An Investigation of First-Class Locations in the Functional Machine Calculus

Max Bryars-Mansell

BSc Computer Science and Mathematics
University of Bath
2022-2023

An Investigation of First-Class Locations in the Functional Machine Calculus

Submitted by Max Bryars-Mansell

Copyright

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Abstract

Pointers, which are often associated with unsafeness in imperative languages, are a primary enabler of computational (side) effects, and thus, their integration into functional programming languages has been a challenge or considered counter-productive.

The Functional Machine Calculus is a recent hybrid calculus that combines functional and imperative semantics: with this, it provides a new avenue for exploring pointer-like objects. Our primary aim is to design and investigate a variant of this calculus that promotes these objects, called locations, to be first-class citizens. We propose that this enables an array of useful properties found in higher-level programming languages, like the C-family.

This project ultimately aims to contribute to the ongoing effort to unify the semantics of functional and imperative programming languages, creating a more powerful and flexible language paradigm.

Contents

Acknowledgements

I would like to thank my family, my girlfriend, my friends, and in particular my supervisor, Dr Willem Heijltjes, for the tremendous amount of support they provided me throughout this project.

Chapter 1

Introduction

Programming languages often have a trade-off between *control* and *safety*. On one end of the spectrum lies purely imperative languages, like those of the C-family, where we have full control over memory, data and operations which work on it, and thus, the means for producing fast, efficient and widely applicable programs. On the other end of the spectrum lies purely functional languages, like the *Haskell Programming Language* (n.d.), where we have higher-order expressibility and a guarantee that our program does only what we want, and thus, the means for producing safe, stable and optimised programs.

There is a desire to unify semantics from the functional and imperative worlds: both aspects are highly desirable in modern programming languages, as evidenced by languages like the *Rust Programming Language* (n.d.). Unfortunately, some semantics that are naturally imperative, like *effects*, are a struggle to replicate functionally without compromising useful properties and safety.

One concept which is often associated with imperative languages and unsafeness are *pointers*, which are a primary enabler of effects. Programming and computers in general are very reliant on pointers as a tool to carefully and precisely control/manipulate data in memory. However, they are the tool that will 'blow your foot off' when used badly. How might they fit into the semantics of a functionally inspired language?

Mathematical calculi, mostly stemming from the λ-calculus (Church, 1936), provide the inspiration and backbone for many functional programming languages. A recent functional calculus called the Functional Machine Calculus (Heijltjes, 2023) opens a new avenue for exploring pointer-like objects in the functional world, in particular, a direct analogue to pointers called *locations*. Locations were designed to provide semantics that enables the usage of reader/writer effects in the λ-calculus. This motivates *first-class locations*, which could potentially expand the capabilities and usefulness of the Functional Machine Calculus.

1.1 Aim

This project aims to design and investigate a variant of Functional Machine Calculus which promotes location objects to be first-class citizens. Being first-class, they can: be created at run-time, be used as function arguments, be returned from functions, and be referenced as variables. With semantics like this, location objects start to look like conventional pointers, and we investigate whether they would be useful in the Functional Machine Calculus and in functional programming more generally.

Chapter 2

Literature, Technology & Data Survey

This chapter presents direct (and related) background literature along with some important concepts of discussion. We start with computational (side) effects and the λ-calculus to provide a precursor to Functional Machine Calculus (FMC). With this, we discuss the FMC itself, why it has been developed, and how it has an analogue to pointers which we propose is useful. Finally, we finish by overviewing conventional pointers and why they are useful but also dangerous.

2.1 Effects

An effect is when a function or operation changes some state outside its local environment. Functions which don't rely on the presence of some external state to work are referred to as *pure functions*. Any program which does meaningful work will more than likely end up having some non-pure functions, like writing to an output stream or modifying a global state. This ties into the idea of *unsafe* code whereby we cannot guarantee with absolute certainty that the result of some computation will be what we desired or even exist at all.

There is much study towards understanding effects and how they interact with the rest of a program. Once we understand exactly how they affect our program, we can understand which transformations will fundamentally change our program. Thus, we can produce more efficient code and/or increase the understanding of our programs. Consider *referential transparency* of functions: the result of a function can be replaced with its resulting value and have no change to program behaviour. If we can prove this property holds for a function, then we can simplify our program and thus optimise it. The results of such studies have crucial applications in optimisation, safety and parallelism.

As mentioned, effects are often a struggle in functionally inspired programming. There have been many approaches for encoding effects, in particular, monads (Moggi, 1989) and thunks (Ingerman, 1961) have shown success. In the context of input/output (IO), thunks can defer the evaluation of an effectful IO operation (by wrapping in some higher-order function), thus, allowing it to be executed when it is safe to do so. However, they include no notion of sequenced actions. More generally, monads provide a way to both: encode effectful IO operations and define well-defined sequences of such operations.

It is common to associate effects in functional languages with an *imperative part* of the program which is kept distinct from the remaining *functional parts*. As described by Gifford and Lucassen (1986), these so-called *fluent* languages encourage a programming style where potentially unsafe sections of code are easily identified. Haskell provides *do* notation, which mimics the feel of imperative languages, to perform sequenced IO operations. This is possible by encoding an effectful action in typed values which can be composed together in a sequence with monad transformers (Peyton Jones and Wadler, 1993).

2.2 The λ-Calculus

The Functional Machine Calculus is inspired by the λ-calculus (Church, 1936), which is a system of mathematics which encodes computation. Its fundamental building blocks consist of *variables*, *abstractions*, and *applications*. By combining them, we build so-called λ-terms which can encode computational programs. The computational meaning of λ-terms are identified by observing their behaviour when performing *reduction* operations on them.

2.2.1 Grammar

We define λ-terms M (or N) inductively by

$$M, N \quad ::= \quad x \quad | \quad MN \quad | \quad \lambda x.\, M$$

which comprises three distinct constructors. From left to right we have: variables x, which, if bound, enable a λ-term to take (be substituted into) their place; applications MN, which encodes function application, i.e. apply function M with argument N; and abstractions $\lambda x.\, M$, which encodes function definitions, i.e. specify a parameter x for use in the function body M.

Example 2.1

The following terms are formed with the λ-calculus grammar

$$x \qquad \lambda x.\, xx \qquad (\lambda x.\, x)y$$

Note we use brackets for clarity.

2.2.2 Substitution

To understand the steps of computation in the λ-calculus, it is important to briefly discuss the process of *substitution*. The variable x in an abstraction $\lambda x.\, M$ is referred to as a *binding variable*. That is to say, applying N, using an application, means that we should substitute N into every *bound* occurrence of variable x in the abstraction body M. To denote this, we write $\{N/x\}M$ to mean substitute N for x in M. Bound occurrences of variable x refer to ones which are scoped to a binding abstraction. Conversely, *free* occurrences of variables refer to ones which are not scoped to any binding abstraction. We often require that substitution is *capture-avoiding*, which means it doesn't cause free variables to become bound. This ensures that our substitution doesn't change the computational meaning of terms. We use a process called α-conversion to provide a mechanism for renaming variables without changing their status of being bound or free.

2.2.3 Reduction

Given some λ-term, we wish to assign it a computational meaning: we do this with a process called reduction, which provides the mechanism to compute. The most important form of reduction is called β-*reduction*, which captures the process of applying functions with arguments. This is possible when an abstraction term is applied with some argument with an application. We call this a *redex* (reducible expression), and thus, we can apply β-reduction. Given some λ-term, β-reduction can be thought of as taking a computational step.

Additionally, there is η-reduction, which captures the process of converting a function to an equivalent (more simple) form by eliminating unnecessary abstractions (function parameters).

Example 2.2

The term

$$\underline{(\lambda x.\, xx)\; M}$$

encodes a program that applies the term M with itself by using the abstraction $\lambda x.\, xx$. This has a redex (underlined), so we can apply β-reduction which gives us the λ-term MM.

We can encode computation by building λ-terms that β-reduce to fixed known forms that have no redexes. These are referred to as *normal forms* and the Church–Rosser theorem (Church and Rosser, 1936) implies that all terms will β-reduce, irrespective of reduction strategy, to the same normal form if it exists. This property is called *confluence*. However, it is possible to build λ-terms with no normal form in the (untyped) λ-calculus.

2.3 The Functional Machine Calculus

Confluence in λ-calculus (see section 2.2.3) is an extremely useful property since it allows us to reason about computation outside of the reduction strategy. With this, we can make certain assumptions about our programs which gives much potential for optimisation and safety. On top of this, it is highly expressible and has powerful typing rules which are useful for programmers.

The λ-calculus has no way to encode effects (see section 2.1) without breaking confluence. This forces us to follow a particular reduction strategy. Because of this, much literature has focused on modifying/generalising the λ-calculus to achieve an encoding of effects that is practical enough for real-world functional programming languages.

The unfortunate downside of many approaches is that they are strict and they introduce cumbersome constructions. To address this, Heijltjes (2023) describes The Functional Machine Calculus (FMC) as "a simple model of higher-order computation with 'reader/writer' effects", which fundamentally encodes a notion of effects in an attempt to make them more natural to work with.

It builds upon fundamental concepts from the λ-calculus by introducing various generalisations designed to improve applicability and usefulness - most importantly, an encoding for reader/writer effects. Its backbone consists of two generalisations called *locations* and

sequencing. It is important to note that these generalisations are designed to be sufficiently subtle to allow for desired semantics to appear naturally.

2.3.1 Locations

A notable observation about the λ-calculus is that we can reason about it from the perspective of a machine operating on a stack of λ-terms. An application $M\,N$ pushes the term N to the stack and continues with M. An abstraction $\lambda x.\,M$ pops a term N from the top of the stack and continues with $\{N/x\}M$, where $\{N/x\}$ is the capture-avoiding substitution of N into M. Extending this idea, Heijltjes (2023) describes a λ-calculus generalisation called *poly-λ-calculus*, which enables us to work with multiple stacks that are each accessible using fixed objects called *locations*.

Locations come from a set A which is ranged over by identifiers $a, b, c, \cdots$. For applications and abstractions, we parameterise a location to specify the stack they act upon. Importantly, the regular λ-calculus embeds in this generalisation through a dedicated main location called λ.

First-Class Locations

Locations are constant and predefined from the perspective of the machine. That is to say, the machine interacts with a fixed number of stacks: their identifiers (locations) are embedded in the program and cannot be mutated at run-time. The aim of this project is to make them into *values* the machine can directly interact with, namely, *first-class objects*.

This grants the ability to create much more dynamic programs: we can introduce new locations at run-time, we can parameterise functions with locations, and we can perform logic/arithmetic using locations. We propose that the implications of these semantics are extremely useful. For example, we can implement: mutable data structures, as mentioned by Heijltjes (2023); and recursive data structures (linked-lists, trees, etc.).

