CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Towards a Practical Execution Model for Functional Languages with Linear Types

Master's thesis in Computer science and engineering

Filip Nordmark

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2024

Master's thesis 2024

Towards a Practical Execution Model for Functional Languages with Linear Types

Filip Nordmark

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

Towards a Practical Execution Model for Functional Languages with Linear Types

Filip Nordmark

Supervisor: Jean-Philippe Bernardy, Computer Science and Engineering
Examiner: Alejandro Russo, Computer Science and Engineering

Master's Thesis 2024
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2024

Towards a Practical Execution Model for Functional Languages with Linear Types

Filip Nordmark
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Typed functional programming has an attractive logical foundation which ensures safety and modularity. System-level programming, on the other hand, offers fine-grained control over program execution. Is there a way to combine those two strengths?

In this thesis, we take a first step in compiling functional languages to a C-like paradigm. Specifically, we propose a functional compiler intermediate language designed to bridge the gap between high-level functional programming and low-level system programming. To retain the generality of functional programming while allowing for finer control, we adopt a modular design where higher-level features decompose into lower-level constructs within a shared language.

Our design is not solely driven by practical considerations but rather we achieve memory control by following the structure of linear logic. In fact, our language is heavily based on the proof structure of a sequent calculus presentation of linear logic. In leveraging this formal system, our language gain theoretical interest and many desirable properties while simultaneously enabling a more direct execution model.

To demonstrate the viability of our design, we compile our language into a virtual machine language. The virtual machine language, which is specifically design as a target for our intermediate language, has an assembly-like style that maps naturally onto existing computer architecture. This suggest the potential for compiling our language into actual machine code.

Given that this work represents an initial effort of compiling functional languages to a C-like paradigm, we do not address all aspects of language design. Our primary contribution focuses on the treatment of functions, with particular emphasis on their representation and calling conventions.

Keywords: Linear logic, Linear Types, sequent calculus, functional programming, virtual machine.

Acknowledgements

I give my sincerest thanks to my supervisor Jean-Philippe Bernardy, for proposing this thesis topic, his guidance throughout, and for spending many hours with me in fruitful dialog.

I also want to thank my family for always being there for me and supporting me in all things.

Filip Nordmark, Gothenburg, 2024-06-24

Contents

List of Figures

List of Tables

1

Introduction

Functional programming merits itself on properties such as *referential transparency*, *higher-order functions*, *algebraic datatypes* and *strong type systems* [1]. However, some of these properties come with an associated runtime cost, often linked to excessive memory usage and consequential clean-up. Common criticisms of functional programming therefore revolve around its lack of performance, but perhaps more critical is its absence of *predictable* performance. This situation can be contrasted with system-level programming where the programmer essentially has full awareness of the generated code due to the close correspondence between language constructs and machine code.

Despite the criticism, functional programs can achieve great performance due to extensive compile time efforts. Noteworthy optimisations that compilers employ are unboxing, inlining and fusion. The big caveat, however, is that whether these optimisations take place or not is *heuristically* decided. As such, the functional programmer have little control over the runtime behaviour of their programs which in turn makes functional languages unsuitable for system-level programming. An important question is thus if this is an inherent limitation to functional programming or if there exist a way to have increased runtime control and stronger guarantees of optimisations while maintaining the upsides that functional programming offers?

In this thesis we begin to answer this question by initiating the designing a compiler intermediate language for functional languages with a more operational reading which enables it to express low-level details. A novelty of our design is that this is achieved, not despite of but as a prime reason of, a strong logical foundation – à la Curry-Howard. Specifically, the language bases itself on linear logic. Linear logic, with its emphasis on resource management, offers a promising framework to address the lack-of-control concerns associated with functional programming. By incorporating linear logic principles into the design of a compiler intermediate language, we lay the framework to bridge the gap between high-level abstractions and low-level details, ultimately providing programmers with increased control and predictability over their functional programs. As a long term goal we wish to stretch the functional paradigm closer to that of system-level programming languages such as C.

1.1 Minding the gap

A daunting challenge when compiling high-level languages, exemplified by functional languages, is the large semantics gap between them and the operation of a conventional computer. The smaller this gap can be made, the easier it is to reason about and compile programs. As such, for languages where this gap is large, intermediate representations (IRs) are introduced as supporting bridges.

Historically, many functional IRs have been rooted in the lambda calculus [2], [3]. This choice may partially be motivated by the Curry-Howard correspondence, which establishes a deep connection between the typed lambda calculus and intuitionistic logic. Consequently, lambda calculus-based IRs benefit from a strong logical foundation that underpins many of its optimization strategies. However, while effective for certain optimizations, these IRs often fall short when it comes to optimizations closer to machine level, particularly in terms of memory management.

The notable disconcern of memory management among lambda calculus-based IRs is arguably reflected by a corresponding gap in their underpinning logic. To address this gap, we base our design of an intermediate language on a sequent-calculus presentation of linear logic.

Sequent calculus make hypotheses explicit. In terms of programming language, the scope of variables becomes explicit.

Linear logic imposes usage restrictions and guarantees on values which can reduce overhead otherwise necessary in an unrestricted context. It offers an interesting perspective on resource management as it dually emphasizes production and *consumption*.

By incorporating these principles into our intermediate language design, we aim to bridge the divide between high-level abstractions and low-level machine concerns.

Despite the amount of research that has already gone into linear types, most of this research, we feel, has been too high level and theoretical, see Related work in section 2.4. Abstract machines proposed in these studies do not give insight into how they can be implemented effectively on contemporary computer. In particular we feel that there is a discernible gap in the literature concerning the *practical* handling of low-level resources such as memory; despite resource modeling being the very thing which makes linear logic interesting. As such, a focal point of this thesis is to demonstrate that linear logic can be utilised for concrete memory management.

We believe the lack of applied use of linear logic in low-level code generation might in part be due to the inherent difficulty of generating sensible machine code. Therefore, as a proof of concept, we adopt a pragmatic approach by targeting a virtual machine rather than machine code directly. Crucially, this virtual machine should be sufficiently low-level as to witness our stated goal, but also general enough to simplify the compilation process.

1.2 Structure

We give a brief overview of the thesis' content. Background material is presented in chapter 2, which we largely divide into logic and implementation, see section 2.1 and section 2.3 respectively. The former mainly concerns logic and its connection to computation while the latter discusses programming paradigms and their realisation on contemporary computers. Moreover, this chapter also includes related work in section 2.4.

In chapter 3 we give a specification of our intermediate language, the Low Level Linear Language, a functional programming language based on a linear logic sequent calculus. The language will be developed in succeeding steps adding more details at each iteration. Further, in chapter 4, we specify the Linear Virtual Machine, intended as a machine-like target for the Low Level Linear Language.

Chapter 5 show our languages in action. As proof of concept, we have developed a real compiler from the Low Level Linear Language to the Linear Virtual Machine as well as a runtime for the latter. Utilising the compiler, we analyse concrete programs by studying its compiled code.

Lastly, in chapter 6 we discuss limitations, future work and give a conclusion of our work.

2

Background

This chapter presents the underlying background for our design which we divide into two sides of the same coin: logic and implementation. To begin, we discuss logic and its connection to computation. We then contrast theory with reality by discussing low-level details of modern computer and implementation techniques – emphasising the large disparity between theoretical models and actual computers. In between these sides, we devote a section to *continuations* which have both a logical and a implementational significance. To end this chapter we also give an overview of related work.

