarrow \texttt{int*Q3}
\end{tikzcd}
$$

Figure 5.19: `Q1`, `Q2`, and `Q3` are abbreviations for `QueueOne.Q`, `QueueTwo.Q`, and `QueueThree.Q`, respectively. T_0 is the representation relation between `QueueOne.Q` and `QueueTwo.Q` and T_1 is the representation relation between `QueueOne.Q` and `QueueThree.Q`. *Qref* is a space of pointers to pairs of lists. T_2 is the representation relation between (`int*QueueOne.Q`) and (`int*QueueThree.Q`). T_1 is used for relating the inputs of `QueueOne.dequeue` and `QueueThree.dequeue` whereas T_2 is used for relating the outputs of `QueueOne.dequeue` and `QueueThree.dequeue`.

2. The relation T between the data type representations is more involved. The relation T_0 in Figure 5.19 shows that we do not need the current state to observe representations of `QueueOne.Q` and `QueueTwo.Q`. In fact this is true for any pure ML code. The situation is different when we try to relate two representations where one (or both) of the representations are references. We have to interpret the values relative to the current states to get a fuller account of their behaviour. This is needed because what is significant in a `QueueThree.Q` representation (which is represented as a reference) is not the actual reference itself, but the value pointed to by the reference in the current store. The representation of `QueueOne.Q` is *int list* and the representation of `QueueThree` is *Qref* (a space of references). The relation T_1 between them is a binary relation of type $S \times \mathit{int\ list} \leftrightarrow S \times \mathit{Qref}$ (see Figure 5.19). We are also interested in relating observable values of type (`int*QueueOne.Q`) with ones of type (`int*QueueThree.Q`). This is used for relating the output of `QueueOne.dequeue` and `QueueThree.dequeue`. The relation T_2 between them is a binary relation of type $S \times (N \times \mathit{int\ list}) \leftrightarrow S \times (N \times \mathit{Qref})$. Sometimes we will omit the brackets for clarity purpose. The formal definitions of T_1 and T_2 are as follow:

$$T_1((s_i, l), (s_j, q)) \iff l = \mathit{valOf}(s_j(q)) \tag{5.2}$$

$$T_2((s_i, n, l), (s_j, m, q)) \iff l = \mathit{valOf}(s_j(q)) \wedge n = m \tag{5.3}$$

where $m, n \in N$. Note that T_1 can be simplified into a relation of type $\mathit{int\ list} \leftrightarrow S \times \mathit{Qref}$, but for uniformity purpose we adopt the earlier type.

3. In the monadic semantics, `QueueOne.dequeue` is interpreted as

$$
\begin{aligned}
deq_1 &\in \mathit{int\ list} \to T(N \times \mathit{int\ list}) \\
&\in \mathit{int\ list} \to (S \to S \times (N \times \mathit{int\ list}))
\end{aligned}
$$

We are interested in using the uncurried version of deq_1 :

$\mathit{uncurry}(deq_1) \in S \times \mathit{int\ list} \to S \times (N \times \mathit{int\ list})$.

When it is clear from context, we will omit the *uncurry* symbol. Using similar steps, we can interpret `QueueThree.dequeue` as

$\mathit{uncurry}(deq_3) \in S \times \mathit{Qref} \to S \times (N \times \mathit{Qref})$.

Again, we will omit the *uncurry* symbol when it is clear from context. Notice that the domains of $\mathit{uncurry}(deq_1)$ and $\mathit{uncurry}(deq_3)$ matches the domain-codomain of T_1 and the codomains of $\mathit{uncurry}(deq_1)$ and $\mathit{uncurry}(deq_3)$ matches the domain-codomain of T_2. At this stage we can draw the right diagrams for relating deq_1 with deq_3, enq_1 with enq_3, and emp_1 with emp_3 (where enq_1 is the interpretation of `QueueOne.enqueue`, emp_1 is the interpretation of `QueueOne.empty`, enq_3 is the interpretation of `QueueThree.enqueue`, and emp_3 is the interpretation of `QueueThree.empty`). Figure 5.20 shows the diagrams.

$$
\begin{tikzcd}
S \times \mathit{int\ list} \arrow[r, "deq_1"] \arrow[d, leftrightarrow, "T_1"] & S \times (N \times \mathit{int\ list}) \arrow[d, leftrightarrow, "T_2"] \\
S \times \mathit{Qref} \arrow[r, "deq_3"] & S \times (N \times \mathit{Qref})
\end{tikzcd}
$$

$$
\begin{tikzcd}
S \times (N \times \mathit{int\ list}) \arrow[r, "enq_1"] \arrow[d, leftrightarrow, "T_2"] & S \times \mathit{int\ list} \arrow[d, leftrightarrow, "T_1"] \\
S \times (N \times \mathit{Qref}) \arrow[r, "enq_3"] & S \times \mathit{Qref}
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathbf{1} \arrow[r, "emp_1"] \arrow[d, leftrightarrow, "id"] & S \times \mathit{int\ list} \arrow[d, leftrightarrow, "T_1"] \\
\mathbf{1} \arrow[r, "emp_3"] & S \times \mathit{Qref}
\end{tikzcd}
$$

Figure 5.20: Diagrams relating the meanings of `QueueOne` and `QueueThree`

4. One role of R is in restricting the allowable state transformations that deq_1,deq_3,enq_1, and enq_3 could perform. Consider the first diagram in Figure 5.20. In addition to the diagram, we need to assert that the output state of $deq_3(s, q)$ is related to the input state s via R.

5. The last issue is the role of R in quantifying the diagrams in Figure 5.20. We are only interested in quantifying the diagram over states reachable from the point the `structure QueueOne`(or `QueueThree`) is declared.

The above method essentially uses the notion of *C-indistinguishable within* R_1,R_2 defined in Figure 4.6 on page 66. In this setting we model an ML structure as a computation and define indistinguishability between tuples of values as:

$$\begin{aligned}&(a_1, ..., a_n), (b_1, ..., b_n)\ \ \textit{V-indistinguishable}_{\sigma_1 * ... * \sigma_n}\ R, R' \iff \\ &\forall i \in 1..n : \\ &\quad a_i, b_i\, \textit{V-indistinguishable}_{\sigma_i}\ R, R'.\end{aligned} \tag{5.4}$$

To sum up, the method for showing `QueueOne` equivalent to `QueueThree` is:

- Define the interpretation of `QueueOne` in the monadic style.
- Define the interpretation of `QueueThree` in the monadic style.
- Define a relation $T_1 \in S \times \textit{int list} \leftrightarrow S \times \textit{Qref}$ between the data type representations `QueueOne.Q` and `QueueThree.Q`. Define a relation $T_2 \in S \times (N \times \textit{int list}) \leftrightarrow S \times (N \times \textit{Qref})$ between the data type representations (`int*QueueOne.Q`) and (`int*QueueThree.Q`).
- Prove `QueueOne` equivalent to `QueueThree` by proving the diagrams in Figure 5.20, where the diagrams are quantified over relevant states within R (see point 5). In addition to this, also prove that the side effects produced by `QueueOne` and `QueueThree` also satisfy R (see point 4).

The next section contains the details of the method.

5.3.5 Discussions

The idea of using a pair of lists as a representation of a queue is explained in [Gri81] and [Bur82]. The idea of `QueueThree` is suggested by [Fou95].

The allocation of new references whenever the `QueueThree` is used is crucial. At first sight, it looks unnecessary and one might think of using the same reference and let the module mechanism protect the access to the pointer. Code which uses the same reference is shown in Figure 5.21. It is basically the same as `QueueThree` but the allocation of new references is done only at empty operation. It is not equivalent to `QueueTwo` nor to `QueueThree` as the ML session in Figure 5.22 shows.

The exception is raised because `r1` interferes with `r2`. Internally, `r1` and `r2` are the same pointers. When we execute

```
structure QueueFour :  QSIG =
struct
  exception Q
  type Q = (int list * int list) ref
  val empty = ref([],[]) :  Q
  fun enqueue (a, q as ref(qin, qout)) =
      (q := (a ::  qin, qout); q)
  fun dequeue (q as ref(qin, (h ::  qout))) =
         (q := (qin, qout); (h,q))
    | dequeue (ref([],[])) = raise Q
    | dequeue (q as ref(qin,[])) = (q := ([], rev qin); dequeue q)
end;
```

Figure 5.21: A naive implementation of Queue

```
- open QueueFour;
open QueueFour
exception Q = Q
val empty = ref ([],[]) :  Q
val enqueue = fn :  int * Q -> Q
val dequeue = fn :  Q -> int * Q
- let
val q = enqueue(4,enqueue(3,empty))
val (h1,r1) = dequeue q
val (h2,r2) = dequeue r1
val r3 = enqueue(h1,r1)
val (_,r4) = dequeue r3
val (a,r5) = dequeue r4
in
a
end;
= = = = = = = = =
uncaught exception Q
-
```

Figure 5.22: An ML session for QueueFour

```
val (h2,r2) = dequeue r1
```

`r2` and `r1` both point to a pair of empty lists. In the case of `QueueTwo` and `QueueThree`, the resulting expressions in Figure 5.22 yield `3`.

The issue of whether imperative functional languages are superior to the pure ones is a controversial one; each camp believes in their own philosophy, even still argues for other features of its programming language when there is a sign of defeat. Pippenger [Pip97] shows that there exists an efficient impure Lisp program which cannot be matched by a pure Lisp program for a real-time online computation.

The method of showing `QueueOne` and `QueueThree` are equivalent uses logical relations and is similar to other logical-relation based methods for proving equality between programs. Stark [Sta94, Sta96] develops a logic of equality for a language with names which is complete up to first order types. Meyer and Sieber [MS88] develops a model of Idealized Algol based on invariant preserving relations. The model is refined by Sieber [Sie96a] to be fully abstract up to second order types. O'Hearn and Tennent [OT95] develops a different model of Idealized Algol based on parametricity and representation independence — a concept primarily proposed by Reynolds [Rey83] for studing properties of abstract data types.

It is unknown how to scale up Stark's logic to languages that include assignments statements and modules. The above Idealized Algol models are good for stack based languages, but work in using the approaches for heap based languages is yet to be explored.