2.3.2 Sequencing

Inspired by the stack machine perspective, λ-terms are a sequence of push/pop actions that must end in a variable. By relaxing the variable-must-go-last restriction and introducing a *skip* (*nil*) construct, we form a more fundamental notion of sequencing. With this viewpoint, λ-terms always end in skip (nil), much like imperative paradigms of computation. This prompts a change in perspective from the λ-calculus, whereby the result(s) of some computation is expected to be pushed to the stack rather than being some leftover term. With this idea, Heijltjes (2023) describes a λ-calculus generalisation called *sequential-λ-calculus*, which enables this idea of sequentially composing terms together.

2.3.3 Grammar

We define terms M (or N) inductively by the grammar

$$M,\, N \quad ::= \quad \star \;\mid\; x.\,M \;\mid\; [N]a.\,M \;\mid\; a\langle x\rangle.\,M$$

which comprises four distinct constructors. From left to right we have: *skip* (or *nil*) $\star$, which encodes the imperative *skip* that we expect our terms to end with; (sequential) variables $x.\,M$,

which, if bound, enable a term to take (be substituted into) their place; applications $[N]a.\,M$, which push a term N to the stack identified by location a; and abstractions $a\langle x\rangle.\,M$, which pop a term from the stack at location a and bind it in variable x.

We often omit the trailing $.\star$ when writing terms. Furthermore, we often omit the location λ (which embeds regular λ-calculus) when parametrising applications and abstractions that act upon the regular λ-stack.

2.3.4 Machine

An important distinction is that, unlike the λ-calculus, computation happens *imperatively* on a stack machine. That is to say, the FMC *does* computation by executing a sequence of operation-encoding terms with a stack machine, whereas, the λ-calculus *encodes* computation in the process of reducing a term. Importantly, we still preserve the concept of reduction, which provides a notion of *compile-time* execution of the machine. This provides a way to identify and resolve instances of referential transparency before the machine is executed.

To define a machine, we require memory and state:

- A *stack* of terms S is given by

$$S \;::=\; \epsilon \mid S \cdot M$$

  where ϵ denotes an empty stack and M is the term at the top of the stack. As mentioned by Heijltjes (2023), we should consider this definition as coinductive to include infinite collections of terms called *streams*.

- Given a set of locations A, a *memory* S_A is a family of stacks/streams given by

$$S_A = \{S_a \mid a \in A\}$$

  where stack S_a is the stack corresponding to location a. We write $S_A; S_a$ to identify stack S_a in memory S_A.

- A *state* is a pair (S_A, M) of a memory S_A and a term M. A *run* of the machine is a sequence of *steps*, written

$$(S_A, M) \Downarrow (T_A, N) \qquad \text{or} \qquad \frac{(S_A;\ M)}{(T_A;\ N)}$$

  which mutates the memory S_A to T_A by executing some sequence of push/pop actions given by M reducing to N.

We use *stack diagrams* to show how the state evolves through each machine step, which visualises a machine run. We define the machine by giving rules for each term constructor. In particular, the rules for application and abstraction are given by

$$\frac{(S_A\ ;\ S_a \quad ,\ [N]a.\,M)}{(S_A\ ;\ S_a \cdot N \ ,\ M)} \qquad\qquad \frac{(S_A\ ;\ S_a \cdot N \ ,\ a\langle x\rangle.\,M)}{(S_A\ ;\ S_a \quad ,\ \{N/x\}M)}$$

which shows how the memory changes from the first state (top) to the next state (bottom).

Example 2.3

The term

$$a\langle x\rangle.\,[x]b.\star$$

encodes a program that: pops (via an abstraction) a term from the location a and binds it in variable x; then pushes (via an application) x, which is bound in the preceding abstraction, to the stack identified by location b. It has the following run of the machine which is initialised with term N at the top of stack S_a.

$$\begin{array}{rl} (S_A & ; S_a \cdot N \;\; ; S_b \quad , \quad a\langle x\rangle.\,[x]b.\star) \\ (S_A & ; S_a \quad\;\; ; S_b \quad , \quad [N]b.\star) \\ (S_A & ; S_a \quad\;\; ; S_b \cdot N \quad , \quad \star) \end{array}$$

2.3.5 Encoding Reader & Writer Effects

To use input/output effects, we define dedicated location stacks (or streams) which act as *input* and *output* streams. With this, reading is done by popping from the location *in*, and printing is done by pushing to the location *out*.

Example 2.4

The term

$$\langle x\rangle.\,[x]out$$

encodes a program that: pops a term from the regular λ-stack location and binds it in variable x; then pushes x, which is bound in the preceding abstraction, to the output stream location.

Similarly, we can define *memory cells* as stacks of depth one which store a single value. We can define functions for set, which updates a cell with a new value; and get, which fetches the value from a cell.

$$\begin{aligned} \text{get}\ a &\triangleq a\langle x\rangle.\,[x]a.\,[x] \\ \text{set}\ a &\triangleq \langle x\rangle.\,a\langle \_\rangle.\,[x]a \end{aligned}$$

Notice the use of _ in the abstraction, which denotes a variable that is free everywhere in its body, i.e. it is unused.

Interestingly, this looks very similar to *C-Style file input/output* (n.d.), whereby input/output streams are identified with pointers to *FILE* objects. The API provides various functions to operate on these stream pointers, i.e. get data or set (write) data. Note: the standard input and output streams are identified by fixed *FILE* objects called *stdin* and *stdout*, respectively. In the context of the FMC, a location directly parallels a pointer to some *FILE* object. In particular, *stdin* and *stdout* are mirrored by fixed *in* and *out* locations.

2.4 Pointers

A powerful ability in many imperative programming languages is the ability dynamically read/write sections of memory with pointers. Programming languages often provide a *new* (or *malloc*) function which returns a pointer to some available section of (heap) memory that has a specified size. With this, we have the ability to dynamically create (and free) memory for storing program data.

Pointers enable some useful semantics for programmers, including:

- We can allocate and deallocate memory dynamically, which is essential for programs that need to work with data structures of varying sizes or those that must manage memory efficiently.
- We can share data between functions and across program components, which allows different parts of a program to access and manipulate the same data in memory.
- We can interface with and manipulate foreign code, which refers to external code, such as operating system calls or libraries, that are not part of our program.

2.4.1 Pointer Issues

While pointers provide a lot of power and flexibility, they also come with risks, such as the potential for memory leaks, buffer overflows, and other security vulnerabilities. These issues are the primary reason why functional languages avoid their use (in the conventional way), or provide strict safety features to manage them.

- *Null pointers* are a special pointer value reserved for cases when we wish to specify a pointer-to-nothing. This is useful (and required) for identifying uninitialised pointers. We frequently have to check whether a pointer is null in order to avoid *dereferencing null*.
- *Pointer aliasing* is a problem that can occur when two different pointers (as variables) refer to the same section of memory. It can result in data dependency issues, for example, we might write to the same place we read and affect our result.
- *Undefined behaviour* (in the context of pointers) often refers to cases where we end up accessing sections of memory we're not meant to. This can happen if memory is not initialised or pointer-arithmetic gives a *bad* pointer.
- *Memory leaks* are a problem that occurs when memory is claimed (*malloc'd*) from the system but never released (*free'd*). A program which continuously claims more memory without freeing it can result in out-of-memory errors. Languages take many approaches to solve this: garbage collection, borrow checking (Pearce, 2021), smart pointers, etc.

Similarly to functionally inspired languages, by introducing restrictions we can form concrete assumptions about how our program operates. We can use this to spot and reduce instances of these errors, which takes some pressure off the programmer.

2.4.2 Pointer-Location Parallels

Pointers identify memory cells of system memory by using unique integer indices associated with them, and therefore, pointers convert to integers and vice-versa. Usually, languages abstract over the idea of integer-indexed memory cells by using types to semantically differentiate them from other values. In the C-family, pointers are identified with a type-modifier (*C Language Type*, n.d.), (*C Language Pointer*, n.d.). Comparatively, locations in the FMC are a label which refers to a stack whereby push/pop operations interact with the top term only. Importantly, we can encode memory cells in the FMC by modelling them as a stack of depth one.

Any local (stack) variable has a pointer given by the memory cell where it resides (on the stack). Converting it to a pointer is often referred to as an *address-of* operation. The FMC has no direct analogue of this since we only have access to terms at the top of the stack.

2.4.3 Functional Languages With Pointers

The *OCaml Programming Language* (n.d.) provides a rather basic implementation of pointer-like objects alongside its more conventional functional semantics. Although their use is generally discouraged, there are cases where they can be useful (or necessary). The language mostly deals with pointers in an implicit way, i.e. the compiler decides where and when to use them. Although, the programming can be explicit if such a use-case arises. As mentioned in the documentation (*OCaml Pointers*, n.d.), this is most likely to occur when translating from an imperative-style program. Unlike our proposal for first-class locations in the FMC, pointers in OCaml are not fundamental components of the language (as the type system is etc.).

While OCaml pointers are only used to represent pointers to memory locations, FMC locations can represent other mutable states, such as files, network connections, etc. Additionally, locations allow for more precise control over effects by associating them with specific locations, while OCaml effects are managed using a global state monad, which is a harsh approach.

Chapter 3

Overview of Aims

The primary goal of this project is to *design* how first-class locations could look and work in the Functional Machine Calculus, and to *investigate* the implications of their introduction.

To achieve this, we will produce a formal definition of them by modifying the existing definitions given by Heijltjes (2023). Additionally, we will implement an interpreter for our modified calculus and use it to theoretically and practically investigate our modified calculus.

It is important to note that we do not desire to implement fully 'safe' first-class locations as this is unfeasible for a project of this scope. This project is an initial dive into their potential design and immediate effects by means of a naive definition and implementation. This is because, unfortunately, there are many issues which are difficult to tackle due to the novelty of the FMC.

To review our results, we discuss ideas that the semantics of conventional pointers enable. In particular, we make parallels to the C-family by implementing:

- Mutable data structures, as briefly identified by Heijltjes (2023): consider a struct-like collection of data (*C Language Struct*, n.d.) where related values are stored structurally in memory.
- Recursive data structures: consider dynamic linked-lists or graphs, whereby the *location* of the next element is specified by some *value*.
- Call-by-reference: locations can be passed to functions which can use/modify its store.

To investigate the implications of introducing first-class locations to the Functional Machine Calculus, we ask:

- How does it affect applicability, usefulness and practicability: can we implement things we couldn't before; or can we create smaller and more efficient implementations?

Chapter 4

Designing First-Class Locations

In this chapter, we define how our first-class locations look in the FMC. To begin, we introduce our ideas with a less formal approach by providing important intuition and justification for them. As we build upon our ideas and literature, we move towards a formal definition of our calculus, called the First-Class Location Functional Machine Calculus (or FCL-FMC).