2.1 Logic and computation

Given that the reader may not be familiar with linear logic or sequent calculus in general, this section aims to introduce and motivate our affection towards them. Assuming a familiarity with functional programming, we will primarily illustrate the relevance of linear logic and sequent calculus through the Curry-Howard correspondence, which connects logic and computation. We will begin by discussing natural deduction and general proof systems to set the foundation for our exploration.

Proof systems provide a formal and precise means of reasoning by specifying how derivations may be carried out as a succession of small, irrefutable steps. For example, a proof system might state that to deduce A and B one has to deduce A and independently deduce B. In typical notation, we would write this judgment as:

$$\frac{A \qquad B}{A \wedge B}$$

This notation indicates that the deductions above the line lead to the conclusion below the line.

Proof systems later found their way into type theory, encapsulated by the slogan "propositions as types, proofs as programs", under the name of the Curry-Howard correspondence [4]. The Curry-Howard correspondence relates natural deduction with the lambda calculus and has been greatly influential in the design of functional programming languages. This correspondence states that each proof can be seen as a program, and each proposition can be seen as a type, and vice versa. However, natural deduction is not the only proof system applicable to this correspondence.

As we will showcase, the sequent calculus can also be given multiple computational interpretations similar to the lambda calculus.

Despite both natural deduction and sequent calculus being introduced by Gentzen in 1935 [5], it is mainly natural deduction that has seen widespread adoption in computer science. We note, however, that Gentzen's initial motivation for developing the sequent calculus was to prove the *cut elimination theorem*, which he was unable to prove for natural deduction. Cut elimination represents the process of simplifying proofs by removing intermediate steps, making it an essential concept for understanding the computational and normalizing aspects of logical systems.

2.1.1 Natural Deduction and the lambda calculus

Natural deduction is an intuitionistic proof system by Gentzen which aim to formalize the "natural" way of reasoning [5]. Consider a calculus with *and*, *or* and *implies*:

$$A, B ::= A \wedge B \mid A \vee B \mid A \implies B$$

A deduction will be in the form $\Gamma \vdash A$ where Γ is a set of assumptions, $A_1, A_2, ..., A_n$. Thus $A_1, A_2, ..., A_n \vdash B$ should read A_1 *and* A_2 *and...*A_n *implies* B. This notation, with explicit assumptions and a turnstile ($\vdash$), might appear unusual for natural deduction but is in practice useful. The system can be referred to as *sequent* natural deduction but for simplicity we will simply adhere to the name of natural deduction.

The simplest rule is the Axiom:

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

which simply stats that if we assume A we can deduce A.

Next, the *introduction* rules states how a proposition can be built up:

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \qquad \frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \implies B}$$

Lastly, *elimination* rules states what one can deduce from larger proposition.

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} \qquad \frac{\Gamma \vdash A \vee B \qquad \Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma \vdash C}$$

$$\frac{\Gamma \vdash A \implies B \qquad \Gamma \vdash A}{\Gamma \vdash B}$$

We note that, with the exception of $\implies$ introduction, all the action is on the right of the turnstile.

The lambda calculus with pairs and sums is given by the terms:

$$\begin{aligned} v, w ::= {} & x \mid (v, w) \mid InL\ v \mid InR\ w \mid \lambda x.v \\ & \mid fst\ v \mid snd\ v \mid case\ v\ of (InL\ x \to v_1 \mid InR\ y \to v_2) \mid v\ w \end{aligned}$$

$$\frac{}{\Gamma, x : A \vdash x : A} \qquad \frac{\Gamma \vdash v : A \qquad \Gamma \vdash w : B}{\Gamma \vdash (v, w) : A \wedge B} \qquad \frac{\Gamma \vdash v : A}{\Gamma \vdash InL\ v : A \vee B}$$

$$\frac{\Gamma \vdash w : B}{\Gamma \vdash InR\ w : A \vee B} \qquad \frac{\Gamma, x : A \vdash v : B}{\Gamma \vdash \lambda x.v : A \implies B} \qquad \frac{\Gamma \vdash v : A \wedge B}{\Gamma \vdash fst\ v : A} \qquad \frac{\Gamma \vdash v : A \wedge B}{\Gamma \vdash snd\ v : B}$$

$$\frac{\Gamma \vdash v : A \vee B \qquad \Gamma, x : A \vdash v_1 : C \qquad \Gamma, y : B \vdash v_2 : C}{\Gamma \vdash case\ v\ of (InL\ x \to v_1 \mid InR\ y \to v_2) : C}$$

$$\frac{\Gamma \vdash v : A \implies B \qquad \Gamma \vdash w : A}{\Gamma \vdash v\ w : B}$$

Figure 2.1: Lambda calculus typing judgments

where x, y acts as variables.

The lambda calculus is computational in the sense that it has a reduction relation, of the form $v \to v'$, which specifies how terms can reduce. As witness to this relation we have $fst\ (v, w) \to v$ and most famously $(\lambda x.v)\ w \to v[w/x]$.

Terms of the lambda calculus can be assign a *type*, $\Gamma \vdash v : A$, as seen in figure 2.1. As should be noted, all of the rules directly match the logical rules of natural deduction.

What is perhaps most interesting of the simply typed lambda calculus is the interplay between semantics and types in the form of *type soundness*. We highlight two theorems.

Theorem 1 (Perservation)**.** *if* $\vdash t : A$ *and* $t \to t'$ *then* $\vdash t' : A$

That is, reduction preserves typing.

Theorem 2 (Progress)**.** *if* $\vdash t : A$ *then t is either a value or there exists a t' such that* $t \to t'$

Where value denotes a term that can not be reduced further, a sort of canonical term of the type.

Taking these two theorems together, we see that a well formed term can either be reduced to a value or it will "reduce" indefinitely.

2.1.2 Sequent Calculus

In the words of Jean-Yves Girard – the creator of linear logic – "The sequent calculus, due to Gentzen, is the prettiest illustration of the symmetries of Logic" [6].

In a sequent calculus, all logical connectives are given *left* and *right* rules. Thus, in contrast with intuitionistic logic, derivations may happen on either side of the

turnstile. From a logical point of view, this allows us to directly reason about *falsity* as well as truth [7], [8].

The rules of the sequent calculus be grouped into the *identity* group containing identity and cut;

$$\frac{}{\Gamma, A \vdash \Delta, A} \qquad \frac{\Gamma \vdash \Delta, A \qquad \Gamma', A \vdash \Delta'}{\Gamma, \Gamma' \vdash \Delta, \Delta'}$$

the *structural* group containing, left and right, weakening and contraction;

$$\frac{\Gamma \vdash \Delta}{\Gamma, A \vdash \Delta} \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash A, \Delta} \qquad \frac{\Gamma, A, A \vdash \Delta}{\Gamma, A \vdash \Delta} \qquad \frac{\Gamma \vdash A, A\Delta}{\Gamma \vdash A, \Delta}$$

and the *logical* group containing left and right introduction rules for all connectives of our calculi, in our case $\wedge$, $\vee$, $\Longrightarrow$

$$\frac{\Gamma \vdash \Delta, A \qquad \Gamma \vdash \Delta, B}{\Gamma \vdash A \wedge B, \Delta} \qquad \frac{\Gamma, A \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta} \qquad \frac{\Gamma, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}$$

$$\frac{\Gamma \vdash \Delta, A}{\Gamma \vdash \Delta, A \vee B} \qquad \frac{\Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \vee B} \qquad \frac{\Gamma, A \vdash \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, A \vee B \vdash \Delta}$$

$$\frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \implies B, \Delta} \qquad \frac{\Gamma \vdash A, \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, A \implies B \vdash \Delta}$$

The style of proofs becomes immediately different from that of natural deduction, as the only way to elimination propositions is through the cut rule. We refer to the book Proofs and Types by Girard, Taylor, and Lafont [6] for the reader interested in more logical theory.