5.4 QueueOne equivalent to QueueThree

This section gives the setting for interpreting ML structures and characterising indistinguishability between two interpretations. Subsection 5.5.1 defines an interpretation $Queue_1$ of `QueueOne` and an interpretation $Queue_3$ of `QueueThree`. Subsection 5.5.2 formalises the definitions of *const-within* R for $Queue_1$ and $Queue_3$, and shows $Queue_1$ *const-within* R and $Queue_3$ *const-within* R, where R is the expansive relation defined on Figure 5.18. Subsection 5.5.3 formalises the definition of $Queue_1$,$Queue_3$ *indistinguishable* R,R. The definition is essentially based on the notion of *indistinguishable within* defined in Chapter 4.4, but here we have to handle two different representations.

5.4.1 Definitions

The way the semantics of `QueueThree` is obtained is essentially done using the definition in Chapter 4 on page 41. We have the following additional assumptions.

- The only storable values are pairs of lists (see Figure 5.18). This is to simplify our analysis.
- An ML structure is interpreted as a computation. Declaring a structure is essentially the same as declaring a tuple of values. There is a side effect when the declaration is executed.
- We have *reverse* (or *rev*), *cons* (or ::), *head*, and *tail* operations on finite lists of natural numbers. Also assume we can do pattern matching.

Figure 5.23 and 5.24 shows the interpretation of `QueueOne`. Figure 5.25, 5.26, 5.27, and 5.28 shows the interpretation of `QueueThree`.

$$
\boxed{\begin{array}{ll}
Queue_1 & \in S \to S\times [\![\texttt{Q * (int * Q} \to \texttt{Q) * (Q} \to \texttt{int * Q)}]\!] \\
 & \in S \to S\times ([\![\texttt{Q}]\!] \times [\![\texttt{int * Q} \to \texttt{Q}]\!] \times [\![\texttt{Q} \to \texttt{int * Q}]\!]) \\
 & \in S \to S\times (\mathit{int\ list} \times \\
 & \qquad\qquad N \times \mathit{int\ list} \to (S \to S \times \mathit{int\ list}) \times \\
 & \qquad\qquad \mathit{int\ list} \to (S \to S \times (N \times \mathit{int\ list})))
\end{array}}
$$

Figure 5.23: The type of $Queue_1 = \{\![\texttt{QueueOne}]\!\}$

$$
\boxed{\begin{array}{ll}
(Queue_1\ s) = & \underline{let} \\
 & \quad emp_1 = [\,] \\
 & \quad enq_1\ (n,l)\ s = (s, n{::}l) \\
 & \quad deq_1\ l\ s = \underline{let}\ (h{::}l') = rev\ l \\
 & \qquad\qquad\qquad \underline{in}(s, h, rev\ l') \\
 & \qquad\qquad\qquad \underline{end} \\
 & \underline{in}\ (s, (emp_1, enq_1, deq_1)) \\
 & \underline{end}
\end{array}}
$$

Figure 5.24: The monadic interpretation of `QueueOne`

Note that the type of *flush* in Figure 5.28 is $Qref \to (S \to S)$. Given a pointer q to a pair of lists and a current state s, what it does is that it rearranges the internal representation of the queue pointed by q and reflects the change to the store. This is the only section in the definition of $Queue_3$ that changes the store. The change is safe since it preserves the computational invariant R (see Figure 5.29).

$$
\begin{aligned}
Queue_3 &\in S \to S \times [\![\texttt{Q} * \texttt{(int} * \texttt{Q} \to \texttt{Q)} * \texttt{(Q} \to \texttt{int} * \texttt{Q)}]\!] \\
&\in S \to S \times ([\![\texttt{Qrep}]\!] \times [\![\texttt{int} * \texttt{Qrep} \to \texttt{Qrep}]\!] \times [\![\texttt{Qrep} \to \texttt{int} * \texttt{Qrep}]\!]) \\
&\in S \to S \times (Qref \times \\
&\qquad\qquad N \times Qref \to (S \to S \times Qref) \times \\
&\qquad\qquad Qref \to (S \to S \times (N \times Qref)))
\end{aligned}
$$

Figure 5.25: The type of $Queue_3 = \{\![\texttt{QueueThree}]\!\}$

$$
\begin{aligned}
(Queue_3\ s) = {}& \underline{let} \\
& \quad q \notin dom\ s \\
& \quad emp_3 = q \\
& \quad enq_3 = \text{...see Figure 5.27 ...} \\
& \quad deq_3 = \text{...see Figure 5.28 ...} \\
& \underline{in} \\
& \quad (s[q \mapsto ([\,], [\,])], (emp_3, enq_3, deq_3)) \\
& \underline{end}
\end{aligned}
$$

Figure 5.26: The monadic interpretation of `QueueThree`

$$
\begin{aligned}
enq_3\ (n, q)\ s = {}& \underline{let} \\
& \quad (qin, qout) = s(q) \\
& \quad q' \notin dom\ s \\
& \underline{in} \\
& \quad (s[q' \mapsto (n{::}qin, qout)], q') \\
& \underline{end}
\end{aligned}
$$

Figure 5.27: The definition of enq_3

```
deq3 q s = let
             (qin, qout) = s(q)
             f (qin, n::qout') s = let
                                     q' ∉ dom s
                                   in
                                     (s[q' ↦ (qin, qout'), n, q'])
                                   end
             flush q s = let
                           (qin, qout) = s(q)
                         in
                           (s[q ↦ ([ ], qout@(reverse qin))])
                         end
           in
           if (qout ≠ [ ]) then f (qin, qout) s
           else if (qout = [ ] ∧ qin = [ ]) then error
           else let s' = flush q s
                  in f (s'(q)) s'
                end
           end
```

Figure 5.28: The definition of deq_3

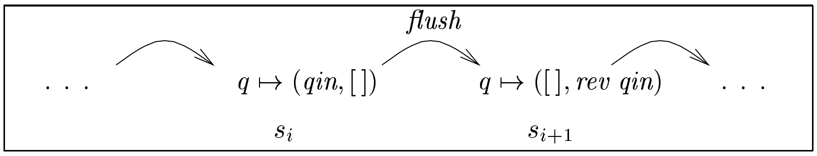

Figure 5.29: The function *flush* does modify the state, but it still repects the invariant R. The symbol q denotes a location in states s_i and s_{i+1}
.

Convention: for clarity purpose, sometimes we would write $(s, (a, b, c))$ as (s, a, b, c). For the rest of the chapter, R refers to the transition system defined in Figure 5.18 on page 85.

5.4.2 Proof of being constant

One of the most useful properties we want about $Queue_1$ and $Queue_3$ is that they are observationally equivalent. However, since we are working at the denotational level and adequacy results are beyond the scope of this thesis, we would study their equivalence at the denotational level. The notion *indistinguishability-within* is a good criteria for achieving observational equivalence. Another criteria is to show that externally, $Queue_3$ is an applicative module. At the denotational level, we express it as: $Queue_3$ *const-within* R. The definition and the proof of the property is the topic of this subsection.

5.4.2.1 $Queue_3$ const-within R

Definition 5.4.1 below defines what it means for $Queue_3$ module to be *const-within*.

Definition 5.4.1.

$Queue_3$ const-within R

iff

forall $s_i, s_j \in dom\ R$:

<u>let</u>

$(s'_i, emp_i, enq_i, deq_i) = Queue_3\ s_i$

$(s'_j, emp_j, enq_j, deq_j) = Queue_3\ s_j$

$R'_i\ = (reachable^R_{s'_i} \lhd R)$

$R'_j\ = (reachable^R_{s'_j} \lhd R)$

<u>in assert</u>

1. $R(s_i, s'_i) \wedge R(s_j, s'_j)$
2.
   a. *emp_i, emp_j V-indistinguishable R'_i, R'_j*

   b. *enq_i, enq_j V-indistinguishable R'_i, R'_j*

   c. *deq_i, deq_j V-indistinguishable R'_i, R'_j*

<u>end</u>

The definition is essentially a rewriting of the definition of *C-const-within* R defined in Figure 4.7 on page 66. Values indistinguishability is handled by (5.4). It is instructive to expand (2.a), (2.b), and (2.c), and study them at the representation level. This is needed because we would like to study how pointer representations behave.

Definition 5.4.2 is a general case of (2.a). It is a rewriting of the definition *V-indistinguishable*$_{\texttt{ref}}$ R_i, R_j (see Figure 4.6 on page 66) using T_3 for relating the pointer representations.

$$\begin{aligned} &T_3((s_i, q_i), (s_j, q_j)) \iff \\ &valOf(s_i(q_i)) = valOf(s_j(q_j)). \end{aligned} \tag{5.5}$$

Definition 5.4.2.

q_i, q_j V-indistinguishable$_{\texttt{Q}}$ R_i, R_j

iff

1. *q_i defined-in R_i*
2. *q_j defined-in R_j*
3. *forall $s_i \in dom\ R_i, s_j \in dom\ R_j : T_3((s_i, q_i), (s_j, q_j))$.*

Definition 5.4.3 and Definition 5.4.4 are rewritten in a similar way.

Definition 5.4.3.

enq_i, enq_j *V-indistinguishable*$_{\texttt{int*Q}\rightarrow\texttt{Q}}$ R_i, R_j
iff
$\forall q_1, q_2 \in Qref, a \in N$:
q_1, q_2 *V-indistinguishable*$_{\texttt{Q}}$ R_i, R_j
implies
$\forall s_0 \in dom\ R_i, s_1 \in dom\ R_j$:
<u>*let*</u>
$(s_i, q_1') = enq_i(a, q_1)s_0$
$(s_j, q_2') = enq_j(a, q_2)s_1$
<u>*in assert*</u>
1. $R_i(s_0, s_i) \wedge R_j(s_1, s_j)$
2. $\forall s_i' \in reachable_{s_i}, s_j' \in reachable_{s_j}$:
$valOf(s_i'(q_1')) = valOf(s_j'(q_2'))$

<u>*end*</u>

Definition 5.4.4.

deq_i, deq_j *V-indistinguishable*$_{\texttt{Q}\rightarrow\texttt{int*Q}}$ R_i, R_j
iff
$\forall q_1, q_2 \in Qref$:
q_1, q_2 *V-indistinguishable*$_{\texttt{Q}}$ R_i, R_j
implies
$\forall s_0 \in dom\ R_i, s_1 \in dom\ R_j$:
<u>*let*</u>
$(s_i, a, q_1') = deq_i\ q_1\ s_0$
$(s_j, b, q_2') = deq_j\ q_2\ s_1$
<u>*in assert*</u>
1. $R_i(s_0, s_i) \wedge R_j(s_1, s_j)$
2. $a = b$
3. $\forall s_i' \in reachable_{s_i}, s_j' \in reachable_{s_j}$:
$valOf(s_i'(q_1')) = valOf(s_j'(q_2'))$

<u>*end*</u>

We want to prove Definition 5.4.1 for R defined in Figure 5.18 on page 85. Our method is by proving conditions (1), (2.a), (2.b), and (2.c) of Definition 5.4.1 separately. Below are the steps.