4.1 Ideas

A problem we must tackle is that of creating and introducing *locations* as terms and distinguishing them from existing terms. Terms themselves can be pushed to or popped from the machine's stacks as values. With this in mind, naively allowing any term to be a location prompts an important question: what makes a location *useable* and *valid*? In particular, does some term consisting of abstractions and applications make any sense as a location, and furthermore, what happens if a location value refers to a stack that doesn't exist.

If we restrict our interpretation of locations to specific term structures, then perhaps encodings like Church Numerals would work, although, they may not make sense for practical reasons (they are verbose). Hence, it is better to provide some set of primitive location terms with a way to convert between them and locations.

It is likely that types would help to solve these problems by adding restrictions to the kind of values we accept as a location. We could enforce that terms used as location values must result in a location type. However, as it stands, types (in the FMC) are far from complete, hence, a solution like this is out of the scope of the project. Unfortunately, this means that we will likely encounter cases where our locations are *invalid*; this is a very common issue with memory models and work has been done to address it (Kang et al., 2015).

4.2 Location Values

To start with, we need some notion of what it means to be a *location value*. Given a set of locations A that are ranged over by identifiers a, b, c, ..., we define their values to be the terms called $\#a$, $\#b$, $\#c$, $\cdots$. The exact encoding of location values doesn't matter since we are just interested in them being well-defined (and invertible) for all locations. For example, we could use Church Numerals (Church, 1936), however, they are impractical (see section 4.1), hence, instead, take them as primitives.

If location values were some object besides terms, we would need to construct our machine with the ability to store both terms and locations on stacks. By making the location values be terms, we simplify the process of extending our machine. Moreover, we enable the ability to apply functions to locations. With an encoding that has well-defined binary operations like *add* and *subtract*, we can model pointer arithmetic (we will see this in section 6.1.5). It is worth noting that it would also be possible if they were not terms by defining built-in function symbols and δ-rules.

We can view $\#$ as a function $\# : \mathit{Location} \to \mathit{Term}$ which maps locations to terms, and we read $\#a$ as *take value of* location a. Now, we want to have an operation which reverses $\#$ to give a location from a *valid* $\#a$. Define $@(\#a) = a$, then

$$\begin{array}{cc} @\,(\#a) = a & \#(@M) = M \\ \text{for any location } a & \text{for any term } M \text{ of form } \#a \text{ for some } a \end{array} \tag{4.1}$$

and so $@$ gives us an inverse of $\#$ when restricted to the domain of valid location terms. Thus, we have $@ : \mathrm{im}\,\# \to \mathit{Loc}$ where $\mathrm{im}\,\#$ denotes the image of $\#$. Thus, we can form a bijection $\mathit{Term} \supset \mathrm{im}\,\# \cong \mathit{Location}$ between terms and locations, given by $@$. Because of this, it is not possible to convert any term M to a location (see section 4.2.1) and we must proceed with care.

Given some location a, we wish to push it to the stack, as a term, in order to manipulate it. To differentiate from standard applications, we introduce a new application $[\#b]a.\,M$, called a *location application*, on location a where location b is pushed to the stack as its location value, and we continue with body M. It has the effect of pushing the term $N = \#b$ to stack a.

Example 4.1

We can write

$$[\#\mathit{out}]a$$

to be the term which pushes the value of *out* to the stack identified by a.

We are somewhat inspired by the & operator of C-family languages whereby $\&x$ gives a pointer value of a variable x. We can take the viewpoint that a location a acts like a variable name, and its location value $\#a$ acts like a pointer to the variable. In fact, stacks of depth one (memory cells) with an identifying location directly parallel stack variables of conventional imperative languages (we will see this in section 6.1.1). We have made sure to use a different symbol $\#$ in location abstractions to avoid confusion since our operation is fundamentally different in that it works on a location rather than a variable. In particular, we treat the value $\#a$ of a location a as a pointer-like object that can be manipulated on the stack.

With this, given some location value on a stack, we wish to pop it and use it as a location parameter for abstractions and applications. To do this, we make location identifiers $a, b, c, \cdots$ into variables, called *location variables*, and provide a substitution mechanism via abstractions. To differentiate from standard abstractions, we introduce a new abstraction $a\langle @b\rangle.\,M$, called a *location abstraction*, on location a where location variable b becomes bound in the body M, and we continue with body M. It has the effect of popping some term N from stack a and binding $@N$ in location variable b. As discussed earlier, this assumes N is of the form $\#c$ for some location c, and we have an error if it is not (see section 4.2.1).

Example 4.2

We can write

$$[\#out].\,\langle @a\rangle$$

to be the term which binds the value of out in the location variable a.

We are further inspired by the * operator of C-family languages whereby $^*p$ dereferences a pointer p, which enables reading and writing of the data it points to. In particular, the location abstraction $a\langle @b\rangle$ can be viewed as dereferencing and binding whatever location value is at the top of the stack as location variable b. Thus, much like location applications, we have chosen a different symbol to avoid confusion.

It is important to note that we consider both location applications and location abstractions to be new constructors whereby $\#$ and $@$, respectively, are their differentiating notation. The primary reason for this is that we want a distinction between abstractions that bind terms in variables and (location) abstractions that bind locations in (location) variables. However, with types, it may be more useful to consider them as pattern matching in existing application and abstraction constructors because we would have distinction in a more generalised way (with types).

4.2.1 Location Usability & Validity

As identified in section 4.1, we face the problem of usability and validity. In particular, how can we be sure that a location value is useable as a location, and furthermore, whether it refers to a valid stack? For example, it doesn't *always* make sense to write

$$[M].\,\langle @a\rangle.\,a\langle x\rangle$$

as the term M may not be a valid or even useable location value (see section 4.2.1), and thus, it would be incompatible with the machine. This is another reason for us to provide a clear distinction between location applications and abstractions in our typeless calculus.

Types would be an ideal solution here as they would restrict the way in which we can write and reduce terms so that they evaluate, through reduction or the machine, to valid computations. Unfortunately, types are out of the scope of this project, so we lack the luxury of their imposed restrictions. Therefore, in our calculus, we can easily form *bad* terms which do not reduce or execute on the machine. In a typed variant, bad terms would be *incorrect*, hence, we could not form them.

An Alternative Approach

An alternative approach is to restrict what can be pushed/popped to a set of primitives. To do this, we start by defining this set of *primitive* values which have the property that any primitive can be converted/interpreted as a location, i.e. integers as memory cell indices. With this, we need not worry about whether popped values can be *used* as locations. However, this eliminates first-class functions whereby we can push/pop terms themselves. Because of this, we require higher-order functions like conditionals and loops to be implemented separately as δ-reduction rules, which makes the calculus much more like conventional imperative languages. This loss of expressibility is undesired, and thus, we steer clear of this approach.

4.2.2 Fresh Locations

With location values defined, we need a way to introduce fresh locations. To do this naturally, we provide a fixed location called *new* which contains a stream of fresh location values. Pulling them with a location abstraction allows us to introduce a fresh location to our program.

Example 4.3

We can write

$$new\langle @a \rangle$$

to be the abstraction which pulls and binds a new location a ready for use.

With the introduction of fresh locations, we must provide a notion of *empty locations*. This is because we need some way to indicate that a location value is not valid or not ready for use. This is useful for implementing recursive data structures, like linked-lists, where we need some way to indicate the recursive base case. In many higher-level programming languages, empty pointers are referred to as *null pointers* or *pointers to null*. Thus, we take inspiration and call our empty location *null*. If *new*, for some reason, doesn't contain a fresh location, then it will provide *null* to represent an invalid location. This is paralleled in C-family languages whereby *new* or *malloc* will return a *null pointer* if there was a failure, i.e. we are out of memory.

Interestingly, the *null* location parallels the idea of the *null* streams, whereby anything written to them will be discarded. For example, Unix-based systems provide a null file which discards anything that is written (piped/redirected) into it. In our case, we define a similar notion whereby pushing to *null* can be thought of as discarding the pushed term or location value.

4.3 Formal Definition

We now pull together our design for first-class locations and build a formal definition of our calculus. Since our changes extend the FMC, much of the existing work carries forward from Heijltjes (2023) (including that of Barrett, Heijltjes and McCusker (2023)). Our formal definition must include the following components: grammar constructors to describe how terms are built, composition and substitution rules to describe how terms interact with each other, and machine/reduction rules to describe how computation works.

4.3.1 Grammar

We extend the original grammar to include our two new constructors identified in section 4.2.

Definition 4.1 [Grammar]

A term M (or N) is given inductively by

$$M, N ::= \star \mid x.\, M \mid [N]a.\, M \mid a\langle x \rangle.\, M \mid [\#b]a.\, M \mid a\langle @b \rangle.\, M$$

for variables x and location variables a and b.

From left to right the constructors are called: *Nil*, (Sequential) *Variable*, *Application*, *Abstraction*, *Location Application* and *Location Abstraction*.

As usual, for clarity, we often omit the trailing $.\star$ and the regular λ location from our terms.

Example 4.4

The following terms are formed from our grammar

$$[x]out.\langle f\rangle.f \qquad [\#out].\langle f\rangle \qquad new\langle @p\rangle.\langle x\rangle.[x]p$$

4.3.2 Composition

We define how *composition* of terms behaves with our two new constructors, location application and location abstraction.

Definition 4.2 [Composition]

Composition $M\,;N$ of terms M with N is given as standard with the addition of the two rules

$$\begin{aligned} [\#b]a.M\,;P &= [\#b]a.(M\,;P) \\ a\langle @b\rangle.M\,;P &= a\langle @b\rangle.(M\,;P)\ (b \notin flv(P)) \end{aligned}$$

where $flv(P)$, read as *free location variables*, is the set of location variables which are *free* in P, i.e. location variables which are not within the scope of a binding location abstraction. We require this so that b doesn't become bound in P.

4.3.3 Substitution

We first define how (variable) *substitution* of terms behaves with our two new constructors, location application and location abstraction.

Definition 4.3 [Capture-Avoiding Substitution]

Capture-avoiding substitution $\{N/x\}M$ of a term N for variable x is given as standard with the addition of the two rules

$$\begin{aligned} \{P/x\}[\#b]a.M &= [\#b]a.\{P/x\}M \\ \{P/x\}a\langle @b\rangle.M &= a\langle @b\rangle.\{P/x\}M \end{aligned}$$

which only needs to inductively apply the substitution to the body.

To give meaning to location abstractions (with respect to the machine), we must introduce *location substitution*. We write $|b/a|M$ (for distinction from variable substitution) to mean substitute location b for location variable a in M.

First the original constructors.