From a computational point of view, sequent calculus makes co-value or continuations an inherent part of the language [7]. The computational interpretation of the sequent calculus has not been as uniform as that of natural deduction and the lambda calculus. Instead, a multitude of languages have been conceptualised with varying syntax and semantics [8]–[12]. Common for the languages are the duality between values and co-values and a semantics driven by the cut between them. The semantics of theses languages are often close to that of abstract machines.

2.1.3 Linear Logic

Linear logic was first introduced by Jean-Yves Girard in 1987 [13]. Girard's linear logic differentiates itself in being a logic concerned by *resources*, meaning that judgments are not merely truth derivations but rather they describe recipes that transmute objects into other objects – like a recipe. A consequence of this resource

view is that propositions can not be discarded nor duplicated and as such they have to be used exactly once.

In terms of linear logics' proof system, this is realised by disallowing contraction and weakening on arbitrary propositions in the sequent calculus. This seemingly small change have drastic consequences for the logic. Consider the following two judgments for deducing $\wedge$:

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \qquad\qquad \frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma, \Delta \vdash A \wedge B}$$

With weakening these two forms are logically equivalent – either weakening before or after the introduction will yield the other.

$$\frac{\dfrac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma, \Gamma \vdash A \wedge B}}{\Gamma \vdash A \wedge B} \qquad\qquad \frac{\dfrac{\Gamma \vdash A}{\Gamma, \Delta \vdash A} \qquad \dfrac{\Delta \vdash B}{\Gamma, \Delta \vdash B}}{\Gamma, \Delta \vdash A \wedge B}$$

We cheat slightly by generalising weakening to contexts rather than single propositions, but their equivalence is immediate. In linear logic they are however distinct, with the left rule introducing the *additive* conjunction & and the right rule introducing the *multiplicative* conjunction $\otimes$. Similarly distinction can be made for $\vee$, separating it into the additive $\oplus$ and the multiplicative $\parr$.

The new connectives are given the following rules:

$$\frac{\Gamma \vdash \Delta, A \qquad \Gamma' \vdash \Delta', B}{\Gamma, \Gamma' \vdash \Delta, \Delta' A \otimes B} \qquad\qquad \frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \otimes B \vdash \Delta}$$

$$\frac{\Gamma \vdash \Delta, A, B}{\Gamma \vdash \Delta, A \parr B} \qquad\qquad \frac{\Gamma, A \vdash \Delta \qquad \Gamma', A \vdash \Delta'}{\Gamma, \Gamma' A \parr B \vdash \Delta, \Delta'}$$

$$\frac{\Gamma \vdash \Delta, A}{\Gamma \vdash A \oplus B} \qquad\qquad \frac{\Gamma \vdash \Delta, B}{\Gamma \vdash A \oplus B} \qquad\qquad \frac{\Gamma, A \vdash \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, A \oplus B \vdash \Delta}$$

$$\frac{\Gamma \vdash \Delta, A \qquad \Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \,\&\, B} \qquad\qquad \frac{\Gamma, A \,\&\, B \vdash \Delta}{\Gamma, A \vdash \Delta} \qquad\qquad \frac{\Gamma, A \,\&\, B \vdash \Delta}{\Gamma, B \vdash \Delta}$$

$$\frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \Delta, \neg A} \qquad\qquad \frac{\Gamma \vdash \Delta, B}{\Gamma, \neg B \vdash \Delta}$$

Of course, not allowing weakening and contraction at all results in a fairly weak logic. In linear logic weakening and contraction are re-introduce in a restricted form by the connectives ! and ?.

$$\frac{!\Gamma \vdash A, ?\Delta}{!\Gamma \vdash !A, ?\Delta} \qquad \frac{\Gamma, A \vdash \Delta}{\Gamma, !A \vdash \Delta} \qquad \frac{\Gamma \vdash A, \Delta}{\Gamma \vdash ?A, \Delta} \qquad \frac{!\Gamma, A \vdash ?\Delta}{!\Gamma, !A \vdash ?\Delta}$$

! allows retraction and weakening on the left and ? allows it on the right.

The benefits and applications of linear logic in a computational setting has been investigated since its introduction, see section 2.4 for a selection of works incorporating linearity. Prime motivations of linearity ranges from safe protocols [14], in-place updates [15] and guaranteed fusion [16].

2.1.3.1 Polarised Linear Logic

Polarisation arises in the study of proof structures. It classifies logical connectors into either *positive* or *negative* polarity. The notion was originally found in the context of logical programming, where it was found that left rules of positive types and right rules of negative types can always be reversed in proof searches [17]. Continued research on polarity have later been explored in the context of contraction/weakening [18] and call by name/call by value duality [10].

Concretely, as can be seen by the rules in section 2.1.3, $\otimes$ and $\oplus$ are positive whereas $⅋$ and & are negative.

2.2 Continuations

While perhaps not well known among programmers, continuations appear extensively in sequent calculus-based languages [8]–[11] and has been used with great success in functional IRs [3], [19]. As it will likewise serve an integral part of our compiler language, see chapter 3, we aim to give some intuition towards them from a programmers perspective.

Continuations can intuitively be explained as objects that capture "the rest of the computation". A continuation acts as a function that does not return a value, instead it only describes the consumption of a value. In other words, it encapsulates the remaining steps to be performed.

In imperative languages, the concept of a `return` statement can be viewed as a special case of a continuation. When a function returns, it transfers control back to the point where it was called. This return mechanism operates differently from normal function calls, as it does not push a new frame onto the stack but rather pops the current frame off. This mechanism ensures that the caller's context is restored, making the return functionally similar to invoking a continuation. We will discuss this view and how it relates to our language further in section 6.2.2.

While continuations are seldom used directly by the user, they have played a notable role in compiler intermediate languages. In Andrew Appel's book *Compiling with Continuations* [3], they are the centerpiece of his compilation scheme. They are used to streamline control flow and facilitates systematic optimization and code transformations. Similarly, in GHC, continuations are leveraged in the Spineless Tagless G-machine (STG) [19], [20], the intermediate representation used after GHC Core. The STG machine uses continuations to manage thunks, function application and case analysis, allowing for efficient implementation of Haskell's lazy evaluation

semantics. In both cases continuations fundamentally change how a language gets evaluated. We argue that it is reminiscent to the operational style associated with sequent calculus based languages.

2.3 Implementation considerations

The computational models introduced in the previous section all suffer from a critical limitation: they are non trivial to implement efficiently on contemporary computers. There exists a significant gap between their high-level semantics and the execution models of actual computers. Furthermore, a naive implementation is likely to miss a lot of performance opportunities due to neglecting the intricacies of modern hardware.