Proposition 5.4.5.

$\forall r_i, r_j \in dom\ R:$
<u>*let*</u>

$(r'_i, emp_i, enq_i, deq_i) = Queue_3\ r_i$
$(r'_j, emp_j, enq_j, deq_j) = Queue_3\ r_j$

<u>*in assert*</u>

$R_i(r_i, r'_i) \wedge R_j(r_j, r'_j)$

<u>*end*</u>

Proof: Immediate from the definition of $Queue_3$. ∎

.

Proposition 5.4.6.
$\forall r_i, r_j \in dom\ R:$
<u>*let*</u>

$(r'_i, emp_i, enq_i, deq_i) = Queue_3\ r_i$
$(r'_j, emp_j, enq_j, deq_j) = Queue_3\ r_j$
$R_i\ = (reachable^R_{r'_i} \lhd R)$
$R_j\ = (reachable^R_{r'_j} \lhd R)$

<u>*in assert*</u>

$emp_i, emp_j\ \ V\text{-}indistinguishable_{\mathtt{Q}}\ R_i, R_j$

<u>*end*</u>

Proof: see Appendix. ∎

.

Proposition 5.4.7.
$\forall r_i, r_j \in dom\ R:$
<u>*let*</u>

$(r'_i, emp_i, enq_i, deq_i) = Queue_3\ r_i$
$(r'_j, emp_j, enq_j, deq_j) = Queue_3\ r_j$
$R_i\ = (reachable^R_{r'_i} \lhd R)$
$R_j\ = (reachable^R_{r'_j} \lhd R)$

<u>*in assert*</u>

$enq_i, enq_j\ \ V\text{-}indistinguishable_{\mathtt{int*Q}\rightarrow\mathtt{Q}}\ R_i, R_j$

<u>*end*</u>

Proof: see Appendix. ∎

.

Proposition 5.4.8.
$\forall r_i, r_j \in$ *dom R :*
<u>*let*</u>
$(r'_i, emp_i, enq_i, deq_i) = Queue_3\ r_i$
$(r'_j, emp_j, enq_j, deq_j) = Queue_3\ r_j$
$R_i\ = (reachable^R_{r'_i} \lhd R)$
$R_j\ = (reachable^R_{r'_j} \lhd R)$
<u>*in assert*</u>
deq_i, deq_j *V-indistinguishable*$_{\texttt{Q}\rightarrow\texttt{int*Q}}$ R_i, R_j
<u>*end*</u>

Proof: see Appendix. ∎

.

Theorem 5.4.9. *$Queue_3$ const-within R.*

Proof: It follows immediately from Proposition 5.4.5, 5.4.6, 5.4.7, and 5.4.8. ∎

To complete the discussion, we can also show that $Queue_1$ *const-within R*. The details of the definition and the proof are in Appendix B.1.

5.4.3 Proof of equivalence

Definition 5.4.10 below defines what it means for $Queue_1$,$Queue_3$ *indistinguishable R,R*.

Definition 5.4.10.
$Queue_1$,$Queue_3$ *indistinguishable R,R*
iff
forall $s_i, s_j \in$ *dom R :*
<u>*let*</u>
$(s'_i, emp_1, enq_1, deq_1) = Queue_1\ s_i$
$(s'_j, emp_3, enq_3, deq_3) = Queue_3\ s_j$
$R'_i\ = (reachable^R_{s'_i} \lhd R)$
$R'_j\ = (reachable^R_{s'_j} \lhd R)$
<u>*in assert*</u>
1. $R(s_i, s'_i) \wedge R(s_j, s'_j)$
2.
a. emp_1, emp_3 *V-indistinguishable* R'_i, R'_j

b. enq_1, enq_3 *V-indistinguishable* R'_i, R'_j

c. deq_1, deq_3 *V-indistinguishable* R'_i, R'_j

end

The definition is a rewriting of the definition of *C-indistinguishable within* R_1,R_2 defined in Figure 4.6 on page 66. Value tuples indistinguishability is handled by (5.4) on page 88.

The definition is similar to the notion $Queue_3$ *const-within* R (see Definition 5.4.1) since *const-within* is a special case of indistinguishability within. The difference is that the latter compare $Queue_3$ with itself, whereas the former compare $Queue_3$ with $Queue_1$.

We need to expand the conditions (2.a), (2.b), and (2.c), and study them at the representation level. This is required because we would like to study how a list representation is related to a pointer representation.

Definition 5.4.11 is a general case of (2.a). It is essentially a rewriting of the definition of *V-indistinguishable*$_{\texttt{ref}}$ R_i, R_j (see Figure 4.6 on page 66) using T_1 for relating the representations.

Definition 5.4.11.

q_1, q_3 *V-indistinguishable*$_{\texttt{Q}}$ R_i, R_j

iff

1. q_3 *defined-in* R_j

2. forall $s_i \in$ *dom* $R_i, s_j \in$ *dom* $R_j : T_1((s_i, q_1), (s_j, q_3))$.

Definition 5.4.12 and Definition 5.4.13 are rewritten in a similar way. Implicit in the definitions is the use of T_2 for relating the inputs of enq_1 and enq_3 and the outputs of deq_1 and deq_3.

Definition 5.4.12.

enq_1, enq_3 *V-indistinguishable*$_{\texttt{int*Q}\to\texttt{Q}}$ R_i, R_j

iff

$\forall q_1, q_2 \in Qref, a \in N$:

q_1, q_2 *V-indistinguishable*$_{\texttt{Q}}$ R_i, R_j

implies

$\forall s_0 \in$ *dom* $R_i, s_1 \in$ *dom* R_j :

let

$(s_i, q'_1) = enq_1(a, q_1)s_0$

$(s_j, q'_2) = enq_3(a, q_2)s_1$

in assert

1. $R_i(s_0, s_i) \wedge R_j(s_1, s_j)$

2. $\forall s'_i \in reachable_{s_i}, s'_j \in reachable_{s_j}$ *:*

$valOf(s'_i(q'_1)) = valOf(s'_j(q'_2))$

<u>*end*</u>

Definition 5.4.13.

deq_1, deq_3 *V-indistinguishable*$_{\texttt{Q}\to\texttt{int*Q}}$ R_i, R_j

iff

$\forall q_1, q_2 \in$ *Qref :*

q_1, q_2 *V-indistinguishable*$_{\texttt{Q}}$ R_i, R_j

implies

$\forall s_0 \in$ *dom* $R_i, s_1 \in$ *dom* R_j *:*

<u>*let*</u>

$(s_i, a, q'_1) = deq_1\ q_1\ s_0$

$(s_j, b, q'_2) = deq_3\ q_2\ s_1$

<u>*in assert*</u>

1. $R_i(s_0, s_i) \wedge R_j(s_1, s_j)$

2. $a = b$

3. $\forall s'_i \in reachable_{s_i}, s'_j \in reachable_{s_j}$ *:*

$valOf(s'_i(q'_1)) = valOf(s'_j(q'_2))$

<u>*end*</u>

We want to prove Definition 5.4.10 for R defined in Figure 5.18 on page 85. Our method is by proving conditions (1), (2.a), (2.b), and (2.c) of Definition 5.4.10 separately. Below are the steps.

Proposition 5.4.14.

$\forall r_i, r_j \in$ *dom* R *:*

<u>*let*</u>

$(r'_i, emp_1, enq_1, deq_1) = Queue_1\ r_i$

$(r'_j, emp_3, enq_3, deq_3) = Queue_3\ r_j$

<u>*in assert*</u>

$R_i(r_i, r'_i) \wedge R_j(r_j, r'_j)$

<u>*end*</u>

Proof: Immediate from the definition of $Queue_1$ and $Queue_3$. ∎

.

Proposition 5.4.15.

$\forall r_i, r_j \in$ *dom* R *:*

<u>*let*</u>

$(r_i', emp_1, enq_1, deq_1) = Queue_1 \ r_i$
$(r_j', emp_3, enq_3, deq_3) = Queue_3 \ r_j$
$R_i \ = (reachable^R_{r_i'} \lhd R)$
$R_j \ = (reachable^R_{r_j'} \lhd R)$
<u>*in assert*</u>
$emp_1, emp_3 \ \textit{V-indistinguishable}_{\mathtt{Q}} \ R_i, R_j$
<u>*end*</u>

Proof: see Appendix. ∎

.

Proposition 5.4.16.
$\forall r_i, r_j \in dom \ R$:
<u>*let*</u>
$(r_i', emp_1, enq_1, deq_1) = Queue_1 \ r_i$
$(r_j', emp_3, enq_3, deq_3) = Queue_3 \ r_j$
$R_i \ = (reachable^R_{r_i'} \lhd R)$
$R_j \ = (reachable^R_{r_j'} \lhd R)$
<u>*in assert*</u>
$enq_1, enq_3 \ \textit{V-indistinguishable}_{\mathtt{int*Q \to Q}} \ R_i, R_j$
<u>*end*</u>

Proof: see Appendix. ∎

.

Proposition 5.4.17.
$\forall r_i, r_j \in dom \ R$:
<u>*let*</u>
$(r_i', emp_1, enq_1, deq_1) = Queue_1 \ r_i$
$(r_j', emp_3, enq_3, deq_3) = Queue_3 \ r_j$
$R_i \ = (reachable^R_{r_i'} \lhd R)$
$R_j \ = (reachable^R_{r_j'} \lhd R)$
<u>*in assert*</u>
$deq_1, deq_3 \ \textit{V-indistinguishable}_{\mathtt{Q \to int*Q}} \ R_i, R_j$
<u>*end*</u>

Proof: see Appendix. ∎

.