Definition 4.4 [Capture-Avoiding Location Substitution 1]

For standard constructors, nil, (sequential) variable, application and abstraction, capture-avoiding location substitution $|b/a|M$ is given by the rules

$$\begin{array}{rcl}
|b/a|\star & = & \star \\
|b/a|x.\,M & = & x.\,|b/a|M \\
|b/a|[N]a.\,M & = & [|b/a|N]b.\,|b/a|M \\
|b/a|[N]c.\,M & = & [|b/a|N]c.\,|b/a|M \quad (a \neq c) \\
|b/a|a\langle x\rangle.\,M & = & b\langle x\rangle.\,|b/a|M \\
|b/a|c\langle x\rangle.\,M & = & c\langle x\rangle.\,|b/a|M \quad (a \neq c)
\end{array}$$

which changes the location (if it matches) and inductively applies the substitution to the body.

Next our new constructors.

Definition 4.5 [Capture-Avoiding Location Substitution 2]

For the two new constructors, location application and location abstraction, capture-avoiding location substitution $|b/a|M$ is given by the rules

$$\begin{array}{rcl}
|b/a|[\#a]a.\,M & = & [\#b]b.\,|b/a|M \\
|b/a|[\#a]c.\,M & = & [\#b]c.\,|b/a|M \quad (a \neq c) \\
|b/a|[\#d]a.\,M & = & [\#d]b.\,|b/a|M \\
|b/a|[\#d]c.\,M & = & [\#d]c.\,|b/a|M \quad (a \neq c) \\
|b/a|a\langle @a\rangle.\,M & = & b\langle @a\rangle.\,M \\
|b/a|c\langle @a\rangle.\,M & = & c\langle @a\rangle.\,M \quad (a \neq c) \\
|b/a|a\langle @d\rangle.\,M & = & b\langle @d\rangle.\,|b/a|M \quad (b \neq d) \\
|b/a|c\langle @d\rangle.\,M & = & c\langle @d\rangle.\,|b/a|M \quad (a \neq c, b \neq d)
\end{array}$$

which changes the location (if it matches) and inductively applies the substitution to the body. Here we see the one *interesting* rule, whereby, substituting for a bound location variable (of a location abstraction) leaves the body unchanged to avoid capture.

These rules give an exhaustive definition of capture-avoiding location substitution because it is possible to satisfy all conditions by applying α-conversion.

Example 4.5

Substituting c for a in $a\langle f\rangle.\,[\langle @a\rangle.\,[f]a]b.\,[f]a.\,f$ gives the following propagation of rules

$$\begin{aligned}|c/a|\left[a\langle f\rangle.\,[\langle @a\rangle.\,[f]a]b.\,[f]a.\,f\right] &= c\langle f\rangle.\,|c/a|[\langle @a\rangle.\,[f]a]b.\,[f]a.\,f\\ &= c\langle f\rangle.\,[|c/a|\langle @a\rangle.\,[f]a]b.\,|c/a|[f]a.\,f\\ &= c\langle f\rangle.\,[\langle @a\rangle.\,[f]a]b.\,[f]c.\,|c/a|f\\ &= c\langle f\rangle.\,[\langle @a\rangle.\,[f]a]b.\,[f]c.\,f\end{aligned}$$

4.3.4 Machine

We extend the machine given by Heijltjes (2023) by introducing two new machine rules, for location applications and location abstractions.

Definition 4.6 [Machine]

For the two new constructors, location application and location abstraction, a machine step is given by the rules

$$\frac{(S_A\,;\,S_a\quad,\quad[\#b]a.\,M)}{(S_A\,;\,S_a\cdot\#b\quad,\quad M)}\qquad\qquad\frac{(S_A\,;\,S_a\cdot N\quad,\quad a\langle @b\rangle.\,M)}{(S_A\,;\,S_a\quad,\quad|@N/b|M)}$$

where $@N$ is given by equation (4.1).

As mentioned in section 4.2.1, N must be of the form $N = \#c$ for some $c \in A$. If it is not, then $@N$ is undefined, and thus, we cannot step the machine. In this project, we assume terms are *valid*, which means this problem will never occur. However, it is still possible to write bad terms, and thus, we make sure that our demo implementation (see section 5) produces an error in this case.

Example 4.6

The term $[\#out].\,\langle @a\rangle.\,[0]a$ has the following run of the machine, where the terms 0 and $\#out$ are taken to be some encoding (or primitives).

$$\begin{array}{l}(S_{out}\quad;\epsilon_\lambda\quad,\quad[\#out].\,\langle @a\rangle.\,[0]a\quad)\\ \hline (S_{out}\quad;\epsilon_\lambda\cdot\#out\quad,\quad\langle @a\rangle.\,[0]a\quad)\\ \hline (S_{out}\quad;\epsilon_\lambda\quad,\quad[0]out)\\ \hline (S_{out}\cdot 0\quad;\epsilon_\lambda\quad,\quad\star)\end{array}$$

Given a set of locations A, we pick a subset of locations (possibly infinite) $\{p_1, p_2, p_3, \cdots\} \subseteq A$ to be our fresh locations. The number of locations we choose here corresponds to how many locations we wish to make available as fresh. With this, we define a dedicated *new* location $new \in A$. The machine is initialised with stream $S_{new} = \cdots \#p_3 \cdot \#p_2 \cdot \#p_1$.

With this, we can introduce fresh locations by popping from new.

$$\frac{(S_A\ ;\ S_{new} \cdot \#p_1 \quad ,\quad new\langle @p\rangle.\ M)}{(S_A\ ;\ S_{new} \quad ,\quad |p_1/p|M)}$$

Additionally, we also have a dedicated *null* location $null \in A$ which collects terms that will be discarded during evaluation.

$$\frac{(S_A\ ;\ S_{null} \quad ,\quad [N]null.\ M)}{(S_A\ ;\ S_{null} \cdot N \quad ,\quad M)}$$

Moreover, the location value $\#null$ acts as the empty location in cases where a valid location has not been obtained (see section 4.2.2).

Finally, as standard, we have dedicated streams for *input*, *output*, *probabilistic sums*, *non-deterministic sums*, etc. It is important to note that *memory cells* become more dynamic in our calculus because we can introduce them during machine evaluation as opposed to statically during machine initialisation. In fact, this idea of dynamic memory cells plays a key role in modelling the semantics of imperative languages (see section 6.1).

Example 4.7

The term $new\langle @p\rangle.\ [5]p.\ [2]p$ has the following run of the machine, where the new location is initialised with a stream that has $\#a$ at the head, and the terms 5, 2, and $\#a$ are taken to be some encoding (or primitives)

$$\begin{array}{lllll}
(S_{new} \cdot \#a & ;\epsilon_a & \epsilon_\lambda & , & new\langle @p\rangle.\ [5]p.\ [2]p) \\
\hline
(S_{new} & ;\epsilon_a & ;\epsilon_\lambda & , & [5]a.\ [2]a) \\
\hline
(S_{new} & ;\epsilon_a \cdot 5 & ;\epsilon_\lambda & , & [2]a) \\
\hline
(S_{new} & ;\epsilon_a \cdot 5 \cdot 2 & ;\epsilon_\lambda & , & \star)
\end{array}$$

4.3.5 Reduction

We extend β-rules by introducing rules for successive location application and abstraction. Our definition is loose because allowing locations to bind presents some challenges.

Definition 4.7 [β-Rules]

$$\begin{array}{lll}
[\#p]a.\ a\langle @q\rangle.\ M & \to_\beta & |p/q|M \\
[\#p]b.\ a\langle @q\rangle.\ M & \to_\beta & a\langle @q\rangle.\ [\#p]b.\ M \qquad \text{if } a \text{ and } b \text{ are } different^a \text{ and } q \neq p \\
[N]b.\ a\langle @q\rangle.\ M & \to_\beta & a\langle @q\rangle.\ [N]b.\ M \qquad \text{if } a \text{ and } b \text{ are } different^a \text{ and } q \notin blv(N)
\end{array}$$

where $blv(M)$ denotes the set of location variables bound in a term M.

[a] Referring to different stacks: location variables can be bound, and this can complicate matters. Specifically, location variables may refer to different stacks depending on their binding, which makes it difficult to keep track of their values.

In the final two rules, the inequality condition involving a and b implies that it is important to resolve location variables before checking their difference. Thus, we should first apply the first rule to ensure that location variables are fully resolved before determining whether they are *different*. An example of this idea can be seen in example 4.8 whereby applying the first rule gives a clearer picture of whether the second rule is valid.

Example 4.8

By applying the first rule of definition 4.7, we can β-reduce the following

$$[\#b]a.\,\underline{[\#a].\,\langle @p\rangle}.\,[\#b]p.\,a\langle @c\rangle \to_\beta \quad [\#b]a.\,[\#b]a.\,a\langle @c\rangle$$

It is important to note that the second rule does not apply to the underlined portion of

$$[\#b]a.\,[\#a].\,\langle @p\rangle.\,\underline{[\#b]p.\,a\langle @c\rangle}$$

because the location variables p and a do in fact refer to the same location - applying the first rule (above) showed us this.

Additionally, we extend η-rules by introducing rules for successive location abstraction and application. Like before, our definition is loose.

Definition 4.8 [η-Rules]

$a\langle @p\rangle.\,[\#p]a.\,M$	$\to_\eta$	M	if $p \notin blv(M)$
$a\langle @p\rangle.\,[\#q]b.\,M$	$\to_\eta$	$[\#q]b.\,a\langle @p\rangle.\,M$	if a and b are *different*[a] and $p \neq q$
$a\langle @p\rangle.\,[N]b.\,M$	$\to_\beta$	$[N]b.\,a\langle @p\rangle.\,M$	if a and b are *different*[a] and $p \notin blv(N)$

where $blv(M)$ denotes the set of location variables bound in a term M.

[a] Referring to different stacks for the same reasons as in definition 4.7.

Note that the two final η-rules are the same as reversing the second β-rule or vice-versa.

Example 4.9

Assuming locations a and b are fixed and not λ, we can apply β-reduction rules to

$$\begin{aligned}[\#b]a.\,\underline{[\#a].\,\langle @p\rangle}.\,[M].\,a\langle @c\rangle &\to_\beta [\#b]a.\,\underline{[M].\,a\langle @c\rangle} \\ &\to_\beta \underline{[\#b]a.\,a\langle @c\rangle}.\,[M] \\ &\to_\beta [M]\end{aligned}$$

Example 4.9 illustrates how reduction can be used to perform a *compile-time* optimisation by removing pushes/pops that are ultimately unnecessary.

Chapter 5

Developing an Interpreter

To investigate our calculus, we have developed an interpreter, called *cFMC*, for parsing and executing terms. It has provided a strong basis for investigation by providing useful debugging tools for insights into the workings of our calculus. A collection of demonstrations can be found in section A, and each solidifies the ideas from our analysis in section 6.

This project primarily focuses on the design and review of first-class locations in the FMC, hence, we don't include many details of implementation, however, we give a brief overview in section B.