In this section we contrast functional languages with system-level languages, assuming the style of the latter is tailored to bring the most out of modern hardware architecture. Going along with this comparison, we highlight design considerations for our intermediate languages, see chapter 3, aiming to bridge the gap between functional and system-level languages.

2.3.1 C-like paradigm

C has for a long time been considered *the* system programming language. This is natural because C, and its relatives, are characterized by their low-level control over memory and direct code generation scheme. For a language implementer that seeks to maximize performance, it is thus interesting to consider the design choices that go in to generating code with such low overhead. Furthermore, we assume that most computer architectures optimise for the type of machine code that C and its relatives produce.

Putting our design in sharp contrast to typical functional languages, we aim to emulate certain characteristics of C-like programming languages, including stack based memory management, unboxed data and stack based argument passing. By aligning our language design with the principles of system-level programming, we hope to, in the long run, benefit from the extensive work put into contemporary hardware architectures. Our approach partially serves as an investigation on the extent to which modern computer architectures are optimized for C-like languages. At the same time, we note that our design is not yet mature enough for this evaluation, see section 6.2.

2.3.2 ML-like paradigm

Functional programming languages, exemplified by the ML and Haskell families, introduce a paradigm shift in programming, emphasizing immutability, higher-order functions, and expressive type systems. While valued for their elegance and abstraction, these languages often face criticisms regarding their runtime efficiency, especially the concern that the produced code may have vastly different characteristics based on heuristic choices during compilation. The inefficiencies associated with ML-like paradigms stem from factors such as heap-based memory management, reliance

on garbage collection, and challenges in optimizing function calls and closures. Our work represents a pioneering effort in addressing these inefficiencies by compiling functional languages into a more C-like execution model which is effectively driven by the linear type system, thereby potentially mitigating the performance drawbacks traditionally associated with functional programming.

By compiling functional programming languages to a C-like paradigm, we aim to combine the expressive power of functional programming with the level of control afforded by system-level programming. Our approach takes a first step towards joining these paradigms, offering insights into the design considerations and trade-offs involved in achieving efficient-code generation for functional programs.

2.3.3 Stack vs. Heap

When a program requires memory it can either use stack or heap based memory. Prioritizing stack memory over heap memory offers several advantages, including avoidance of memory fragmentation and improved cache locality. The act of acquiring memory is also vastly different between the two, as heap allocation, in the worst case, requires a system call while stack allocations corresponds to simple pointer arithmetic [1]. The overhead of a system call is vast; the operating system needs to interrupt execution, checks for available memory in its shared memory pool, update its bookkeeping and assign the memory to the requester. Nonetheless, heap-usage is in practice necessary for certain data structures and dynamic memory allocation, as stack based memory is inherently limited to a first-in-first-out allocation scheme.

In system-level programming, such as C, stack allocations are the default with heap allocations being optional. In contrast, functional programming languages use heap allocations as the default with stack allocations being essentially unavailable for the programmer. This choice is further reflected in the argument passing style of the two paradigms, with C generally passing values on the stack while functional languages passes heap-pointers. By the assumption that system-level programs are better optimized to the hardware, we will also adopt the stack based argument passing style for our design.

2.3.3.1 Garbage collection

Typically, languages that mainly use heap memory favour garbage collection over manual memory management. As such, all prominent functional programming languages are garbage collected. Garbage collection offers automatic memory management but presents drawbacks in performance predictability and memory footprint. In performance-critical scenarios, manual memory management techniques are therefore favored for their ability to deliver predictable performance and memory usage. We believe that the utilisation of linear types allows us to forgo a garbage collector while at the same time eliminating common pitfalls of manual memory management – such as read after free or double free of memory.

[1] Reality is more nuanced. Due to the high cost of a system call, systems will in various ways avoid them, for example by allocations more memory than needed which can then be given out according to its own bookkeeping.

2.4 Related work

The design of compiler intermediate language for functional languages as well as logic inspired languages has been extensively researched. In this section we present some of these works and aim to relate them to our goal of creating a functional language suitable for system-level programming. To begin, we give a brief rundown of linearity in language design

Despite the theoretical promises of linear logic, practical adoption has been limited. The perhaps most notable exception is that of Linear Haskell [14]. Linear Haskell extends GHC, the *de facto* Haskell compiler, with linear type checking. The extension enables developers to annotate functions as linear; in which its argument is enforced to be linear. However, Linear Haskell is purely a type checking extension, and thus all linearity annotations are lost in later stages of compilation. Consequently, there is no inherent increase in performance during runtime as optimisations are still only driven by heuristics rather by linearity.

Abstract machines for linear languages have been specified in the literature, notably by Lafont [21] and Abramsky [22]. However, despite these formal specifications, the practical challenges of appropriately implementing linear languages largely remain. As it stands, the resource modeling capabilities of linear logic does not seem to have been transferable to the concrete management of real computer resources. Several executable linear languages have despite this been developed. We showcase some noteworthy languages from the literature in the following list:

- *Lilac: a functional programming language based on linear logic* [23] served as an exploration of the usage of linear types from the user perspective. Notably it developed a type checking algorithm for linear types similar to that of Milner's W algorithm [24]. However, the focus of that language was not to implement a concrete execution model.
- In *Efficient Implementation of a Linear Logic Programming Language*, Hodas, Watkins, Tamura, *et al.* [25] implemented a compiler for the linear logic language *Lolli*. However, Lolli [26] is inherently a logic programming language, thus declarative rather than functional. Consequently, it is not well-suited for general purpose programming, which is the aim of our work.
- *Efficient Functional Programming using Linear Types: The Array Fragment* [16], a prior Chalmers master's thesis, developed a functional linear logic based language. The thesis had a strong emphasis on the performance aspects of linearity by utilising its guarantee to fuse effectively. However, whilst it did produce efficient code, it did so by transpiling their language into C. Thus, the question whether linearity actually can contribute in the creation of an efficient runtime remains. Furthermore, the language was not aimed for general code, rather it had a sole focus on the treatment of arrays.
- *A type system for bounded space and functional in-place update* [27] demonstrates how to transpile a simple linearly typed functional language into malloc()-free C code. The technique they showcase is however restricted to functions

with a first order type, that is, they do not support higher order functions.

The list covers linear languages at large. We continue with related works concerning compilation and intermediate representations for functional languages. In particular, we will look at both logic based IRs as well as more pragmatic works.

2.4.1 Sequent Calculus as a Compiler Intermediate Language

Downen, Maurer, Ariola, *et al.* [11] show that the sequent calculus can serve as a compiler intermediate language by implementing *Sequent Core*, a sequence calculus version of GHC Core. Sequent Core specializes itself as a compiler intermediate language by including let bindings and jumps; Two important features for producing space and time efficient code from functional languages. Whilst it is a concrete IR, it was not used for code generation – it was however integrated in GHC as a compiler plugin.

2.4.2 The logical abstract machine

Ohori [28] defines a logic calculus that closely resembles that of machine code. In showing that this calculus corresponds to that of natural deduction he achieves a compilation scheme from the lambda calculus to machine code.

2.4.3 One-shot continuations

Bruggeman, Waddell, and Dybvig [29] explored the performance benefits of *one-shot* continuations, that is, continuations that may only be called once. They concluded that this restriction allows for a more optimised representation which displays better performance, both in terms of speed and memory, than general continuations. Furthermore, they argue that the vast use cases of continuations are in fact one-shot continuations.