Theorem 5.4.18. *$Queue_1$, $Queue_3$ indistinguishable R,R*

Proof: It follows immediately from Proposition 5.4.14, 5.4.15, 5.4.16, and 5.4.17. ∎

Chapter 6

Conclusions and directions for further research

In this thesis, we have developed a method of characterising constant terms in an ML-like language. First of all, we define the notion of indistinguishable throughout a reachable set and being-constant throughout a reachable set at the denotational level. The same method is also applicable at the operational level. We then extend the method by using transition systems, instead of reachable sets and define

$$\{\textit{indistinguishable throughout}_\sigma\ R\}$$
$$\{\textit{const-throughout}_\sigma\ R\}$$

where R is a transition system. The definition of *indistinguishable throughout* R is an equivalence relation over elements that are *const-throughout* R, and the property of *const-throughout* R is preserved under function application. The tools that we use are not new: they are store semantics with side effect monad, logical relation over values and computations, and transition system. The contribution of this thesis is the decision to use transition system as an abstract notion of a class of contexts.

The above structure are suitable for analysing terms with flat stores, but it does not handle dynamic allocation well. For example, it cannot equate the denotations of `(ref 8)` and `((ref 7); (ref 8))`. In order to equate such terms, we develop a notion called *indistinguishable within* R_1,R_2. By parameterising the indistinguishability relation over a pair of transition relations, we can handle the freedom of allocating fresh locations in *new*. This definition is also applicable to reasoning about imperative implementation of Queue module where a queue is implemented as a pointer.

The use of a pair of transition systems has its own consequences. It is not known whether the notion is transitive in the following sense:

if a, b *indistinguishable within*$_\sigma$ R_1, R_2 $\wedge$
b, c *indistinguishable within*$_\sigma$ R_2, R_3
then
a, c *indistinguishable within*$_\sigma$ R_1, R_3.

For ground types, the above is satisfied, but for function types it is unknown whether the above is satisfied. It seems that it is unlikely to be satisfied, but we have not devised a counterexample yet.

Trying to restrict transitivity over a pair of identical relations does not simplify the problem, because in comparing function type values (say of type $\sigma \rightarrow \tau$), we need *C-indistinguishable within*$_\tau$ R,R, which in turn needs *V-indistinguishable within*$_\tau$ R_1,R_2[1]. One future work is to find a suitable notion of transitivity for *indistinguishable within*. This might involve modifying the definition of *indistinguishable within*.

The thesis is driven by the practical motivation of implementing applicative programs using programs that internally use references. Here we avoid dealing directly with local references. We parameterise the notion of being constant over transition systems (or pairs of transition systems) and give an informal motivation that a transition system is used for capturing an invariant that is satisfied by local variables in a program. We have not given a formal relationships between the notion of local variable and the structure transition system. This would require additional structures for describing features of locality and privacy (e.g. a structure for describing local references that cannot escape from the programs they are residing). This may be done by extending our existing framework or replacing our framework with more sophisticated structures. With more sophisticated structures we hope to have an in-built equality between two denotations which is adequate with respect to the language's operational semantics.

In Chapter 5, we discuss Queue implementations at the denotational level and show $Queue_1$, $Queue_3$ *indistinguishable* R,R. Since we do not have adequacy result, there is yet no proof that they are observationally equivalent. However, within opaque data structure mechanism, we believe that they are observationally equivalent. One possible further work is to prove that it is so. The key of showing them equivalent depends on the following facts.

1. The formal results that they are indistinguishable within a pair of identical transition relations R.

[1] R_1 and R_2 are obtained from a particular $s \in \mathit{dom}\ R$ (see the definition in Figure 4.6 on page 66).

2. That point (1) above implies that $empty_1, enq_1, deq_1, empty_3, enq_3$, and deq_3 satisfy the invariant R.

3. The informal observation that the opaque mechanism of ML module encapsulates the reference implementation of $Queue_3$. Thus since $empty_3, enq_3$, and deq_3 satisfy the invariant R, any computation that uses $empty_3, enq_3$, or deq_3 also satisfy the invariant R.

Appendix A

Ingredients of monadic semantics

A standard way of giving a monadic denotational model of a programming language is:

> Given a programming language (call it *myPL*), find an appropriate monad structure T, a computational meta language *CML* induced by the structure, and establish interpretations between the programming language and the computational metalanguage and between the computational metalanguage and the monad structure.

This method can be succintly described in terms of the following diagram.

$$myPL \xrightarrow{(\!|\ |\!)} CML \xrightarrow{[\![\ ]\!]} (T, \eta, \_^*)$$

Chapter 4 of this thesis uses this approach in interpreting iML.

The computational metalanguage *CML* is more explicit than the programming language: the types are more explicit since it distinguishes values and computations and the term constructors are more explicit too, particularly in expressing the order of a computation.

Chapter3 of this thesis defines a programming language iML and Chapter4 defines a computational metalanguage $\mathrm{CML}_{\mathrm{iML}}$ for interpreting iML. The interpretation functions $[\![\ ]\!]$ and $(\!|\ |\!)$ are defined on page 50 and page 51, respectively. Note that the type expressions of $\mathrm{CML}_{\mathrm{iML}}$ have an extra construct $T\sigma$ which does not appear in the type expressions of iML.

Other examples of programming languages with their corresponding computational metalanguages are described in [Sta94] (for a language with names and side effects), and [Cen96] (for a language with exceptions).

Since a computational metalanguage consists of simply-typed lambda calculus added with extra constructs for expressing various notions of computation, it can be viewed as a programming language in its own right. The only thing that it

$$\begin{tikzcd}[column sep=huge]
myPL \arrow[r, "(\![\ ]\!)"] & CML \\
+ & + \\
\text{operational semantics} & \text{equational logic/ evaluation logic} \arrow[l, dashed, "\text{adequacy theorem}"']
\end{tikzcd}$$

Figure A.1: An adequacy result of the equational/evaluation logic allows us to reason about some aspects of the operational semantics of the source language.

is lacking is an operational semantics. However, this deficiency is compensated by the existence of program logics on the computational metalanguage. At a simple level, we have equational logics ([Mog89]), and at a more sophisticated level, we have evaluation logics which include modalities ([Pit91]). The logics serve (at least) two purposes: at the denotational level they give insights and understanding of monadic structures, and at the operational level they often characterise operational properties of the source programming language quite accurately (in the form of adequacy theorems) — thus giving a more syntactic and proof theoretic account of some aspects of the source programming language. This syntactic level of explanation is useful for programmers and nonmathematicians who want to reason about the programming language. Diagram A.1 illustrates the explanation.

The following are a list of examples of reasoning about programming languages in terms of computational metalanguages.

1. [Cen96] defines a pure fragment of ML augmented with exceptions (called *TMLE*). The semantics behaviour is described in terms of an operational semantics. A suitable computational metalanguage ML_T for dealing with exceptions is defined and an equational logic on the metalanguage is defined too. Then he defines a translation $(\![\ ]\!)$ from *TMLE* to ML_T and shows that any closed term M reduces to n iff $(\![M]\!) = (\![n]\!)$ is provable in the logic of the metalanguage (where n is an integer value). Hence the logic gives rise to an adequate model of the programming language.

2. Stark (in [Sta94, Sta96]) defines a lambda calculus with names (called nu-calculus) and its operational semantics. On the other hand, he defines a computational metalanguage *CML* (which includes *new* construct) and its equational logic. He defines a translation from nu-calculus to *CML* and shows that the equational logic is adequate with respect to the operational semantics.

3. [Pit91] defines a pure fragment of ML augmented with flat store and recursion (called TINY-ML). An operational semantics in the form of

   $$s,\mathsf{e} \twoheadrightarrow s',\mathsf{c}$$

   is given (where e is an expression, c a canonical value, and s,s' are stores). An evaluation logic with the appropriate constants and exioms is defined (the logic includes a computational metalanguage). Then he defines a translation of TINY-ML expressions (of the form $\mathsf{e}{:}\sigma$) into a metalanguage expression (of the form $[\![\mathsf{e}]\!]{:}T[\![\sigma]\!]$). The logic defined is adequate with respect to the operational semantics.

4. [Mog96] encodes the Variable Typed Logic of Effects (*VTLoE*) into an evaluation logic. *VTLoE* is a programming logic for reasoning about an untyped version of pure ML fragment augmented with references (called λ_{mk}). Moggi defines an evaluation logic EL_T with the appropriate signature (which includes locations and values types, and reference allocation, reference lookup, and reference update constructs). A translation between *VTLoE* and EL_T is defined and Proposition 4.6 of [Mog96] shows that the equational logic EL_T is adequate with respect to the equational logic of *VTLoE*.

Another tools which is useful for programmers is to have computational metalanguages simulated or coded in programming languages (they are usually higher order functional languages). The reason for doing this is that monad is a tool for writing modular programs. [Wad92b] gives an example of an interpreter written in Haskell, where he starts with a bare interpreter which reduces a term to a value, and provides possible extensions for incorporating error messages, error messages with positions, execution count, output, or nondeterminism. The extensions are coded as monads and the bare interpreter is parameterised over monads. So an extension of a bare interpreter involves only plugging in the appropriatee monad and sometimes modifying the interpreter code a bit. Although in some cases we need to modify the bare code, the changes are minimal.

For programmers who are programming using a lot of list comprehensions [BW88], the theory of monads is useful because monads in mathematics corresponds to generalised list comprehensions in functional languages [Wad92a]. Hence we can use the monad theorems for reasoning about comprehensions in general and list comprehensions in particular. Note that this technique can also be used on other structures when we have a correspondence between a particular mathematical structure and a programming construct.

The aim of using monads for structuring various features of programming language is still an ongoing research since it is not clear how we combine monads. A more fundamental approach is the work of Power and Robinson [PR97] where they start from premonoidal categories and use them to model notions of computation. For encoding program logics into evaluation logics, [Mog96] mentions of the thought of encoding Henessy-Milner Logic [HM85] or Reynolds Specification Logic [Rey82] in evaluation logics. Topics on monads in functional programming are still growing and the best way to find up-to-date information on them is via URL links on those topics. [Wal97] is such a link.

Appendix B

Proofs

B.1 $Queue_1$ const-within R

Definition B.1.1 below defines what it means for $Queue_1$ to be *const-within*.