5.1 Requirements

Here we give a high-level overview of the interpreter requirements: they have been developed with the rationale of giving us a working prototype for aiding our investigation.

- **Parsing terms**
  - The parser **must** be able to parse terms which agree with our calculus's grammar (see section 4.3.1).
  - The parser **should** provide informative error messages when a term is invalid.
- **Running the machine**
  - The machine **must** be capable of performing a run on our calculus's terms with a given state (see section 4.3.4).
  - The machine **must** support reader/writer effects through the use of dedicated locations for input and output (see section 4.3.4) called *in* and *out*, respectively.
  - The machine **must** support the introduction of fresh location through the use of a dedicated location (see section 4.2.2) called *new*.
  - The machine **must** support discarding of terms through the use of a dedicated location (see section 4.2.2) called *null*.
  - The machine **should** provide informative error messages when it is not able to run (see section 4.2.1).

- The machine **should** provide debugging information which displays the state, memory, and bound variables.

5.2 Design

5.2.1 Programs

To keep our parser simple, we opt to parse the same grammar designed in section 4.3.1. This way we can easily move between our mathematical theory and our practical implementation. Additionally, we streamline a common use case by adding some syntactic sugar for function definitions. To do this, we extend our grammar with a new construct called a *program*.

Definition 5.1 [Program]

A program P is given inductively by

$$P \ ::= \ \mathit{main}{=}(M) \mid f{=}(M)\ P$$

for term bodies M where f is the function identifier for $f = (M)$.

With this, the term M in $\mathit{main} = (M)$ is called the *entry point* of the program and the machine will be initialized with it. Functions $f = (N)$ are bound as variables f (to N), which streamlines the process of defining and using common functions. We usually write each function definition on a new line as seen in example 5.1, which mirrors conventional programming languages.

Example 5.1

The program

$$\mathit{print} = (\langle x \rangle.\, [x]\mathit{out})$$
$$\mathit{main} = ([0].\, \mathit{print}.\, [1].\, \mathit{print})$$

consists of a function called *print*, which prints a term, and it is used inside *main* to print 0 and 1. The program is equivalent to the term

$$[0].\, [\langle x \rangle.\, [x]\mathit{out}].\, \langle \mathit{print} \rangle.\, \mathit{print}.\, [1].\, [\langle x \rangle.\, [x]\mathit{out}].\, \langle \mathit{print} \rangle.\, \mathit{print}$$

which pushes and binds the function *print* each time it is used.

Function definitions like this make it much easier to use recursive functions without directly using fixed-point combinators.

5.2.2 Syntax Sugar & Quality of Life Features

Primitive Values

To reduce the verbosity of terms, we take numbers and location values to be primitives (rather than encodings). In our case, we take the type of numbers to be a signed 32-bit integer and the type of location values to be a string. This works by extending our grammar with a constructor for *value* which can either be a number or a location value.

Binary Operations

To complement our primitive number values, we support simple binary operations in the form of *plus* and *minus*. This works as standard for a stack machine, i.e. pop the previous two values and push back the result of the binary operation.

Example 5.2

The term

$$[2].\,[3].\,+\,.\,[7].\,-$$

adds 2 and 3 together, and then subtracts 7, which results in -2.

Grammar Omissions

We support the omission of trailing $.\star$ and omission of regular λ locations as parameters. To write regular λ location we use the full word *lambda*. Additionally, we also support the omission of variable names in abstractions by means of an underscore, i.e. $a\langle\_\rangle.\,M$ binds an unusable variable from a.

Conditional Cases

To further reduce the verbosity of terms, we add some syntactic sugar in the form of conditional *cases*, which are written

$$(c_1 \to M_1, c_2 \to M_2, \ldots, c_n \to M_n).\,N$$

for a sequence of values c_i with $i \in \{1, \ldots, n\}$. Each c_i *maps* to a term M_i. We have that

$$[c_i].\,\cdots.\,(c_1 \to M_1, c_2 \to M_2, \ldots, c_n \to M_n).\,N \;\triangleq\; \cdots.\,M_i\,;N$$

provided all applications/abstractions after $[c_i]$ do not modify the regular λ-stack. It has the effect of popping the term c_i at the top of the stack and composing the term N after the matched term M_i. With this, we can easily model if-else statements (which are possible, although verbose, with regular terms). Furthermore, we require an *otherwise* case that is selected if none of the others match. This looks like

$$(c_1 \to M_1, c_2 \to M_2, \ldots, c_n \to M_n, \mathit{otherwise} \to O).\,N$$

for a sequence of values c_i with $i \in \{1, \ldots, n\}$. As before, each c_i maps to a term M_i, and *otherwise* maps to a term O.

Example 5.3

The program

$$main = ([\#null].\,(null \to M, otherwise \to N))$$

matches $\#null$ at the top of the stack which results in the term $M\,;\star$ to give M.

Chapter 6

Analysis

Here we review our work by providing an analysis of the implications of introducing first-class locations: this will consist of practical examples/demonstrations and theoretical analysis/review.

- We show how first-class locations can be used to model aspects of C, in particular, variables, structs, recursive data structures, call-by-reference, and pointer arithmetic.
- We discuss the differences between the original calculus and our modified version. We demonstrate how many of the existing function definitions can become parameterised to form generalised versions. We look at how first-class locations enable much more flexibility with respect to moving terms around our program. Additionally, we propose a syntax for using functions that mirrors Haskell.

6.1 Modelling Aspects of C

6.1.1 Variables

Our first-class location semantics enable us to model C-like stack variable declaration, assignment and usage.

Listing 6.1: Declaring, assigning, and using stack variables in C.

```
int a = 5;
int b = 7;
int c = a + b;
```

The stack variables *a* and *b* in listing 6.1 can be accessed at any time provided they are in scope. Conversely, in the FMC, the machine can only access the top element of stacks, hence, to use terms further down the stack we have to pop all those above it (and possibly restore them). This results in many push/pop operations which pollute our terms and introduce verbosity. Instead, we should employ first-class locations and model variables as memory cells by creating new locations.

The code in listing 6.1 has a direct analogue (demonstrated in listing A.8) in

$$new\langle @a\rangle\ .\ [5]a\ .$$
$$new\langle @b\rangle\ .\ [7]b\ .$$
$$new\langle @c\rangle\ .\ a\langle x\rangle\ .\ [x]a\ .\ b\langle y\rangle\ .\ [y]b\ .\ [x+y]c$$

where we define the name of variables by creating a location abstraction binding with their name. Additionally, we take $x + y$ inside the application for succinctness - in reality, it would be a stack operation. Here: the abstractions which pull from location *new* act like declarations for our variables; the applications which push to our variable locations act like assignments to our variables. Additionally, these variable bindings exist until we are no longer in their scope.

We must make a clear distinction between copying a variable and making a pointer to a variable.

Listing 6.2: Copying a variable vs. making a pointer to a variable in C.

```
int a = 5;
int b = a;    // b is a copy of a
int *p = &a; // p is a pointer to a
```

In listing 6.2: b is a copy of a, which means we create a new memory cell for it and copy the value; p is a pointer to a, which means we refer to the same memory cell (with a new name). The analogue of this can be seen in

$$new\langle @a\rangle\ .\ [5]a\ .$$
$$new\langle @b\rangle\ .\ a\langle x\rangle\ .\ [x]a\ .\ [x]b\ .$$
$$[\#a]\ .\ \langle @p\rangle$$

where we first pull from *new* to create a new memory cell for b before copying the contents of a into it.

6.1.2 Structs

Our first-class location semantics enable us to model C-like struct definition, declaration, assignment and usage. In particular, we can assign structure to a collection of terms on a stack with functions to act upon it.

Listing 6.3: Defining, declaring, and assigning a pair struct in C.

```
struct Pair
{
  int x;
  int y;
};
```

The code in listing 6.3 has a direct analogue in

$$new\langle @p\rangle \,.\, [x]p \,.\, [y]p$$

where we define the name of stack variables by creating a location abstraction binding with their name and assign structure by ordering its components on the stack. The downside is that it is not obvious what the structure is, and thus, we have to be careful to pop elements in the correct order when using our struct. Additionally, to access x, we must pop and restore y. For these reasons, it is useful to define: a function (constructor) to create a struct; and auxiliary functions (accessors) to extract components of a struct. In this case, we can define: Pair to construct a pair; fst to extract x; and snd to extract y.

$$\begin{aligned}
\text{Pair} &\triangleq \langle y\rangle \,.\, \langle x\rangle \,.\, new\langle @p\rangle \,.\, [y]p \,.\, [x] \,.\, p. \,[\#p] \\
\text{fst} &\triangleq \langle @p\rangle \,.\, p\langle x\rangle \,.\, [x]p \,.\, [x] \\
\text{snd} &\triangleq \langle @p\rangle \,.\, p\langle x\rangle \,.\, p\langle y\rangle \,.\, [y]p \,.\, [x]p \,.\, [y]
\end{aligned}$$

With this, we can easily write a program that constructs a pair and prints its two elements.

$$\begin{aligned}
&[5] \,.\, [2] \,.\, \text{Pair} \,.\, \langle @p\rangle \,. \\
&[\#p] \,.\, \text{snd} \,.\, \langle y\rangle \,.\, [y]out \\
&[\#p] \,.\, \text{fst} \,.\, \langle x\rangle \,.\, [x]out
\end{aligned}$$

Notice how we use a stack of depth two in order to fit our two struct entries. For this reason, we can say that struct variables are modelled as stacks of depth n and primitive variables are modelled as stacks of depth one (memory cells).

6.1.3 Recursive Data Structures

Our first-class location semantics enable us to model C-like recursive data structures. These are structs (see section 6.1.2) whereby we store a first-class location which points to another struct (as the recursive part).

Listing 6.4: Defining a linked-list (element) struct in C.

```
struct LinkedList
{
  int        v;  // Value of element
  LinkedList *p; // Pointer to next element
};
```

We can see this design in listing 6.4: each element, as an instance of LinkedList, stores a value and pointer to the next element. We signify the end of the list by specifying NULL as the pointer value.