In the context of linearity, all continuations are one-shot continuations, unless they are captured by a (!). This indicates that linearity can be utilised to achieve simpler representations with less overhead compared to non-linear objects.

2.4.4 The STG Machine

In GHC's compilation process, Core gets transformed into a intermediate representation called STG [19]. STG is an abstract machine.

The state of the STG machine is represented by the triple (e, S, H), where e is the expression to be evaluated, S a stack of continuations and H is a map to heap objects [20]. A small step relation of the form $(e, S, H) \implies (e', S', H')$ is used to specify the operational semantics of the machine.

3

The Low Level Linear Language

This chapter presents the Low Level Linear Language (LLLL). Before going into details of the language we will shortly discuss its envisioned purpose and scope. We will also outline the structure of the upcoming chapter.

LLLL is intended as a functional language compiler intermediate representation, akin to that of GHC Core. As outlined in section 2.3, we feel like there is much potential in targeting a more C-like execution model; something that current functional IRs do not expose enough details for. Consequently, LLLL departs from the lambda calculus and its natural deduction basis, instead drawing inspiration from a linear logic sequent calculus. The justification of this choice is outlined in section 2.1, but briefly we base it on the following considerations: the calculus emphasises a duality between production consumption and has a well-behaved cut elimination property which serves as the foundation for optimization techniques such as inlining and fusion. By leveraging linear logic instead of intuitionistic, LLLL gains the potential for finer control over resources on a logical level.

A downside to increased control is usually the loss of generality. To counteract this, we propose a modular approach in which high- and low-level features coexists and where a programmer can opt-in to gain low-level control. A compiler is later responsible of decomposing the whole program into the lowest-level parts of the language. We emphasise that, while this step requires heuristics, a programmer could have written it at this level.

To guide the reader, we will present LLLL in three iterations, with each iteration making the language finer grained. In doing this we successively bridge the semantic gap between high-level functional languages and lower-level details. Crucially, at no point in this process do we lose the logical foundation of the language. The iterations can be summaries as follows: LLLL1 is presented as a abstract language without much focus on implementation concerns. LLLL2 focuses on data representations and various kinds of function representations. Lastly, LLLL3 deals with closure conversion by introducing primitive constructs such as explicit allocations and pointers. At all stages, the logical foundation is at the forefront, yet we are able to express a higher level of control than is usually possible in functional languages thanks to our logical foundation.

3.1 LLLL1: A logical language

The first iteration of LLLL is a computational model much in the same vain as the lambda calculus. At this stage we do not make any particular considerations to implementation, that will be the focus of later iterations.

While LLLL is a functional language, it does not have functions in the lambda calculus sense of $A \to B$, that is with an argument and a return value. Instead functions are expressed through continuations of the form $\neg A$ which only takes an argument and does not return a value, for a general introduction and discussion of continuations see section 2.2. As such, LLLL follows in the tradition of functional IRs in continuation passing style (CPS), and shares the common goal of making control flow – and in extension control structures – explicit. Furthermore, continuations are particularly suited in the framework of linear logic [29], [30].

The grammar of the language is defined in four parts: commands, c; values, a, b; environments, σ, ρ; and redexes, r.

$$c ::= () = x; c \mid (x, y) = z; c \mid \text{Case } z \text{ of}(x \to c_1 \mid y \to c_2) \mid z\ a$$

$$a, b ::= () \mid (a, b) \mid \text{Inj}_1\ a \mid \text{Inj}_2\ b \mid \lambda x.\ c \mid x \qquad\qquad r ::= \langle \sigma \mid c \rangle$$

x, y and z are variables. Environments are maps from unique variables to values that we will write as $\sigma ::= x_1 \mapsto a_1, x_2 \mapsto a_2, \ldots, x_n \mapsto a_n$.

To give some intuition over the language: Values simple construct data – much in the same way as would be expressed in an any functional language. Commands, on the other hand, describe how to *de*construct environments, with each command term describing the deconstruction of a single value in the environment. Both values and commands are always in normal form, meaning that they can not be simplified internally. Instead, reduction arises in the meeting of the two, namely in the redex. We will expand upon this computational notion in section 3.1.2, but first we will consider the typing of terms.

3.1.1 Types

The types of our language corresponds to a subset of Girard's linear logic, see section 2.1.3. In particular, the subset corresponds to the propositions:

$$A, B ::= 1 \mid A \otimes B \mid A \oplus B \mid \neg A$$

1, $\otimes$ and $\oplus$ have positive polarity whilst $\neg$ can be considered a shift in polarity; the meaning of polarity is described in section 2.1.3.1.

The other connectives from linear logic can be encoded as:

$$A \& B = \neg(\neg A \oplus \neg B) \qquad A \parr B = \neg(\neg A \otimes \neg B) \qquad A \multimap B = \neg(A \otimes \neg B)$$

which we morally justify through the use of double negation and De Morgan's laws from classical linear logic. Note however that negation ($\neg$) of our language is not an involution as in classical linear logic.

Commands $\boxed{\Gamma \vdash c}$

$$\frac{\Gamma \vdash c}{\Gamma, x : 1 \vdash () = x; c} \qquad \frac{\Gamma, x : A, y : B \vdash c}{\Gamma, z : A \otimes B \vdash (x, y) = z; c} \qquad \frac{\Gamma \vdash a : A}{z : \neg A, \Gamma \vdash z\ a}$$

$$\frac{\Gamma, x : A \vdash c_1 \qquad \Gamma, y : B \vdash c_2}{\Gamma, z : A \oplus B \vdash \text{Case } z \text{ of}(x \to c_1 \mid y \to c_2)}$$

Values $\boxed{\Gamma \vdash a : A}$

$$\frac{}{x : A \vdash x : A} \qquad \frac{}{\vdash () : 1} \qquad \frac{\Gamma \vdash a : A \qquad \Delta \vdash b : B}{\Gamma, \Delta \vdash (a, b) : A \otimes B} \qquad \frac{\Gamma \vdash a : A}{\Gamma \vdash \text{Inj}_1\ a : A \oplus B}$$

$$\frac{\Gamma \vdash b : B}{\Gamma \vdash \text{Inj}_2\ b : A \oplus B} \qquad \frac{\Gamma, x : A \vdash c}{\Gamma \vdash \lambda x.\ c : \neg A}$$

Environments $\boxed{\sigma : \Gamma \to \Delta}$ Redex $\boxed{r : (\Gamma \vdash)}$

$$\frac{\sigma : \Gamma \to \Delta \qquad \Phi \vdash a : A}{(\sigma, x \mapsto a) : \Gamma, \Phi \to \Delta, x : A} \qquad \frac{}{[] : \{\} \to \{\}} \qquad \frac{\sigma : \Gamma \to \Delta \qquad \Delta \vdash c}{\langle \sigma \mid c \rangle : (\Gamma \vdash)}$$

Figure 3.1: Typing rules for LLLL

The typing rules for the language are given in fig. 3.1. Commands and values rules take the form $\Gamma \vdash c$ and $\Gamma \vdash a : A$ respectively, thus they differ by either having a right side type or not. Values have a right side type to symbolise construction of a value. Commands do *not* have a right side type to symbolise consumption. In fact this is not merely symbolic, values have *right* rules which we will read in a top to bottom fashion to mean introduction while commands have *left* rules which we read in a bottom up fashion to mean consumption. Each rule directly matches their pure linear logic equivalence, disregarding the enforced size of the right sequent.