Definition B.1.1.
$Queue_1$ const-within R
iff
forall $s_i, s_j \in dom\ R$:
<u>*let*</u>
$(s'_i, emp_i, enq_i, deq_i) = Queue_1\ s_i$
$(s'_j, emp_j, enq_j, deq_j) = Queue_1\ s_j$
$R'_i\ = (reachable^R_{s'_i} \lhd R)$
$R'_j\ = (reachable^R_{s'_j} \lhd R)$
<u>*in assert*</u>
1. $R(s_i, s'_i) \wedge R(s_j, s'_j)$
2.
a. emp_i, emp_j *V-indistinguishable* R'_i, R'_j
b. enq_i, enq_j *V-indistinguishable* R'_i, R'_j
c. deq_i, deq_j *V-indistinguishable* R'_i, R'_j
<u>*end*</u>

The definition is a rewriting of the definition of *C-const-within R* defined in Figure 4.7 on page 66. Value tuples indistinguishability is handled by (5.4) on page 88. Throughout this section we omit the word '*within*' in *V-indistinguishable within*. This is to save space. We do not expand the points (2.a), (2.b), and (2.c) since they can be easily expanded using the definition in Figure 4.6. When we expand them, we regard a list of integer as a ground value.

We have the following theorem.

Theorem B.1.2. *$Queue_1$ const-within R.*

Proof:
Let $s_i, s_j \in dom\ R$.
Let
$(s'_i, emp_i, enq_i, deq_i) = Queue_1\ s_i$
$(s'_j, emp_j, enq_j, deq_j) = Queue_1\ s_j$
$R'_i\ = (reachable^R_{s'_i} \lhd R)$
$R'_j\ = (reachable^R_{s'_j} \lhd R)$.
Condition (1) is satisfied, since $(Queue_1\ s)_1 = s$ for all s.
We have
$emp_i = [\,]\ = emp_j$
$enq_i = enq_j = \lambda(n, l).\lambda s.(s, n{::}l)$
$deq_i = deq_j = \lambda l.\lambda s.\ \underline{let}\ (h{::}l') = (reverse\ l)\ \underline{in}\ (s, h, h')\ \underline{end}$.
The above implies the conditions (2.a), (2.b), and (2.c). ∎

B.2 Proof of Proposition 5.4.6

Consider $r_i, r_j \in dom\ R$.
Let
$(r'_i, (emp_i, enq_i, deq_i)) = Queue_3\ r_i$
$(r'_j, (emp_j, enq_j, deq_j)) = Queue_3\ r_j$
$R_i\ = (reachable^R_{r'_i} \lhd R)$
$R_j\ = (reachable^R_{r'_j} \lhd R)$.
By the definition of $Queue_3$, we have
$r'_i(emp_i) = ([\,], [\,]) = r'_j(emp_j)$.
ie.

$$valOf(r'_i(emp_i)) = valOf(r'_j(emp_j)). \tag{B.1}$$

Consider $s_i \in reachable^R_{r'_i}$
$s_j \in reachable^R_{r'_j}$.
We have

$$\begin{aligned} valOf(s_i(emp_i)) &= valOf(r'_i(emp_i)) && \text{by } R \text{ expansive relation} \\ &= valOf(r'_j(emp_j)) && \text{by (B.1)} \\ &= valOf(s_j(emp_j)) && \text{by } R \text{ expansive relation.} \end{aligned}$$

∎

B.3 Proof of Proposition 5.4.7

Consider $r_i, r_j \in dom\ R$.

Let

$(r'_i, (emp_i, enq_i, deq_i)) = Queue_3\ r_i$

$(r'_j, (emp_j, enq_j, deq_j)) = Queue_3\ r_j$

$R_i\ = (reachable^R_{r'_i} \lhd R)$

$R_j\ = (reachable^R_{r'_j} \lhd R)$.

Consider $a\ \in N, q_1, q_2 \in Qref$ such that

q_1, q_2 *V-indistinguishable*$_{\mathtt{Q}}$ R_i, R_j.

Consider $s_0 \in dom\ R_i$

$s_1 \in dom\ R_j$.

Let $(s_i, q'_1) = enq_i\ (a, q_1)s_0$

$(s_j, q'_2) = enq_j\ (a, q_2)s_1$.

We want to show that q'_1,q'_2 are related $\wedge\ R_i(s_0, s_i)\ \wedge\ R_j(s_1, s_j)$. Note that the first conjunction means $valOf(s'_i(q'_1)) = valOf(s'_j(q'_2))$ for all s'_i reachable from s_i and s'_j reachable from s_j. By the definition of enq_i, enq_j, q'_1 and q'_2 will be new locations that points to $(a :: qin_1, qout_1)$ and $(a :: qin_2, qout_2)$, respectively (where $(qin_1, qout_1) = s_0(q_1)$ and $(qin_2, qout_2) = s_1(q_2)$). Let $s'_i \in reachable^R_{r'_i}$ and $s'_j \in reachable^R_{r'_j}$. Since the invariant R is a conservative expansion, we have

$valOf(s'_i(q'_1)) = valOf(s_i(q'_1))$ and

$valOf(s'_j(q'_2)) = valOf(s_j(q'_2))$.

By our assumption we have $qin_1 @ rev\ qout_1 = valOf\ (s_0(q_1)) = valOf\ (s_1(q_2)) = qin_2 @ rev\ qout_2$. To show that q'_1 and q'_2 are related, we do the following.

$valOf(s_i(q'_1))$

$= (a :: qin_1)\ @\ rev\ qout_1$	by *valOf*
$= a :: (qin_1\ @\ rev\ qout_1)$	rearranging the bracket
$= a :: (qin_2\ @\ rev\ qout_2)$	by assumption
$= (a :: qin_2)\ @\ rev\ qout_2)$	rearranging the bracket
$= valOf(s_j(q'_2))$.	by *valOf*

Hence, $valOf(s'_i(q'_1)) = valOf(s'_j(q'_2))$ for all s'_i reachable from s_i and s'_j reachable from s_j.

It is clear that the side effects of enq_i satisfies R_i since $(enq_i(a, q)s)_1$ is an extension of s for any $a \in N, q \in Qref$, and $s \in S$. Similarly, we have enq_j satisfies R_j. In other words, we have $R_i(s_0, s_i)\ \wedge\ R_j(s_1, s_j)$.

Therefore , enq_i, enq_j *V-indistinguishable*$_{\mathtt{int*Q \to Q}}$ R_i, R_j ∎

B.4 Proof of Proposition 5.4.8

Consider $r_i, r_j \in dom\ R$.

Let

$(r'_i, emp_i, enq_i, deq_i) = Queue_3\ r_i$

$(r'_j, emp_j, enq_j, deq_j) = Queue_3\ r_j$

$R_i\ = (reachable^R_{r'_i} \lhd R)$

$R_j\ = (reachable^R_{r'_j} \lhd R)$.

Consider $q_1, q_2 \in Qref$ such that

q_1, q_2 *V-indistinguishable*$_{\mathtt{Q}}$ R_i, R_j.

Consider $s_0 \in dom\ R_i$

$s_1 \in dom\ R_j$.

Let $(s_i, (a, q'_1)) = deq_i\ q_1\ s_0$

$(s_j, (b, q'_2)) = deq_j\ q_2\ s_1$.

We want to show that q'_1, q'_2 are related $\wedge\ a = b\ \wedge\ R_i(s_0, s_i)\ \wedge\ R_j(s_1, s_j)$. Note that the first conjunction means $valOf(s'_i(q'_1)) = valOf(s'_j(q'_2))$ for all s'_i reachable from s_i and s'_j reachable from s_j. Let $s'_i \in reachable^R_{r'_i}$ and $s'_j \in reachable^R_{r'_j}$. First of all, we need to define the internal representations of q_1 and q_2. Let

$(qin_1, qout_1) = s_0(q_1)$ and

$(qin_2, qout_2) = s_1(q_2)$.

We have four cases to consider

1. Case $qout_1 \neq [\,]\ \wedge\ qout_2 \neq [\,]$. Then by the definition of deq_3, q'_1 and q'_2 will be new locations that points to $(qin_1, tail\ qout_1)$ and $(qin_2, tail\ qout_2)$, respectively. Since the invariant R is a conservative expansion, we have

   $valOf(s'_i(q'_1)) = valOf(s_i(q'_1))$ and

   $valOf(s'_j(q'_2)) = valOf(s_j(q'_2))$.

   From our assumption we have $qin_1@(rev\ qout_1) = valOf(s_0(q_1)) = valOf(s_1(q_2))$ $= qin_2@(rev\ qout_2)$ To show that q'_1 and q'_2 are related, we do the following.

   $$\begin{aligned}
   &valOf(s_i(q'_1))\\
   &\quad = qin_1@rev(tail\ qout_1)\\
   &\quad = rev.rev\ (qin_1@rev(tail\ qout_1))\\
   &\quad = rev\ ((tail\ qout_1)@rev\ qin_1)\\
   &\quad = rev.tail\ (qout_1@rev\ qin_1)\\
   &\quad = rev.tail.rev\ (qin_1@rev\ qout_1)\\
   &\quad = rev.tail.rev\ (qin_2@rev\ qout_2)
   \end{aligned}$$

$$
\begin{aligned}
&= rev.tail\ (qout_2 @ rev\ qin_2) \\
&= rev((tail\ qout_2) @ rev\ qin_2) && \text{by } qout_2 \neq [\,] \\
&= rev.rev(qin_2 @ rev(tail\ qout_2)) \\
&= qin_2 @ rev(tail\ qout_2) \\
&= valOf(s_j(q_2')).
\end{aligned}
$$

Hence, $valOf(s_i'(q_1')) = valOf(s_j'(q_2'))$ for all s_i' reachable from s_i and s_j' reachable from s_j.

For proving $a = b$ we do the following.

$$
\begin{aligned}
a &= head\ qout_1 \\
&= head\ (rev(valOf(s_0(q_1)))) \\
&= head\ (rev(valOf(s_1(q_2)))) && \text{by our assumption} \\
&= head\ qout_2 \\
&= b.
\end{aligned}
$$

Finally, by the definitions of deq_i, deq_j the side effects part of deq_i, deq_j satisfy R_i, R_j, respectively. Hence we have $R_i(s_0, s_i) \wedge R_j(s_1, s_j)$.