We can form a direct analogue by creating a LinkedList (constructor) function, which creates a head element with a given value v.

$$\text{LinkedList} \triangleq \langle v\rangle \,.\, new\langle @p\rangle \,.\, [\#null]p \,.\, [v]p \,.\, [\#p]$$

With this, we can easily construct a linked list by supplying a head value and getting a pointer to our head element. Notice our use of $\#null$ as the location value: this mirrors NULL in the C implementation. To append elements, we define the function push_back which adds an element with a given value v to the end of the list.

$$\begin{aligned}\text{push\_back} \triangleq\ & \langle v\rangle\ .\ \langle @p\rangle\ .\ p\langle pv\rangle\ .\ p\langle @pp\rangle\ .\ [\#pp]\ .\ (\\ & \qquad null \rightarrow [v]\ .\ \text{LinkedList}\ .\ \langle @npp\rangle\ .\ [\#npp]p\ .\ [pv]p,\\ & \qquad otherwise \rightarrow [\#pp]p\ .\ [pv]p\ .\ [\#pp]\ .\ [v]\ .\ \text{push\_back},\\ &)\end{aligned}$$

Notice the use of the conditional cases syntax from section 5.2.2 to check whether the pointer is valid. We use this to express our term more succinctly. It works by looping (with recursive calls) to the final element (signified by a $\#null$) and inserting a new element with LinkedList. We can define many functions to work with our linked list, in particular, a higher-order traverse function is useful for applying a given function f to each element of the list. This works in a similar way to push_back.

$$\begin{aligned}\text{traverse} \triangleq\ & \langle f\rangle\ .\ \langle @p\rangle\ .\ p\langle pv\rangle\ .\ p\langle @pp\rangle\ .\ [\#pp]p\ .\ [pv]p\ .\ [\#pp]\ .\ (\\ & \qquad null \rightarrow [pv]\ .\ f,\\ & \qquad otherwise \rightarrow [pv]\ .\ f\ .\ [\#pp]\ .\ [f]\ .\ \text{traverse},\\ &)\end{aligned}$$

With this, we can write a program (demonstrated in listing A.13) that constructs a linked list containing values 1 through 5, and then traverses them with a print function to print them.

$$\begin{aligned}&[1]\ .\ \text{LinkedList}\ .\ \langle @p\rangle\ .\\ &[\#p]\ .\ [2]\ .\ \text{push\_back}\ .\\ &[\#p]\ .\ [3]\ .\ \text{push\_back}\ .\\ &[\#p]\ .\ [4]\ .\ \text{push\_back}\ .\\ &[\#p]\ .\ [5]\ .\ \text{push\_back}\ .\\ &[\#p]\ .\ [\langle x\rangle\ .\ [x]out]\ .\ \text{traverse}\end{aligned}$$

We can easily use the pairs in section 6.1.2 with our linked-list, to create a list of pairs, because our function definitions are generic - see listing A.16 !. Hence with this, we can create some very powerful programs.

Purely Functional Approach

To see how this approach differs from a standard purely functional approach, we compare it to a list built using Church encodings. Here, lists are represented as higher-order functions, and rather than a data structure (like our linked-list approach), we have a function for processing the head of the list and a function for processing the tail of the list - see listing A.20 for an implementation of a purely functional list in our calculus.

The primary advantage of the purely functional approach is that it doesn't require a state, and thus, it's safer and less prone to failure. Additionally, it can be more efficient due to naturally encoding lazy-evaluation (demonstrated in listing A.21), i.e. we don't need to evaluate

terms in a list until they are actually needed. This is crucial in being able to work with infinite lists because it enables us to work with them without having to represent all of them.

We can replicate this in our linked-list by turning elements into *lazy-functions* that store an *evaluator* (that calculates or fetches, etc.) of an element rather than the element itself. For example, to represent a list of square numbers, we could use the higher-order function $[n].[n].\mathit{square}$ which gives the square of n upon evaluation. Additionally, our linked-list is much better in regard to memory complexity because it doesn't *store* the extra auxiliary functions required to construct a higher-order list function.

6.1.4 Call-by-Reference

Our first-class location semantics enable us to model call-by-reference semantics. This works by passing location values (as parameters) to functions which can read/write the information they point to (via the semantics in section 6.1.1). This allows us to pass (on the regular λ-stack) a location value that points to some term as opposed to the entire term itself.

Listing 6.5: Defining a call-by-reference doubling function in C.

```
void double_it(int *p)
{
    int s = (*p) + (*p);
    *p = s;
}

int a = 5;
double_it(&a); // Pass a 'by-reference'
```

In listing 6.5 we define a function called double_it which doubles the value pointed to by p. Notice how the function has return type void: this can be interpreted as specifying that it doesn't put anything back onto the stack. Given the pointer p, we can dereference it and read/write the data it points to (an integer in this case) by using the * operator (we discuss this in section 4.2).

We can form a direct analogue by creating the function double_it, which accepts a given location value parameter that points to the (first) stack element to be doubled.

$$\text{double\_it} \triangleq \langle @p \rangle \,.\, p\langle x \rangle \,.\, [x + x]p$$

This works by popping the contents from the given location p in variable x and writing its sum $x + x$ back. Similarly to the variables example in section 6.1.1, we write $x + x$ inside the application for succinctness - in reality, it would be a stack operation. Notice how we don't have an application to the regular λ-stack at the end of the term: this mirrors void in the function declaration in the C implementation.

With this, we can write a program (demonstrated in listing A.18) that doubles the contents of a variable a.

$$\mathit{new}\langle @a \rangle \,.\, [5]a \,.$$
$$[\#a] \,.\, \text{double\_it}$$

In performance-critical applications, we often desire to reduce the number of *copies* (as they can get expensive) we make. We identify *shallow copies* as ones which only duplicate the pointer to the object; we identify *deep copies* as ones which duplicate the underlying data of the object.

It is worth noting that there is no such thing as *copying* a term since they themselves just encode information (about a function or value etc.), and thus, copying them is redundant. A true *deep copy* would refer to the duplication of a location stack: we would create a new location and copy the terms from the first into it (as seen in section 6.1.1).

6.1.5 Pointer Arithmetic

We discuss a basic parallel of pointer arithmetic whereby we can perform arithmetic on location values.

Listing 6.6: Pointer arithmetic to access arrays in C.

```
int arr[5] = {1, 2, 3, 4, 5};
int second = *(arr + 1);
int fourth = *(arr + 3);
```

In listing 6.6, we define an array of integers that has the values $1, 2, 3, 4$ and 5. The array is really just a pointer to the first element, and thus, we can perform pointer arithmetic to access elements. We do this by adding an offset to the start index which results in the index of our desired element. Each unit added in the binary operation $+$ corresponds to stepping the number of bytes given by the size of the type (integer in this case which is 4-bytes). This only works because we assume all values have a defined size and are stored sequentially in memory.

If we make similar assumptions in our calculus, we could perform something similar. Assume stacks have fixed sizes (determined by term types) and have a sequential well-defined order in some shared memory. Define an arithmetic binary operation $+$ on our location values which offsets it into the sequential stack memory by the amount given by the operand (depending on term type sizes). If the location *arr* gives the first stack in this sequential memory, then we can write an analogue to listing 6.6 in

$$\begin{aligned}&\langle @arr \rangle \,.\\ &[\#arr] \,.\, [1] \,.\, + \,.\, \langle @second \rangle \,.\\ &[\#arr] \,.\, [3] \,.\, + \,.\, \langle @fourth \rangle\end{aligned}$$

whereby the dereference operator * is paralleled in our location abstractions for location variables *second* and *fourth*. Note, the values in the stacks do not have to be integers; they can be any term as long as each sequential stack has the same fixed size. Thus, we could store anything here, including functions, which makes this very powerful.

This assumes we have a sophisticated type system which can infer the size of terms etc. However given this, we can encode a very important feature of conventional pointers, and thus, we can create arrays and manipulate heap memory efficiently.

6.2 Location Parameters

External definitions from Heijltjes (2023) that use fixed locations can be written as internal higher-order functions where the location is passed as a parameter. The definitions of set a and get a for updating a memory cell with location a can be transformed with

$$\begin{array}{rcl@{\qquad}rcl} \mathsf{get}\ a & \triangleq & a\langle x\rangle.\,[x]a.\,[x] & \longrightarrow & \mathsf{get} & \triangleq & \langle @a\rangle.\,a\langle x\rangle.\,[x]a.\,[x] \\ \mathsf{set}\ a & \triangleq & \langle x\rangle.\,a\langle \_\rangle.\,[x]a & \longrightarrow & \mathsf{set} & \triangleq & \langle @a\rangle.\,\langle x\rangle.\,a\langle \_\rangle.\,[x]a \end{array}$$

Furthermore, the definitions of print and read for IO can be transformed with

$$\begin{array}{rcl@{\qquad}rcl} \mathsf{read} & \triangleq & \mathsf{in}\langle x\rangle.\,[x] & \longrightarrow & \mathsf{read} & \triangleq & \langle @a\rangle.\,a\langle x\rangle.\,[x] \\ \mathsf{print} & \triangleq & \langle x\rangle.\,[x]\mathsf{out} & \longrightarrow & \mathsf{write} & \triangleq & \langle @a\rangle.\,\langle x\rangle.\,[x]a \end{array}$$

The name *write* is inspired by *C-Style file input/output* (n.d.), which provides functions *fwrite* and *fread* for the same purpose. Notice that, unlike set, the function write doesn't pop from a before writing. Similarly, the function read doesn't push back to a after reading. This is because we assume they are writing to a *stream* location (like IO). With this, we can define

$$\begin{array}{rcl} \mathsf{input} & \triangleq & [\#\mathsf{in}].\,\mathsf{read} \\ \mathsf{print} & \triangleq & [\#\mathsf{out}].\,\mathsf{write} \end{array}$$

Notice the use of these generic definitions in the programs of listings A.5, A.8, A.10, A.13 and A.16.

It is worth mentioning that we can take this further to create *get a n* as a function which: pops n terms from a, restores them to a, and then puts the n^{th} one on the regular λ-stack. These can be used to define struct accessors (see section 6.1.2) for structs with n elements.

Example 6.1

The pair accessor functions *fst* and *snd*, from section 6.1.2, can be written as *get a* 1 and *get a* 2, respectively.

6.3 Explicit Term Mobility

First-class locations open up the possibility of being much more explicit with how terms are used and moved around the program. Locations can be passed as arguments to functions, returned as results, stored in data structures, and manipulated like any other data type. And thus, we are no longer restricted to just copying a term, in particular, we have the explicit choice of copying data or sharing it (see sections 6.1.1 and 6.1.4). Because of this, we introduce challenges: we need careful management to ensure correctness and security. However, we hope a rich type system would go far in helping approach these challenges.

Functions can mutate location stacks without returning (pushing) to the regular λ-stack. This effectively gives them a type signature of void, which is to say, they are procedures that modify some given state. This was possible before by fixing global state locations ahead of time, although, it's more dynamic here because states can be created and shared at run-time. For example, we could decide at run-time whether to write to a file or the standard output by simply sharing a different location value.

6.4 Proposed Syntax For Functions

We often find ourselves writing applications to push parameters, then calling a function, and finally writing abstractions to pop function parameters. We propose a simple syntax, inspired by *Haskell Programming Language* (n.d.), which reduces the verbosity of this common occurrence. This can be seen in example 6.2.