What does not match as closely is our interpretation of the Cut rule, namely the redex $\langle \sigma \mid c \rangle$. Here a cut occurs between a environment and a command rather than might be first expected; a value and a command. This change is primarily chosen to get the wanted computational behaviour of the language.

3.1.2 Semantics

The operational semantics of the language is presented in fig. 3.2. Redexes have a small step semantics of the form $r \longrightarrow r'$. Values have a big step semantics of the form $a[\sigma] \to t/\rho$. Intuitively, the value semantics says that a in context σ normalises to the value t with remaining context ρ. When $a[\sigma] \to t/\{\}$ we will allow ourselves to write $a[\sigma]$ as a shorthand for t.

The perhaps more interesting semantics is that of the redex. The redex is inspired

Redex $\boxed{r \longrightarrow r'}$

$$\begin{aligned}
\langle \sigma, x \mapsto () \mid () = x; c\rangle &\longrightarrow \langle \sigma \mid c\rangle \\
\langle \sigma, z \mapsto (a, b) \mid (x, y) = z; c\rangle &\longrightarrow \langle \sigma, x \mapsto a, y \mapsto b \mid c\rangle \\
\langle \sigma, z \mapsto \mathrm{Inj}_1\ a \mid \mathrm{Case}\ z\ \mathrm{of}(x \to c_1 \mid y \to c_2)\rangle &\longrightarrow \langle \sigma, x \mapsto a \mid c_1\rangle \\
\langle \sigma, z \mapsto \mathrm{Inj}_2\ b \mid \mathrm{Case}\ z\ \mathrm{of}(x \to c_1 \mid y \to c_2)\rangle &\longrightarrow \langle \sigma, y \mapsto b \mid c_2\rangle \\
\langle z \mapsto [\rho; \lambda x.\ c], \sigma \mid z\ a\rangle &\longrightarrow \langle \rho, x \mapsto a[\sigma] \mid c\rangle
\end{aligned}$$

Values $\boxed{a[\sigma] \to t/\rho}$

$$\frac{}{()[\sigma] \to ()/\sigma} \qquad \frac{a[\rho] \to t/\theta \qquad b[\sigma] \to u/\rho}{(a, b)[\sigma] \to (t, u)/\theta} \qquad \frac{a[\sigma] \to t/\theta}{(\mathrm{Inj}_i\ a)[\sigma] \to (\mathrm{Inj}_i\ t)/\theta}$$

$$\frac{}{(\lambda x.\ c)[\sigma] \to [fv(\lambda x.\ c); \lambda x.\ c]/(\sigma \setminus fv(\lambda x.\ c))} \qquad \frac{}{x[\sigma, x \mapsto t] \to t/\sigma}$$

Figure 3.2: Semantics of LLLL

by the Cut rule but extended to a form of *multi*-cut. As such, cuts here are considered between whole environments rather than singular propositions. The resulting semantic is closer in style with abstract machines, such as STG [20] or the linear abstract machine [21], than with rewrite systems, such as the lambda calculus.

An important point to make for both semantics is that they maintain linearity by never sharing. As such, resources, here in the form of values, are always managed explicitly by the language.

3.1.3 Examples of LLLL programs

It can be helpful to see how LLLL operates as a concrete language. As such, we present some example programs. To begin, the identity function over type A would have the type $\neg(A \otimes \neg A)$ and can be defined as the value

$$id = \lambda z.(x, k) = z; k\ x$$

The swap function would have type $\neg((A \otimes B) \otimes \neg(B \otimes A))$ and value

$$\begin{aligned}
swap = &\lambda z. \\
&(z', k) = z; \\
&(x, y) = z'; \\
&k\ (y, x)
\end{aligned}$$

As a general pattern one notices that a function of type $A \to B$ gets the LLLL type $\neg(A \otimes \neg B)$. This should intuitively read that a function takes an argument and a continuation with which to pass the result.

Of course, the interesting aspect of a function is how it computes. Given the sequent calculus lineage, there has to exist a cut to initiate the computation. Here we showcase the computation of swap by performing a cut with an appropriate environment:

$$\begin{aligned}
\langle f \mapsto \mathit{swap}, x \mapsto a, y \mapsto b, k \mapsto K \mid f\ ((x, y), k)\rangle &\longrightarrow \\
\langle z \mapsto ((a, b), K) \mid (z', k) = z; (x, y) = z'; k\ (y, x)\rangle &\longrightarrow \\
\langle z' \mapsto (a, b), k \mapsto K \mid (x, y) = z'; k\ (y, x)\rangle &\longrightarrow \\
\langle x \mapsto a, y \mapsto b, k \mapsto K \mid k\ (y, x)\rangle &
\end{aligned}$$

Computation would then continue as defined by K.

3.1.4 Inlining

Similarly to the substitution semantics of values, we define substitution for commands which we denote as $c[\sigma]$. This will not posses a strong operational meaning, rather it may be viewed as a form of inlining, allowing us to substitute values under the topmost destructor. In the context of a compiler this serves as an important piece of pre-processing.

$$\begin{aligned}
((x, y) = z; c)[\sigma, z \mapsto (a, b)] &= c[\sigma, x \mapsto a, y \mapsto b] \\
(\text{Case } z \text{ of}(x \to c_1 \mid y \to c_2))[\sigma, z \mapsto \text{Inj}_1\ a] &= c_1[\sigma, x \mapsto a] \\
(\text{Case } z \text{ of}(x \to c_1 \mid y \to c_2))[\sigma, z \mapsto \text{Inj}_2\ b] &= c_2[\sigma, y \mapsto b] \\
(z\ a)[z \mapsto [\rho; \lambda x.\ c], \sigma] &= c[\rho, x \mapsto a[\sigma]] \\
((x, y) = z; c)[\sigma] &= (x, y) = z; (c[\sigma]) \\
(\text{Case } z \text{ of}(x \to c_1 \mid y \to c_2))[\sigma] &= \text{Case } z \text{ of}(x \to c_1[\sigma] \mid y \to c_2[\sigma]) \\
(\text{Case } z \text{ of}(x \to c_1 \mid y \to c_2))[\sigma] &= \text{Case } z \text{ of}(x \to c_1[\sigma] \mid y \to c_2[\sigma]) \\
(z\ a)[\sigma] &= z\ (a[\sigma])
\end{aligned}$$

It should be clear that this operation can not return a larger command, due to there existing no copying, as such it serves as a safe optimization.

3.1.5 Type soundness

In order to show that our type system is sound, we prove type preservation and progress for redexes. We note that these corresponds to the theorems presented for the simply typed lambda calculus, see section 2.1.1.

Theorem 3 (Preservation)**.** *If* $r : (\Gamma \vdash)$ *and* $r \longrightarrow r'$ *then* $r' : (\Gamma \vdash)$.