2. Case $qout_1 = [\,] \wedge qout_2 \neq [\,]$. By the definition of deq_i, deq_j, q_1' and q_2' will be new locations that points to $([\,], tail\ (rev\ qin_1))$ and $(qin_2, tail\ qout_2)$, respectively. Since the invariant R is a conservative expansion, we have
   $valOf(s_i'(q_1')) = valOf(s_i(q_1'))$ and
   $valOf(s_j'(q_2')) = valOf(s_j(q_2'))$.
   From our assumption we have $qin_1 @\ [\,] = valOf(s_0(q_1)) = valOf(s_1(q_2)) = qin_2 @\ (rev\ qout_2)$. Hence $qin_1 = qin_2 @\ (rev\ qout_2)$. To show that q_1' and q_2' are related, we do the following.

$$
\begin{aligned}
&valOf(s_i(q_1')) \\
&\quad = [\,]\ @\ rev(tail(rev(qin_1))) \\
&\quad = rev.tail.rev\ (qin_2 @\ (rev\ qout_2)) \\
&\quad = rev.tail\ (qout_2 @ (rev\ qin_2)) \\
&\quad = rev\ ((tail\ qout_2) @ (rev\ qin_2)) \\
&\quad = qin_2 @ (rev(tail\ qout_2)) \\
&\quad = valOf(s_j(q_2')).
\end{aligned}
$$

Hence, $valOf(s_i'(q_1')) = valOf(s_j'(q_2'))$ for all s_i' reachable from s_i and s_j' reachable from s_j.

To prove $a = b$, we have

$$
\begin{aligned}
a &= head(rev\ qin_1) \\
&= head(rev(qin_2@\ (rev\ qout_2))) \\
&= head(qout_2@(rev\ qin_2)) \\
&= head\ qout_2 && \text{by } qout_2 \neq [\,] \\
&= b.
\end{aligned}
$$

The side effects part of deq_i (and deq_j) satisfies R_i (and R_j) since the internal representation of q_1 after the computation — which is $([\,], (rev\ qin_1))$ — is related to the one before the computation — which is $(qin_1, [\,])$ — via $valOf$. In other words, $valOf(qin_1, [\,]) = valOf([\,], (rev\ qin_1))$. Hence we have $\forall q \in dom\ s_0.valOf(s_0(q)) = valOf(s_i(q))$ and $\forall q \in dom\ s_1.valOf(s_1(q)) = valOf(s_j(q))$. Therefore, we have $R_i(s_0, s_i) \wedge R_j(s_1, s_j)$.

3. Case $qout_1 \neq [\,] \wedge qout_2 = [\,]$ is done in a similar way.

4. Case $qout_1 = [\,] \wedge qout_2 = [\,]$. By the definition of deq_i, deq_j, q_1' and q_2' will be new locations that point to $([\,], tail\ (rev\ qin_1))$ and $([\,], tail\ (rev\ qin_2))$, respectively. Since the invariant R is a conservative expansion, we have
   $valOf(s_i'(q_1')) = valOf(s_i(q_1'))$ and
   $valOf(s_j'(q_2')) = valOf(s_j(q_2'))$.
   From our assumption we have $qin_1@[\,] = valOf(s_0(q_1)) = valOf(s_1(q_2)) = qin_1@[\,]$. In other words, $qin_1 = qin_2$. To show that q_1' and q_2' is related, we do the following.

$$
\begin{aligned}
&valOf(s_i(q_1')) \\
&\quad = [\,]\ @\ (rev.tail.rev\ qin_1) \\
&\quad = rev.tail.rev\ qin_1 \\
&\quad = rev.tail.rev\ qin_2 \\
&\quad = [\,]\ @\ (rev.tail.rev\ qin_2) \\
&\quad = valOf(s_j(q_2'))
\end{aligned}
$$

Hence, $valOf(s_i'(q_1')) = valOf(s_j'(q_2'))$ for all s_i' reachable from s_i and s_j' reachable from s_j.

To prove $a = b$, we have $a = head\ (rev\ qin_1) = head(rev\ qin_2) = b$.

To show that the side effects part of deq_i (and deq_j) satisfies the invariant R_i (and R_j), we have $valOf(qin_1, [\,]) = valOf([\,], rev\ qin_1)$ and $valOf(qin_2, [\,]) = valOf([\,], rev\ qin_2)$. Hence we have $\forall q \in dom\ s_0.valOf(s_0(q)) = valOf(s_i(q))$ and $\forall q \in dom\ s_1.valOf(s_1(q)) = valOf(s_j(q))$. Therefore, we have $R_i(s_0, s_i)$ $\wedge$ $R_j(s_1, s_j)$.

∎

B.5 Proof of Proposition 5.4.15

Consider $r_i, r_j \in \mathit{dom}\ R$.

Let

$(r'_i, (\mathit{emp}_1, \mathit{enq}_1, \mathit{deq}_1)) = \mathit{Queue}_1\ r_i$

$(r'_j, (\mathit{emp}_3, \mathit{enq}_3, \mathit{deq}_3)) = \mathit{Queue}_3\ r_j$

$R_i\ = (\mathit{reachable}^R_{r'_i} \lhd R)$

$R_j\ = (\mathit{reachable}^R_{r'_j} \lhd R)$.

By the definition of Queue_1 and Queue_3, we have

$$\mathit{emp}_1 = [\,]\ = \mathit{valOf}(r'_j(\mathit{emp}_3)). \tag{B.2}$$

Consider $s_i \in \mathit{reachable}^R_{r'_i}$

$s_j \in \mathit{reachable}^R_{r'_j}$.

We have

$\mathit{emp}_1 = \mathit{valOf}(r'_j(\mathit{emp}_3))$ by (B.2)

$\quad = \mathit{valOf}(s_j(\mathit{emp}_3))$ by R expansive relation.

∎

B.6 Proof of Proposition 5.4.16

Consider $r_i, r_j \in \mathit{dom}\ R$.

Let

$(r'_i, (\mathit{emp}_1, \mathit{enq}_1, \mathit{deq}_1)) = \mathit{Queue}_1\ r_i$

$(r'_j, (\mathit{emp}_3, \mathit{enq}_3, \mathit{deq}_3)) = \mathit{Queue}_3\ r_j$

$R_i\ = (\mathit{reachable}^R_{r'_i} \lhd R)$

$R_j\ = (\mathit{reachable}^R_{r'_j} \lhd R)$.

Consider $a\ \in N, q_1 \in \mathit{int\ list}, q_3 \in \mathit{Qref}$ such that

$q_1, q_3\ \mathit{V\text{-}indistinguishable}_{\mathtt{Q}}\ R_i, R_j$.

Consider $s_0 \in \mathit{dom}\ R_i$

$s_1 \in \mathit{dom}\ R_j$.

Let $(s_i, q'_1) = \mathit{enq}_1\ (a, q_1)s_0$

$(s_j, q'_3) = \mathit{enq}_3\ (a, q_3)s_1$.

We want to show that q'_1,q'_3 are related $\wedge\ R_i(s_0, s_i) \wedge R_j(s_1, s_j)$. Note that the first conjunction means $q'_1 = \mathit{valOf}(s'_j(q'_3))$ for all s'_i reachable from s_i and s'_j reachable from s_j. Let $s_i' \in \mathit{reachable}^R_{r_i'}$ and $s_j' \in \mathit{reachable}^R_{r_j'}$. By the definition of enq_3, q_3' is a new location that points to $(a :: qin, qout)$ (where $(qin, qout) = s_1(q_3)$). Since the invariant R is a conservative expansion, we have

$$\mathit{valOf}(s_j'(q_3')) = \mathit{valOf}(s_j(q_3')).$$

From the definition of enq_1, we have $q_1' = a{::}\ q_1$. From our assumption we have $q_1 = qin@rev\ qout$. To show that q_1' and q_3' are related, we do the following.

$$\begin{aligned} q_1' &= a{::}\ q_1 \\ &= a{::}\ (qin\ @\ (rev\ qout)) && \text{by our assumption} \\ &= (a :: qin)\ @\ (rev\ qout) \\ &= \mathit{valOf}(s_j(q_3')). \end{aligned}$$

Hence, $q_1' = \mathit{valOf}(s_j'(q_3'))$ for all s_i' reachable from s_i and s_j' reachable from s_j.

The side effects of enq_3 satisfies R_j since $(enq_3(a, q)s)_1$ is an extension of s for any $a \in N, q \in \mathit{Qref}$, and $s \in S$. It is clear that we have enq_1 satisfies R_i. In other words, we have $R_i(s_0, s_i) \wedge R_j(s_1, s_j)$.

Therefore , enq_1, enq_3 *V-indistinguishable*$_{\mathtt{int*Q \to Q}}$ R_i, R_j ∎

B.7 Proof of Proposition 5.4.17

Consider $r_i, r_j \in dom\ R$.

Let

$$\begin{aligned} (r_i', (emp_1, enq_1, deq_1)) &= Queue_1\ r_i \\ (r_j', (emp_3, enq_3, deq_3)) &= Queue_3\ r_j \\ R_i &= (\mathit{reachable}^R_{r_i'} \lhd R) \\ R_j &= (\mathit{reachable}^R_{r_j'} \lhd R). \end{aligned}$$

Consider $q_1 \in int\ list, q_3 \in \mathit{Qref}$ such that

$$q_1, q_3\ \textit{V-indistinguishable}_{\mathtt{Q}}\ R_i, R_j.$$

In this proof we are only interested in the case where $q_1 \neq [\,]$.

Consider $s_0 \in dom\ R_i$

$s_1 \in dom\ R_j$.

Let $(s_i, (a, q_1')) = deq_1\ q_1\ s_0$

$(s_j, (b, q_3')) = deq_3\ q_3\ s_1$.

We want to show that q_1', q_3' are related $\wedge\ a = b\ \wedge\ R_i(s_0, s_i)\ \wedge\ R_j(s_1, s_j)$. Note that the first conjunction means $q_1' = \mathit{valOf}(s_j'(q_3'))$ for all s_i' reachable from s_i and s_j' reachable from s_j. Let $s_i' \in \mathit{reachable}^R_{r_i'}$ and $s_j' \in \mathit{reachable}^R_{r_j'}$. First of all, we need to define the internal representation of q_3. Let

$(qin, qout) = s_1(q_3)$.

We have three cases to consider

1. Case $qout \neq [\,]$. Then by the definition of deq_3, q_3' is a new location that points to $(qin, tail\, qout)$. Since the invariant R is a conservative expansion, we have
   $valOf(s_j'(q_3')) = valOf(s_j(q_3'))$.
   From our assumption we have $q_1 = qin@(rev\, qout)$. To show that q_1' and q_3' are related, we do the following.

   $$\begin{aligned} q_1' &= rev(tail(rev(q_1))) \\ &= rev.tail.rev\ (qin\ @\ (rev\ qout)) \\ &= rev.tail\ (qout\ @\ (rev\ qin)) \\ &= rev\ ((tail\ qout)\ @(rev\ qin)) \\ &= qin\ @\ (rev\ (tail\ qout)) \\ &= valOf(s_j(q_3')). \end{aligned}$$

   Hence, $q_1' = valOf(s_j'(q_3'))$ for all s_i' reachable from s_i and s_j' reachable from s_j.