Example 6.2

Consider a function find_roots which takes some pointer (location value) to a struct representing a quadratic equation, finds its two[a] roots, and returns them by pushing each to the regular λ-stack. The conversion from our basic syntax would look like

$$\cdots \,.\, [\#q] \,.\, \text{find\_roots} \,.\, \langle x1 \rangle \,.\, \langle x2 \rangle \,.\, \cdots$$
$$\downarrow$$
$$\cdots \,.\, \text{find\_roots}\ \#q \ \rightarrow\ x1, x2 \,.\, \cdots$$

We pass parameters as Haskell does, i.e. by listing them to the right of the function name. We bind the results by listing them to the right of an arrow that comes after the parameters.

[a] Assume it always has two roots.

Interestingly, this looks similar to do-notation in Haskell, which lists bound variables in a similar way. This comparison hints at the imperative nature of our calculus because do-notation is designed to provide imperative semantics in Haskell.

Note that because we can have void functions (see section 6.3), we may have instances where nothing is returned. Additionally, functions which perform an operation on a global state may have no parameters.

Chapter 7

Conclusion

Our extensions have successfully enabled the implementation of many useful features. In particular, mutable store (see section 6.1.2), which is a fundamental basis for building abstractions in computer science, and thus, it is important to encode in the FMC. We have successfully implemented a recursive data structure (see section 6.1.3) using semantics enabled by first-class locations, which successfully provides an alternative approach to the purely functional encoding (see section 6.1.3).

The semantics of explicit mobility (see section 6.3) introduce the ability to be much more dynamic and assertive with how our programs handle usages of terms. We can copy them, or we can share them, and thus, the calculus becomes much more aligned with conventional imperative state-based programming.

A very interesting idea, which is novel to the FMC with first-class locations, is pointer-arithmetic (see section 6.1.5). It is not clear whether this would be practical due to the strict assumptions it requires, however, its implications are clearly useful - be that for manipulating sequences of sequential stacks of terms (as an array) or traversing our memory.

On the other hand, the FMC provides an excellent functional foundation and its results carry forward to provide expressibility and higher-order functions, etc. From a subjective view, the FMC with first-class locations feels like a functionally-based C - the inspiration for the name of our interpreter, *cFMC*.

For these reasons, our calculus becomes much more applicable to general programming problems. However, practicability is affected due to the dangerous nature of pointers. Without a solid type system, it appears to suffer from the same issues as other pointer-enabled imperative languages. Although despite this, we have demonstrated the usefulness of first-class locations.

The translation of our design into a working interpreter serves as a concrete validation of its effectiveness. The numerous examples showcased in section A provide practical evidence of the value our extensions have added. We have been able to develop intricate programs with minimal deviation from the pure FMC's grammar, syntax, and machine, highlighting the practical utility of our approach.

To summarise, our first-class locations provide many more avenues to use *imperative-style* ideas (see section 6.1), which can assist the programmer:

- Performance: functional-style structures are immutable, which can lead to poor performance in certain cases, especially when dealing with large data sets. For example, adding or removing elements from a large list (see section 6.1.3) can be expensive.
- Memory usage: functional-style structures require many auxiliary functions to be encoded, which can be a concern in memory-constrained environments or when dealing with very large lists.
- Complexity: functional-style structures are often laborious to reason about, requiring a deeper understanding of concepts, which makes them harder to write and debug.

This is especially apparent for programmers who are more familiar with imperative programming paradigms, which is the majority. By merging these paradigms, we present the programmer with more opportunities, but it is important to balance them appropriately.

7.1 Future Work

Validating Existing Properties

Due to the scope of our project, we haven't been able to formally check that properties from Heijltjes (2023) and Barrett, Heijltjes and McCusker (2023) still hold, i.e. confluence etc.

Freeing Locations

Currently, there is no mechanism for the machine to know when a location stack is no longer being used and thus that it can be freed - something similar to the Rust borrow checker could be useful here (Pearce, 2021).

Foreign Code

With first-class locations, we could do intrinsically pointer-based tasks: accessing foreign code (shared code written in another language), manipulating sections of heap memory (frame buffers, file contents, streamed network data), and writing memory-efficient programs.

Cross Compiler

Since our calculus shares many similarities with C, a project which implements a cross compiler for C code to FMC code, and vice versa, would be interesting. Such a project might explore how we can identify optimisations and errors in the C code by applying the theory of the FMC.

7.1.1 Extending The Interpreter

Here we give some future ideas for the interpreter: they are out of the scope of this project, and thus, we did not attempt to implement them.

- **Performing reduction**
  - The program would check for and perform reductions before running the machine.

- **Type checking**
  - The program would type check terms.

It would also be useful to develop some kind of "standard library" of common functions that can be used in a program. This would work with an import system (or simply include copy-and-paste macros like C or C++).

Word count = **9934**

Bibliography

Barrett, C., Heijltjes, W. and McCusker, G., 2023. The Functional Machine Calculus II: Semantics. In: B. Klin and E. Pimentel, eds. *31st eacsl annual conference on computer science logic (csl 2023)*. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, *Leibniz international proceedings in informatics (lipics)*, vol. 252, pp.10:1–10:18.

C language pointer [Online], n.d. C++ reference. Available from: https://en.cppreference.com/w/c/language/pointer [Accessed 17 Nov 2022].

C language struct [Online], n.d. C++ reference. Available from: https://en.cppreference.com/w/c/language/struct [Accessed 02 Mar 2023].

C language type [Online], n.d. C++ reference. Available from: https://en.cppreference.com/w/c/language/type [Accessed 17 Nov 2022].

C++ language variant [Online], n.d. C++ reference. Available from: https://en.cppreference.com/w/cpp/utility/variant [Accessed 07 Apr 2023].

C-style file input/output [Online], n.d. C++ reference. Available from: https://en.cppreference.com/w/cpp/io/c [Accessed 04 Feb 2022].

Church, A., 1936. An unsolvable problem of elementary number theory. *American journal of mathematics*, 58(2), pp.345–363.

Church, A. and Rosser, J.B., 1936. Some properties of conversion. *Transactions of the american mathematical society*, 39(3), pp.472–482.

Gifford, D.K. and Lucassen, J.M., 1986. Integrating functional and imperative programming. *Proceedings of the 1986 acm conference on lisp and functional programming, lfp 1986*. pp.28–38.

Haskell programming language [Online], n.d. Haskell. Available from: https://www.haskell.org/ [Accessed 03 Feb 2023].

Heijltjes, W., 2023. The Functional Machine Calculus. *Electronic Notes in Theoretical Informatics and Computer Science*, Volume 1 - Proceedings of MFPS XXXVIII, February.

Ingerman, P., 1961. Thunks: A way of compiling procedure statements with some comments on procedure declarations. *Communications of the acm*, 4(1), pp.55–58.

Kang, J., Hur, C.K., Mansky, W., Garbuzov, D., Zdancewic, S. and Vafeiadis, V., 2015. A formal c memory model supporting integer-pointer casts. *Sigplan notices*, 50(6), pp.326–335.

Moggi, E., 1989. Computational lambda-calculus and monads. *Fourth annual symposium on logic in computer science*. pp.14–23.

Ocaml pointers [Online], n.d. OCaml. Available from: https://ocaml.org/docs/pointers [Accessed 03 Feb 2023].

Ocaml programming language [Online], n.d. OCaml. Available from: https://ocaml.org/ [Accessed 03 Feb 2023].

Pearce, D.J., 2021. A lightweight formalism for reference lifetimes and borrowing in rust. *Acm trans. program. lang. syst.*, 43(1), apr.

Peyton Jones, S.L. and Wadler, P., 1993. Imperative functional programming. *Conference record of the annual acm symposium on principles of programming languages*. pp.71–84.

Rust programming language [Online], n.d. Rust. Available from: https://www.rust-lang.org/ [Accessed 03 Feb 2023].

Appendix A

Interpreter Demonstration

In this chapter, we provide a series of both simple and complex practical implementations and runs of programs. We also show how the interpreter deals with various errors such as parse errors and machine errors.

A.1 Basics

Listing A.1: Program which duplicates 0 on the stack.

```
main = ([0] . <x> . [x] . [x])
```

Listing A.2: Stack after running listing A.1.

```
——— Stacks ———
  —— (Reserved) Location lambda
     0
     0
————————————————————
```

A.2 Function Definitions

Listing A.3: Program which defines and applies a function *swap* that swaps (twists) the previous two terms on the stack.

```
swap = (<x> . <y> . [x] . [y])
main = ([0] . [1] . swap)
```

Listing A.4: Stack after running listing A.3.

```
——— Stacks ———
  —— (Reserved) Location lambda
     0
     1
————————————————
```

A.3 Reader & Writer Effects

Listing A.5: Program which takes two values as input, and then prints their sum.

```
write = (<@a> . <x> . [x]a)
print = ([#out] . write)

read = (<@a> . a<x> . [x])
input = ([#in] . read)

main = (
    input . input . + . print
)
```

A.4 Recursion

Listing A.6: Program which prints the Fibonacci sequence by recursing in the function *fib_aux*.

```
write = (<@a> . <x> . [x]a)
print = ([#out] . write)

fib_aux = (
    <b> . <a> . [a] . [b] . [a] . [b] . + . [a] . print . fib_aux
)
fib = (
    [0] . [1] . fib_aux
)

main = (
    fib
)
```

Listing A.7: Output after running listing A.6.

```
0
1
1
2
3
5
8
13
21
34
55
89
144
233
...
```

A.5 First-Class Locations

Variables

Listing A.8: Program (implementing the program in section 6.1.1) which creates and uses variables *a*, *b* and *c*.

```
get = (<@a> . a<x> . [x]a . [x])
set = (<@a> . <x> . a<_> . [x]a)

write = (<@a> . <x> . [x]a)
print = ([#out] . write)

main = (
    new<@a> . [5]a .
    new<@b> . [7]b .
    new<@c> . [#a] . get . [#b] . get . + . <z> . [z]c .
    [#a] . get . print .
    [#b] . get . print .
    [#c] . get . print
)
```

Listing A.9: Output after running listing A.8.

```
5
7
12
```

Structs

Listing A.10: Program (implementing the program in section 6.1.2) which constructs a pair of two elements and prints each element.

```
write = (<@a> . <x> . [x]a)
print = ([#out] . write)

Pair  = (<y> . <x> . new<@p> . [y]p . [x]p . [#p])
fst   = (<@p> . p<x> . [x]p . [x])
snd   = (<@p> . p<x> . p<y> . [y]p . [x]p . [y])

main = (
    [5] . [2] . Pair . <@p> .
    [#p] . snd . print .
    [#p] . fst . print
)
```

Listing A.11: Output after running listing A.10.

```
2
5
```

Listing A.12: Stack after running listing A.10: location loc_xxxxx comes from *new* in function *Pair*, which contains the two values 2 and 5 as elements of the pair.

```
——— Stacks ———
  —— Location loc_xxxxx
     5
     2

  —— (Reserved) Location lambda
————————————————
```