Proof. This can be shown through the inference rules; we do this in the Agda formalisation. Here we ignore the proof derivations and only write out the judgments

with the needed context shape.

$$\langle \sigma, z \mapsto (a, b) \mid (x, y) = z; c\rangle : (\Gamma, \Phi, \Psi \vdash) \longrightarrow \langle \sigma, x \mapsto a, y \mapsto b) \mid c\rangle : (\Gamma, \Phi, \Psi \vdash)$$
$$\langle \sigma, x \mapsto () \mid () = x; c\rangle : (\Gamma \vdash) \longrightarrow \langle \sigma \mid c\rangle : (\Gamma \vdash)$$
$$\langle \sigma, z \mapsto \text{Inj}_1\ a \mid \text{Case } z \text{ of}(x \to c_1 \mid y \to c_2)\rangle : (\Gamma, \Psi \vdash) \longrightarrow \langle \sigma, x \mapsto a \mid c_1\rangle : (\Gamma, \Psi \vdash)$$
$$\langle \sigma, z \mapsto \text{Inj}_2\ b \mid \text{Case } z \text{ of}(x \to c_1 \mid y \to c_2)\rangle : (\Gamma, \Psi \vdash) \longrightarrow \langle \sigma, y \mapsto b \mid c_2\rangle : (\Gamma, \Psi \vdash)$$
$$\langle z \mapsto [\rho; \lambda x.\ c], \sigma \mid z\ a\rangle : (\Psi, \Gamma \vdash) \longrightarrow \langle \rho, x \mapsto a[\sigma] \mid c\rangle : (\Psi, \Gamma \vdash)$$

For all cases but the call it follows immediately from the structure of the typing judgments. For the application case we rely on lemma 1. □

Lemma 1 (Value normalisation)**.** *If* $\sigma : \Gamma \to \Delta$ *and* $\Delta \vdash a : A$ *then* $\Gamma \vdash a[\sigma] : A$

Proof. For (), $\text{Inj}_i\ a$ and x it follows immediately by induction. (a, b) requires a stronger proof of If $\sigma : \Gamma \to \Delta', \Delta$ and $\Delta \vdash a : A$ then $a[\sigma] \to t/\rho$ where $\rho : \Theta' \to \Delta'$, $\Theta \vdash t : A$ and $\Gamma = \Theta', \Theta$. This is proven in the Agda implementation. For $\lambda x.c$ it follows immediately. □

Theorem 4 (Progress)**.** *If* $r : (\{\} \vdash)$ *then there exists a redex* r' *such that* $r \longrightarrow r'$.

Proof. Due to the context of r being empty, it immediately follows since the only way for a computation to get "stuck" is in the presence of an unknown variable. □

If the context is not empty computation could get stuck, consider for example the well typed redex $\langle \sigma, z \mapsto z' \mid (x, y) = z; c\rangle : (\Gamma, z' : A \otimes B \vdash)$ which can not be evaluated due to the variable z'.

Given the proofs of preservation and progress, we say that LLLL's typing is sound. In practice, this means that a closed and well typed redex can not get stuck during evaluation.

3.1.6 Overcoming linearity

While linearity has its perks, it is sometimes needlessly restrictive. Consider if LLLL where extended with machine integers, Int, naturally these could be copied or discarded since they inhibit no side effects.

Consider the boolean representation $1 \oplus 1$, we may weaken and contract it by means of inspection. Similarly, a 32bit integer is logically equivalent to $1 \oplus 1 \oplus \cdots \oplus 1$.

We may therefor have specialised inference rules without loss of generality [8]. As an example, consider how we may discard a value $x : 1 \otimes (1 \oplus 1)$ in context Γ.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash c}{\Gamma, x : 1 \vdash () = x; c} \quad \dfrac{\Gamma \vdash c}{\Gamma, y : 1 \vdash () = y; c}}{\Gamma, z : 1 \oplus 1 \vdash \text{Case } z \text{ of}(x \to () = x; c \mid y \to () = y; c)}}{\Gamma, y : 1, z : 1 \oplus 1 \vdash () = y; \text{Case } z \text{ of}(x \to () = x; c \mid y \to () = y; c)}}{\Gamma, x : 1 \otimes (1 \oplus 1) \vdash (y, z) = x; () = y; \text{Case } z \text{ of}(x \to () = x; c \mid y \to () = y; c)}$$

In fact, all types that have positive polarity can be weakened and contracted freely [18]. Henceforth we will simply refer to this class of types as *data*. More formally, we define the class of data inductively as:

- $1, \text{Int} \in D$
- $A \otimes B \in D$ iff $A, B \in D$
- $A \oplus B \in D$ iff $A, B \in D$

With this, we extend LLLL with two *macro* commands,

$$\dfrac{\Gamma, x : A, y : A \vdash c \qquad A \in D}{\Gamma, x : A \vdash y = \text{copy } x; c}\textit{copy} \qquad \dfrac{\Gamma \vdash c \qquad A \in D}{\Gamma, x : A \vdash \text{discard } x; c}\textit{discard}$$

3.1.7 Of course there is copying!

Not allowing any duplication and discarding is rather limiting. Thus, we add ! to our language. Values and commands get extended as follows:

$$v ::= \ldots \mid !a \qquad\qquad c ::= \ldots \mid !y = x; c \mid y = \text{copy}x; c \mid \_ = x; c$$

With the following typing rules:

$$\dfrac{!\Gamma \vdash a : A}{!\Gamma \vdash !a :!A} \qquad \dfrac{\Gamma, y : A \vdash c}{\Gamma, x :!A \vdash !y = x; c} \qquad \dfrac{\Gamma, x :!A, y :!A \vdash c}{\Gamma, x :!A \vdash y = \text{Copy } x; c} \qquad \dfrac{\Gamma \vdash c}{\Gamma, x :!A \vdash \_ = x; c}$$

Complimentary – as this appears to be the more interesting fragment – we give a special introduction and call for bangs that capture commands.

$$\dfrac{!\Gamma, x : A \vdash c}{!\Gamma \vdash !\lambda x.c :!\neg A} \qquad \dfrac{\Gamma \vdash a : A}{\Gamma, f :!\neg A \vdash f\ a}$$

3.1.8 User defined data

Non-recursive data types have an intuitive representation as a sum of product. Syntactically we may extend values with $C\ a_1\ a_2\ \ldots\ a_n$ where C is a constructor name. Commands gets an extended Case that allows matching against all constructors.

Typing and semantics follow naturally based on that of pairs and sums, only requiring some extra care for the constructor tag. We also add records which only generalises products – these can be handled more directly as they do not require any case analyses.

Handling of Recursive data types are however not as clear. We can not have a flat representation – less so one with a fixed size – of a general recursive data type. One method to handle this could be by introducing a new type for pointers along with the duals Alloc and Dealloc.

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \text{Alloc } a : \Box A} \qquad \frac{\Gamma, x : A \vdash c}{\Gamma, x : \Box A \vdash \text{DeAlloc } a; c}$$

From a logical point of view, these rules are similar to the rule for !. From an implementation point of view however, since $\Box$ are still linear, we do not need the overhead of reference counting or garbage collection.

3.2 LLLL2: practical considerations

The Low Level Linear Language, as presented so far, is still fairly abstract. However, since our goal is to approach functional programming in a low-level setting, we will now make some pragmatic adjustments. They main hurdle to overcome is the treatment of negations and representation of data. As such, the main goal of LLLL2 is to tackle those concerns. This section will further introduce an essential feature of our design; the explicit handling of *control stacks* in a functional setting.

3.3 Representing data

The goal of implementing LLLL concretely gives rise to the question of how to represent terms of the language in conventional computer memory. In this section we begin answering this question by means of a graphical representation, which describes contiguous memory areas, potentially connected through pointers.