   To prove $a = b$, we do the following.

   $$\begin{aligned} a &= head.rev\ q_1 \\ &= head.rev\ (qin\ @\ (rev\ qout)) && \text{by our assumption} \\ &= head\ (qout\ @\ (rev\ qin)) \\ &= head\ qout && \text{by } qout \neq [\,] \\ &= b \end{aligned}$$

   Finally, by the definitions of deq_1, deq_3 the side effects part of deq_1, deq_3 satisfy R_i, R_j, respectively. Hence we have $R_i(s_0, s_i) \wedge R_j(s_1, s_j)$.

2. Consider the case $qin = [\,] \wedge qout = [\,]$. This implies that $q_1 = [\,]$. Therefore we have a contradiction.

3. Consider the case $qin \neq [\,] \wedge qout = [\,]$. By the definition of deq_3, q_3' is a new location that points to $([\,], tail\ (rev\ qin))$. Since the invariant R is a conservative expansion, we have
   $valOf(s_j'(q_3')) = valOf(s_j(q_3'))$.
   From our assumption we have $q_1 = qin\ @\ (rev\ qout) = qin$. To show that q_1' and q_3' are related, we do the following.

$$
\begin{aligned}
q_1' &= rev(tail(rev(q_1))) \\
&= rev(tail(rev(qin))) \\
&= [\,] \ @ \ rev(tail(rev(qin))) \\
&= valOf(s_j(q_3')).
\end{aligned}
$$

Hence, $q_1' = valOf(s_j'(q_3'))$ for all s_i' reachable from s_i and s_j' reachable from s_j.

To prove $a = b$, we have

$$
\begin{aligned}
a &= head.rev\ q_1 \\
&= head.rev\ (qin\ @\ (rev\ qout)) \\
&= head(qout\ @\ (rev\ qin)) \\
&= head\ (rev\ qin) && \text{by } qout\ = [\,] \\
&= b && \text{by definition of } deq_3
\end{aligned}
$$

The side effects part of deq_3 satisfies R_j since we have $\forall q \in dom\, s_1.valOf(s_1(q)) = valOf(s_j(q))$. For deq_1, it is clear that we have $\forall q \in dom\ s_0.valOf(s_0(q)) = valOf(s_i(q))$. Therefore, we have $R_i(s_0, s_i) \wedge R_j(s_1, s_j)$.

∎

Bibliography

[Abr91] J. R. Abrial. The B method for large software. specification, design and coding (abstract). In Soren Prehn and Hans Toetenel, editors, *Proceedings of Formal Software Development Methods (VDM '91)*, volume 552 of *LNCS*, pages 398–405, Berlin, Germany, October 1991. Springer.

[AHU87] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1987.

[AM91] Andrew W. Appel and David B. MacQueen. Standard ML of new jersey. In J. Małuszyński and M. Wirsing, editors, *Proceedings of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming, PLILP91, Passau, Germany*, Lecture Notes in Computer Science, pages 1–13. Springer-Verlag, August 1991.

[AM97] Samson Abramsky and Guy McCusker. Call-by-value games. In *Proceedings of CSL '97*, Lecture Notes in Computer Science. Springer-Verlag, 1997. To appear.

[And64] Christian Andersen. *An Introduction to ALGOL 60.* Addison Wesley, Reading, Massachusetts, 1964.

[AT 93] AT and T Bell Laboratories. *The Standard ML of New Jersey Library Reference Manual (Version 0.2)*, August 1993.

[Bur82] F. Warren Burton. Efficient functional implementation of FIFO queues. *Information Processing Letters*, 14(5):205–206, July 1982.

[BW88] Richard Bird and Philip Wadler. *Introduction to Functional Programming.* Prentice-Hall International Series in Computer Science. Prentice-Hall International, 1988.

[CDCV81] Mario Coppo, Mariangiola Dezani-Ciancaglini, and B. Venneri. Functional characters of solvable terms. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 27:45–58, 1981.

[Cen96] Pietro Cenciarelli. *Computational applications of calculi based on monads*. PhD thesis, University of Edinburgh, September 1996.

[CSC72] F. J. CORBAT, J. H. SALTZER, and C. T. CLINGEN. Multics–the first seven years. In *http://www.lilli.com/f7y.html*, 1972.

[CW96] Mary Campione and Hathy Walrath. *The Java Tutorial: O.O Programming for the Internet*. Addison Wesley, 1996.

[EM85] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, New York, N.Y., 1985.

[EM90] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints*, volume 21 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, New York, N.Y., 1990.

[Fel87] Matthias Felleisen. *The Calculi of λ_v-CS conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Indiana University, 1987.

[Fil96] Andrzej Filinski. *Controlling Effects*. PhD thesis, Carnegie Mellon University, May 1996.

[Fou95] Michael Fourman. *queue.ml*, 1995.

[Gab89] Richard P. Gabriel. The common lisp object system. *AI Expert*, 4(3):54–65, March 1989.

[GH90] Juan Guzmán and Paul Hudak. Single-threaded polymorphic lambda calculus. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science, Philadelphia, Pennsylvania*, pages 333–343, June 1990.

[Gil97] Stephen Gilmore. Programming in Standard ML '97: A Tutorial Introduction. Technical Report ECS-LFCS-97-364, Laboratory for

Foundations of Computer Science, University of Edinburgh, September 1997.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50, 1987.

[Gir93] Jean-Yves Girard. On the unity of logic. *Annals of Pure and Applied Logic*, 59:201–217, 1993.

[GJS97] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, 1997.

[Gri81] David Gries. *The Science of Programming*. Springer, New York, 1981.

[GTW78] J. A. Goguen, J. W. Thatcher, and E. G. Wagner. An initial algebra approach to the specification of abstract data types. In R. T. Yeh, editor, *Current Trends in Programming Methodology, Volume IV: Data Structuring*, pages 80–149, Englewood Cliffs, 1978. Prentice-Hall.

[Hen86] P. Henderson. Functional Programming, formal specification, and rapid prototyping. *IEEE Transactions on Software Engineering*, 12(2):241–250, 1986.

[HM85] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Jrnl. A.C.M.*, 23(1):137–161, January 1985.

[HMST95] Furio Honsell, Ian A. Mason, Scott Smith, and Carolyn Talcott. A variable typed logic of effects. *Information and Computation*, 119(1):55–90, 15 May 1995.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[HR95] Howard Huang and Uday Reddy. Type reconstruction for sci. In David N. Turner, editor, *Proceedings of the 1995 Glasgow Workshop on Functional Programming, Ullapool, Scotland*, 1995.

[Hug89] R. J. M. Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, April 1989.

[KST94] Stefan Kahrs, Donald Sannella, and Andrzej Tarlecki. The Definition of Extended ML. Technical Report ECS-LFCS-94-300, Laboratory for Foundations of Computer Science, University of Edinburgh, August 1994.

[Ler93] Xavier Leroy. The caml light system, release 0.6 documentation and user's manual. In *ftp://ftp.inria.fr/lang/caml-light/cl6refman.dvi.Z*, September 1993.

[LG88] Jon M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*. ACM, ACM Press, January 1988.

[Luo94] Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Oxford University Press, 1994.

[Man76] E. G. Manes. *Algebraic Theories*. Springer, New York, 1976.

[McC97a] Guy McCusker. *Personal communication*, 1997.

[McC97b] Guy McCusker. *Personal communication*, 1997.

[Mey88] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, N.Y., 1988.

[Mit90] John C. Mitchell. Type systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 8, pages 365–458. North-Holland, New York, N.Y., 1990.

[Mit96] John C. Mitchell. *Foundations for Programming Languages*. MIT Press, Cambridge, 1 edition, 1996.

[ML86] P. Martin-Lof. Constructive mathematics and computer programming. In C. A. R. Hoare and J. C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 167–184. Prentice Hall, 1986.

[Mog89] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings, Fourth Annual Symposium on Logic in Computer Science*, pages 14–23, Asilomar Conference Center, Pacific Grove, California, 5–8 June 1989. IEEE Computer Society Press.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991. Proposes using the category-theoretic notion of *monad* as a means of parametrizing programming-language semantics by a notion of value-producing computation. This proposal has been widely taken up in research on adding state constructs to functional programming languages.

[Mog96] Eugenio Moggi. Representing *VTLoE* in Evaluation Logic. Technical report, DISI, Univ. of Genova, April 1996.

[MS88] Albert R. Meyer and Kurt Sieber. Towards fully abstract semantics for local variables. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 191–203, San Diego, California, January 13–15, 1988. ACM SIGACT-SIGPLAN, ACM Press. Preliminary Report.

[MT92] I. Mason and C. Talcott. References, local variables, and operational reasoning. In *Proc. of 7th Annual Symposium on Logic in Computer Science*, 1992.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[O'H90] P. W. O'Hearn. *The semantics of non-interference: a natural approach*. PhD thesis, Queen's University, Kingston, Canada, 1990.

[Oka95] C. Okasaki. Amortization, lazy evaluation, and persistence: Lists with catenation via lazy linking. In *36th Annual Symposium on Foundations of Computer Science (FOCS'95)*, pages 646–654, Los Alamitos, October 1995. IEEE Computer Society Press.

[Oka96] Chris Okasaki. *Purely Functional Data Structures*. PhD thesis, Carnegie Mellon University, September 1996.

[OT95] P. W. O'Hearn and R. D. Tennent. Parametricity and local variables. *Journal of the ACM*, 42(3):658–709, May 1995.

[OTTP95] P. O'Hearn, R. Tennent, M. Takeyama, and A. J. Power. Syntactic control of interference revisited. *Elsevier Electronic Notes in Theoretical Computer Science*, 1, 1995.

[Pau87] Lawrence Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*. Cambridge, 1987. Cambridge Tracts in Theoretical Computer Science, Volume 2.

[Pip97] Nicholas Pippenger. Pure versus impure Lisp. *ACM Transactions on Programming Languages and Systems*, 19(2):223–238, March 1997.

[Pit91] A.M. Pitts. Evaluation logic. In G. Birtwistle, editor, *IVth Higher Order Workshop, Banff 1990*. Springer, 1991.

[Plo77] Gordon Plotkin. LCF as a programming language. *Theoretical Computer Science*, 5, 1977.