Recursive Data Structures

Listing A.13: Program (implementing the program in section 6.1.3) which constructs a linked list of values 1 through 5, and then traverses to print them (in order and in reverse).

```
write = (<@a> . <x> . [x]a)
print = ([#out] . write)

LinkedList = (
    <v> . new<@p> . [#null]p . [v]p . [#p]
)
push_back = (
    <v> . <@p> . p<pv> . p<@pp> . [#pp] . (
        null      -> [v] . LinkedList . <@npp> . [#npp]p . [pv]p,
        otherwise -> [#pp]p . [pv]p . [#pp] . [v] . push_back
    )
)
traverse = (
    <f> . <@p> . p<pv> . p<@pp> . [#pp]p . [pv]p . [#pp] . (
        null      -> [pv] . f,
        otherwise -> [pv] . f . [#pp] . [f] . traverse
    )
)
rtraverse = (
    <f> . <@p> . p<pv> . p<@pp> . [#pp]p . [pv]p . [#pp] . (
        null      -> [pv] . f,
        otherwise -> [pv] . [#pp] . [f] . rtraverse . f
    )
)

main = (
    [1] . LinkedList . <@p>
    . [#p] . [2] . push_back
    . [#p] . [3] . push_back
    . [#p] . [4] . push_back
    . [#p] . [5] . push_back
    . [#p] . [print] . traverse
    . [#p] . [print] . rtraverse
)
```

Listing A.14: Output after running listing A.13.

```
1
2
3
4
```

```
5
5
4
3
2
1
```

Listing A.15: Stack after running listing A.13.

```
——— Stacks ———
  —— Location loc_xxxxx
     1
     #loc_yxxxx

  —— (Reserved) Location lambda

  —— Location loc_yxxxx
     2
     #loc_zxxxx

  —— Location loc_zxxxx
     3
     #loc_wxxxx

  —— Location loc_wxxxx
     4
     #loc_vxxxx

  —— Location loc_vxxxx
     5
     #null
————————————————————
```

Listing A.16: Program (implementing the program in section 6.1.3) which constructs a linked list of pairs, and then traverses to print the first and second pair elements.

```
write = (<@a> . <x> . [x]a)
print = ([#out] . write)

LinkedList = (
    <v> . new<@p> . [#null]p . [v]p . [#p]
)
push_back = (
    <v> . <@p> . p<pv> . p<@pp> . [#pp] . (
        null      -> [v] . LinkedList . <@npp> . [#npp]p . [pv]p,
```

```
        otherwise -> [#pp]p . [pv]p . [#pp] . [v] . push_back
    )
)
traverse = (
    <f> . <@p> . p<pv> . p<@pp> . [#pp]p . [pv]p . [#pp] . (
        null      -> [pv] . f,
        otherwise -> [pv] . f . [#pp] . [f] . traverse
    )
)
rtraverse = (
    <f> . <@p> . p<pv> . p<@pp> . [#pp]p . [pv]p . [#pp] . (
        null      -> [pv] . f,
        otherwise -> [pv] . [#pp] . [f] . rtraverse . f
    )
)

Pair  = (<y> . <x> . new<@p> . [y]p . [x]p . [#p])
fst   = (<@p> . p<x> . [x]p . [x])
snd   = (<@p> . p<x> . p<y> . [y]p . [x]p . [y])

main = (
    [3]  . [4]  . Pair . LinkedList . <@p>
    . [#p] . [5]  . [12] . Pair . push_back
    . [#p] . [8]  . [15] . Pair . push_back
    . [#p] . [7]  . [24] . Pair . push_back
    . [#p] . [20] . [21] . Pair . push_back
    . [#p] . [fst . print] . traverse
    . [#p] . [snd . print] . traverse
)
```

Listing A.17: Output after running listing A.16.

```
3
5
8
7
20
4
12
15
24
21
```

Call-by-Reference

Listing A.18: Program (implementing the program in section 6.1.4) which passes variable *a* by reference to *double_it* (to double it).

```
get = (<@a> . a<x> . [x]a . [x])
set = (<@a> . <x> . a<_> . [x]a)

write = (<@a> . <x> . [x]a)
print = ([#out] . write)

double_it = (
    <@p> . [#p] . get . <x> . [x] . [x] . + . [#p] . set
)

main = (
    new<@a> . [5]a .
    [#a] . get . print .
    [#a] . double_it .
    [#a] . get . print
)
```

Listing A.19: Output after running listing A.18.

```
5
10
```

A.6 Purely Functional List

Listing A.20: Program (implementing the program in section 6.1.3) which constructs a list using *cons* via higher-order Church encodings. With this, it performs *map* on the list which squares each element. We can use *tail* and *head* to get the tail of the list and the head of the list, respectively.

```
nil =  (<_> . <x> . [x])
cons = (<h> . <t> . <f> . <x> . [t] . [h] . f)
head = (<l> . [nil] . [<h> . <_> . h] . l)
tail = (<l> . [nil] . [<_> . <t> . t] . l)
map = (<f> . <l> . [nil] . [<h> . <t> . [[[t] . [f] . map] . [[h]
    . f] . cons]] . l)

main = (
    [[[[nil] . [3] . cons] . [2] . cons] . [1] . cons] . [square]
```

```
        . map
)
```

Listing A.21: Stack after running listing A.20. Notice how lazy-evaluation means that the result of map (squaring) only applies to terms that we end up evaluating !

```
——— Stacks ———
  —— (Reserved) Location lambda
     [[[[nil] . [3] . cons] . [2] . cons] . [square] . map] . [[1]
        . square] . cons
```

A.7 Interpreter Errors

Listing A.22: An interpreter parse error when running *./cfmc –debug –source 'swap = (<x> . <y> . [x] . [y]) main = ([0 . [1] . swap)'*.

```
[Parse Error]
| swap = (<x> . <y> . [x] . [y])
| main = ([0 . [1
|            ^ Expected closing ']' of application
|            | Got '.'
```

Listing A.23: An interpreter machine error when running *./cfmc –debug –source 'swap = (<x> . <y> . [x] . [y]) main = ([0] . [1] . sap)'*.

```
[Machine Error] Variable 'sap' is not bound to anything !
| ——— Call Stack ———
| > main => [0] . [1] . sap
| ————————————
| ——— Stacks ———
|   —— (Reserved) Location lambda
|      1
|      0
| ————————————————————
```

Listing A.24: An interpreter machine error when running *./cfmc –debug –source 'swap = (<x> . <y> . [x] . [y]) main = (swap)'*.

```
[Machine Error] Abstraction is attempting to bind from empty
   stack 'lambda' !
| ——— Call Stack ———
| > swap => <x> . <y> . [x] . [y]
|  > main => swap
| ————————————————
| ——— Stacks ———
|   —— (Reserved) Location lambda
| ————————————————————
```

Listing A.25: An interpreter machine error when running *./cfmc –source 'main = ([0]in)'*.

```
[Machine Error] Application is attemping to push to input
   location !
| ——— Call Stack ———
| > main => [0]in
| ————————————————
| ——— Stacks ———
| ————————————————————
```

Appendix B

Interpreter Implementation

B.1 Implementation

The interpreter is implemented using modern C++, and thus, it is fast, safe and efficient. Because of this, it can run on any system which has a compiler, which is basically anything.

B.1.1 Parser

Before parsing, we apply a lexing stage to convert our input stream of characters into a stream of tokens (see listing B.1).

To make sense of the token stream, we apply our parser to construct a program. We store our program as a map from identifiers to terms. We use a generic term type (see listing B.2) to combine our constructors into one by making use of the variant type (*C++ Language variant*, n.d.). This forms a tree-like structure whereby each inductive constructor (application, abstraction, etc.) points to some generic term elsewhere (see listing B.3). The variant enables us to determine the kind of term we are looking at (variable, abstraction, etc.) at run-time and extract it. The parser builds an abstract syntax tree by recursively identifying term constructors and combining them together.

To be specific, we opt to use a top-down parsing approach, namely, we assume our input is in the form of a program and recursively descend (see listing B.4) by applying our grammar rules (see section 4.3.1).

B.1.2 Machine

Our machine uses the reduction rules given in section 4.3.4, although, to avoid substitution with α-conversion (for capture-avoidance) and for speed and readability reasons, we opt to use a binding environment with closures. This means that for each term, we keep track of all bound variables, inside an environment, to prevent variable capture.

Our machine is implemented in one continuous loop which repeats until we reach the end of a term (nil or $\star$). For each iteration, we determine the type of the term (application, abstraction, etc.) and perform the reduction rule given in section 4.3.4. This works by interacting with stacks of terms and the binding environment. Once a rule is applied, we set the next term to be the body of the current term (the $. M$ in each constructor of definition 2.3.3). We use a

stack frame of terms to keep track of cases when we change which term is being executed, i.e. in variable substitution.

B.2 Code Snippets

Listing B.1: An enum containing tokens which are produced by the lexer

```
enum class Token
{
  Lb, Rb,       // ( )
  Lab, Rab,     // < >
  Lsb, Rsb,     // [ ]
  Asterisk,     // *
  Dot,          // .
  Equal,        // =
  Comma,        // ,
  Underscore,   // _
  At,           // @
  Hash,         // #
  Plus,         // +
  Minus,        // -

  Arrow,        // ->

  Primitive,    // Primitive
  Id,           // Identifier

  Eof           // End of line
};
```

Listing B.2: Type of generic terms

```
class Term
{
  ...

  std::variant<
    NilTerm, VarTerm, AbsTerm, AppTerm, LocAbsTerm, LocAppTerm,
    ValTerm, BinOpTerm, CasesTerm
  > m_Term;
}

...
```

```
using TermOwner_t = std::shared_ptr<Term>;
using TermHandle_t = std::shared_ptr<const Term>;
```

Listing B.3: Type of term constructor for application

```
class AppTerm
{
  ...

  Loc_t m_Loc;
  TermOwner_t m_Arg;
  TermOwner_t m_Body;
};
```

Listing B.4: Recursive descent of the parser

```
...

if (m_Lexer->isPeekToken(Token::Asterisk))
{
  m_Lexer->next();
  return Term(NilTerm());
}
else if (auto varOpt = parseVar())
{
  return Term(std::move(varOpt.value()));
}
else if (auto appOpt = parseApp())
{
  return Term(std::move(appOpt.value()));
}
else if (auto absOpt = parseAbs())
{
  return Term(std::move(absOpt.value()));
}
else if (auto locAppOpt = parseLocApp())
{
  return Term(std::move(locAppOpt.value()));
}
else if(auto locAbsOpt = parseLocAbs())
{
  return Term(std::move(locAbsOpt.value()));
}

...
```