[Plo80] Gordon D. Plotkin. Lambda-definability in the full type hierarchy. In Jonathan P. Seldin and J. Roger Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 363–373. Academic Press, London, 1980.

[Plo83] G.D. Plotkin. Domains. Technical report, Dept. Comp. Sci., University of Edinburgh, 1983.

[PR97] John Power and Edmund Robinson. Premonoidal categories and notions of computation. *Mathematical Structures in Computer Science*, 7(5):453–468, October 1997.

[PS93] Andrew M. Pitts and Ian D. B. Stark. Observable properties of higher order functions that dynamically create local names, or: What's new? In *Mathematical Foundations of Computer Science, Proc. 18th Int. Symp., Gdańsk, 1993*, volume 711 of *Lecture Notes in Computer Science*, pages 122–141. Springer-Verlag, Berlin, 1993.

[PS98] Andrew Pitts and Ian Stark. Operational Reasoning for Functions with Local State. In Gordon and Pitts, editor, *Higher Order Operational Techniques in Semantics*, pages 227–274. Cambridge University Press, 1998.

[PW88] Lewis Pinson and Richard Wiener. *An Introduction to Object-Oriented Programming and Smalltalk*. Addison-Wesley, Massachusetts, 1988.

[Red94] Uday S. Reddy. Passivity and independence. In *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 342–352, Paris, France, 4–7 July 1994. IEEE Computer Society Press.

[Rey78] J. C. Reynolds. Syntactic control of interference. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 39–46. ACM Press, 1978.

[Rey81a] J. C. Reynolds. *The craft of programming*. Prentice-Hall International series in computer science, C. A. R. Hoare (Ed.). Prentice-Hall International, Englewood Cliffs, NJ 07632, USA, 1981.

[Rey81b] John C. Reynolds. The essence of Algol. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372, Amsterdam, 1981. North-Holland.

[Rey82] J. C. Reynolds. Idealized Algol and its specification logic. In D. Neel, editor, *Tools and Notions for Program Constructions*, pages 121–161. Cambridge University Press, 1982.

[Rey83] John C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523, Amsterdam, 1983. Elsevier Science Publishers B. V. (North-Holland).

[Rey89] J. C. Reynolds. Syntactic control of interference, part II. In *Proceedings of ICALP '89*, LNCS 372, pages 704–722. Springer-Verlag, 1989.

[Rus98] C.V. Russo. *Types for Modules.* PhD thesis, University of Edinburgh, Edinburgh UK, 1998.

[RV95] Jon G. Riecke and Ramesh Viswanathan. Isolating side effects in sequential languages. In *Conference Record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95)*, pages 1–12, San Francisco, California, January 22–25, 1995. ACM Press.

[Sie96a] K. Sieber. Full abstraction for the second order subset of an ALGOL-like language. *Theoretical Computer Science*, 168(1):155–212, November 1996.

[Sie96b] Kurt Sieber. Full abstraction for the second order subset of an ALGOL-like language. Technical Report Feb8-7, Technical University of Munich, February 8, 1996.

[Sif82a] J. Sifakis. A unified approach for studying the properties of transition systems. *Theoretical Computer Science*, 18(3):227–258, June 1982.

[Sif82b] Joseph Sifakis. Global and local invariants in transition systems. In Mogens Nielsen and Erik Meineche Schmidt, editors, *Automata, Languages and Programming, 9th Colloquium*, volume 140 of *Lecture Notes in Computer Science*, pages 510–522, Aarhus, Denmark, 12–16 July 1982. Springer-Verlag.

[SRI91] V. Swarup, U. S. Reddy, and E. Ireland. Assignments for applicative languages. In R. J. M. Hughes, editor, *Functional Programming & Computer Architecture*, pages 192–214, Berlin, 1991. Springer-Verlag.

[Sta94] Ian Stark. *Names and Higher-Order Functions.* PhD thesis, University of Cambridge, December 1994. Also published as Technical Report 363, University of Cambridge Computer Laboratory.

[Sta96] I. Stark. Categorical models for local names. *J. Lisp and Symb. Comp.*, 9(1):77–107, February 1996.

[Sta97] Ian Stark. Names, Equations, Relations: Practical Ways to Reason about 'new'. Technical Report BRICS-RS-97-39, BRICS, September 1997.

[Ste85] G. L. Steele. *Common Lisp.* Digital Press, Burlington, 1985.

[Sto77] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics.* MIT Press, Cambridge, Massachusetts, 1977.

[Str73] C. Strachey. The varieties of programming language. Monograph PRG-10, Oxford University Computing Laboratory, Programming Research Group, Oxford, UK, 1973.

[Str87] Bjarne Stroustrup. *The C++ programming language.* Addison-Wesley series in computer science. Addison-Wesley, Reading, MA, USA, reprinted with corrections edition, 1987.

[TJ92] J.-P. Talpin and P. Jouvelot. The type and effect discipline. In *Proc. of 7th Annual Symposium on Logic in Computer Science*, pages 162–173, June 1992.

[Tur82] D. A. Turner. Recursion equations as a programming language. In J. Darlington, P. Henderson, and D. A. Turner, editors, *Functional Programming and its Applications.* Cambridge University Press, 1982.

[Tur86] D. A. Turner. Functional programming as executable specifications. In C. A. R. Hoare and J. C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 29–54. Prentice Hall, 1986.

[TWM95] David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *7'th International Conference on Functional Programming* *and Computer Architecture*, pages 1–11, La Jolla, California, June 1995. ACM Press.

[Ull94] J. D. Ullman. *Elements of ML Programming.* Prentice-Hall, 1994.

[Wad90] Philip Wadler. Linear types can change the world! In *Programming Concepts and Methods*, pages 561–581. North Holland, 1990.

[Wad92a] P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.

[Wad92b] P. Wadler. The essence of functional programming. *19th POPL*, pages 1–14, January 1992.

[Wal97] Lisa Walton. *Monads web page*, 1997. http://www.cse.ogi.edu/~walton/monads.html.

[Wir77] N. Wirth. Modula: A language for modular multiprogramming. *Software – Practice and Experience*, 7(1):3–35, January&February 1977.

[Wir88] Nikolaus Wirth. *Programming in Modula-2.* Springer, Berlin, 4 edition, 1988.

[Zei97] Stephen F. Zeigler. Comparing Development Costs of C and Ada. In *http://www.adauk.org.uk/pubs/zeigler.htm*, 1997.

Index

$\sigma{*}\tau$, 79
[ ], 46
$\_^*$, 48
$|\ \ |$, 51
$\triangleleft$, 56
η, 48
$\leq_{\sigma\to\tau}$, 58
$(\!|\ |\!)$, 46, 51
$[\![\ ]\!]$, 46, 50

Ada vs C, 71
AllStates, 23
amortized bound, 78

B-Notation, 71
bug/feature ratio, 71

`Can`, 35
$\mathtt{Can}_\sigma$, 35
$\mathtt{Can}_\sigma(\Gamma)$, 35
C-const-throughout Q, 26
C-indistinguishable throughout Q, 26
C-indistinguishable throughout$_\sigma$ R, 59
C-indistinguishable within$_\sigma$ R_1, R_2, 65
$\mathrm{CML}_{\mathrm{iML}}$
    terms, 46
    type expressions, 46
    type system, 47
computational invariant, 54
cond, 49
const-throughout$_\sigma$ R, 59
const-within
    for $Queue_1$, 109
    for $Queue_3$, 94
context, 22

defined-in, 23
Defn-R, 53
dom, 55
dom
    in Queue example, 85
dom, 23

E-op-indistinguishable throughout Q, 37
`Exp`, 35
$\mathtt{Exp}_\sigma$, 35
$\mathtt{Exp}_\sigma(\Gamma)$, 35
Extended ML, 71

iML
    operational semantics, 35
    terms, 33
    type expressions, 33
    type system, 34
indistinguishability, 24
    between tuples of values, 88
indistinguishable
    between $Queue_1$ and $Queue_3$, 97
indistinguishable throughout R, 59
`int list`, 79
invariant
    in spring example, 20

L
    a set of locations, 23

logical relation, 57
  in relating Set representations, 74
lookup, 49
lt, 49
L-value, 44

$\mathcal{M}$, 48
metalanguage
  computational λ_C, 45
Multics, 70
`myCtxt`, 42

N, 23
new, 49
`new`, 47
new, 64
New Jersey libraries
  Hash library, 75
no copying, 78

observational equivalence, 22
opaque data structure, 84
$s,M \Downarrow C,s'$, 35

partial bijection, 38
plus, 49

$\mathcal{Q}$
  reachable set, 23
Q, 23
Qref, 86
Qref, 85
`QSIG`, 76
Queue implementations, 77
$Queue_1$, 79
$Queue_2$, 80
$Queue_3$, 92
`QueueFour`, 89

R
  in $Queue_1$ and $Queue_3$, 85
ran, 55
reachable, 55
reachable, 55
reachable set, 23
R-value, 44

S, 23
semi-formal technique
  example, 31, 39
  examples, 62
  explanation, 30
set
  relations between two implementations, 74
set
  `signature`, 72
  ADT, 72
  algebraic properties, 73
`SetOne`
  x `structure`, 73
`SetTwo`
  x `structure`, 73
single-threadedness, 76
strong monad, 44

T
  in $Queue_1$ and $Queue_2$, 81
T_1, 86
T_2, 86
T_3, 94
transition relation
  pairs of, 65
transition system, 55
  regular, 55
typed applicative structure, 57

update, 49

valOf

in Queue example, 81

V-const-throughout Q

for functions of ground types, 28

for integer values, 27

for location values, 24

V-indistinguishable throughout Q

for functions of ground types, 28

for integer values, 27

for location values, 24

V-indistinguishable $throughout_\sigma$ R, 59

V-indistinguishable $within_\sigma$ R_1, R_2, 65

$V\text{-}indistinguishable_{\texttt{int*Q}\to\texttt{Q}}$

in $Queue_1$ and $Queue_3$, 98

in $Queue_3$, 95

$V\text{-}indistinguishable_{\texttt{Q}}$

in $Queue_1$ and $Queue_3$, 98

in $Queue_3$, 94

$V\text{-}indistinguishable_{\texttt{Q}\to\texttt{int*Q}}$

in $Queue_1$ and $Queue_3$, 99

in $Queue_3$, 95

V-op-indistinguishable throughout Q, 37