There are two main forms in which we will represent values: fixed layouts and stacks. We will represent these objects visually as bellow, letting A be fixed and B be a stack.

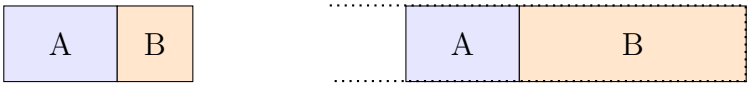
The dotted lines of the stack opening up to the left should indicate that a stack contains free space on top, in this model we will assume that all stacks are unbounded. Consequently, we say that the size of a stack is ∞. In contrast, the size of a fixed layouts can always be given as a statically known number of units.

With these building blocks, we will now give concrete representations for the types of LLLL. For each type we adhere to the rule that if a type has fixed layouts then

all of those layouts will be of the same size, we say that the type have a statically known size. We denote the size of a type A as $|A|$.

Starting with the type 1. The unit 1 contains no information and is thus given a fixed layout of size 0. Visually, the empty layout is represented as a line.

$A \otimes B$ denotes the conjunction of A and B. Its data representation is thus the representation of A followed immediately by the representation of B. There are two ways in which this can take form, either both A and B are fixed layouts, or A is fixed and B is a stack. This gives two valid representations:



For the first case we denote its size as the sum of A and Bs sizes, the size of the second case is ∞.

It is not enough to only consider types when determining the representation of terms. For $A_1 \oplus A_2$ to be of any interest there need to be a way to distinguish between the two cases. As such, the injection *tag* must be part of the representation. Furthermore, adhering to the rule that all sizes of fixed layouts need to agree, padding is necessary when tagging the smaller type. For Inj_i the padding has size $max(|A_1|, |A_2|) - |A_i|$. If both A and B have stack representations then no padding is required.

The size of $A \oplus B$ representation is $1 + max(|A|, |B|)$ where 1 is needed for the tag. In the stack case the size is ∞.

Next would be the representation of $\neg A$, however, as we will argue, $\neg A$ in its current form is too general. More precisely, we recognize that there are multiple valid representation and that deciding upon one alone would limit the use of the language. Furthermore, since there now exists a concrete representation of values, the need for a *calling convention* arises. Both of these concerns will be addressed in the next section.

3.3.1 The many interpretations of negations

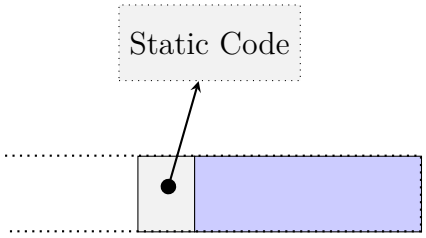
With the introduction of data representations of values from the previous section, we concluded that there are multiple justified representations of $\neg A$. Thus, instead of only allowing one representation we instead opt to divide the single negation, $\neg$, from LLLL1 into three negations that exbibit different data representations and characteristics.

$$A, B ::= \ldots \mid \$A \mid \sim A \mid \%A$$

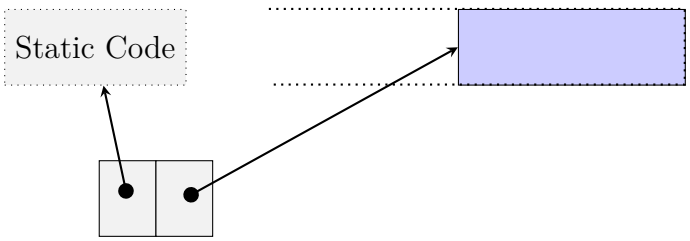
The three negations should be thought of as the following objects:

- $\$A$ a function pointer. That is, a function without an environment.
- $\sim A$ a control stack.
- $\%A$ a coroutine.

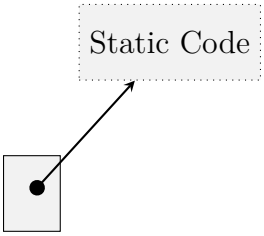
To motivate these different negations we will view them from the system programmer's perspective, especially the C programmer's. The representation of $\sim A$ should be linked to that of a call stack and is one of the main novelties of our design. In C, the stack is an implicit structure that only gets adjusted when entering and exiting functions. In LLLL, the stack is explicit but operates in a similar manner. Crucially, for the stack analogy to exhibit matching characteristic, $\sim A$ has to be linear – indeed, it would be nonsensical to return twice. The representation, as per the style defined in section 3.3, is thus:



$\%A$ should be understood as suspended process. Like $\sim A$, it has its own stack but unlike it, its representation is that of a pointer to a stack.



$\$A$ is a static function and as such can be represented as a label to some static code, much in the same way as a typical C function. This however imposes the limitation that no environment is used, which results in a very simple data layout.



The representation of the negations all share a common thing. The instructions generated by their associated command gets stored in a separate memory location, away from common data. As such, the representation of commands always include a pointer to static code.

As for the typing of the new negations. $\sim$ and % keep the typing of their predecessor

¬, whereas $ must take on a simpler form. We give the formal rules as:

$$\frac{\Gamma, x : A \vdash c}{\Gamma, \vdash \lambda x.c :\sim A} \qquad \frac{\Gamma, x : A \vdash c}{\Gamma, \vdash \lambda x.c : \%A} \qquad \frac{x : A \vdash c}{\vdash \lambda x.c : \$A}$$

The typing for the call remains unchanged for all negations.

Coming back to the pragmatics of the language, we now aim to bring clarity to the question of calling convention. As a consequence of treating call stacks as explicit objects, we impose the invariant that there always exists a stack on to which function arguments can be passed – alternative solutions will be discussed in section 6.2.1. Their respective representations tells us that $\sim$ and % may only accepts argument of a fixed layout, due to them having their own stack, whereas $ has to be called with a stack as argument. Intuitively, at each call a stack becomes "active" on to which arguments are then pushed. In the next section we make this calling convention formal by establishing a kind system.

Before leaving the subject of negation representation, we shortly discuss the split of ¬ in relation to compiler optimization. Part of our idea for LLLL is that a programmer should be able to write code at a chosen level of detail, later relying on a compiler to decompose the whole program from that point. As such, the standard ¬ could be used to signal to the compiler that it may heuristically decompose it into either $\sim$, % or $. There remains a question of how to optimally assign negations for a general program written with the standard ¬ negation. In principle – without the presence of variable size types – ¬ can directly be rewritten as %. Of course, this is not an optimal transformation in terms of data simplicity.

3.3.2 Kinds

Based on the data representations of values, a *kind* system is established. Kinds essentially acts as types of types and have previously been explored in compilers to better support unboxed data [31], function levity and calling convention [32]. For LLLL, kinds are used for three reasons: Firstly, they make formal when a type have a representation in our language. Secondly, they uphold our stack based calling convention, as defined in section 3.3.1. Thirdly, they encode precise sizes of types that are used in compilation.

Using the convention that m and n are sizes of fixed layouts and ∞ is the size of stacks, a kind system is made based on the following judgments:

$$\frac{}{1 : 0} \qquad \frac{A : m \qquad B : n}{A \otimes B : m + n} \qquad \frac{A : m \qquad B : \infty}{A \otimes B : \infty} \qquad \frac{A : m \qquad B : n}{A \oplus B : 1 + max(m, n)}$$

$$\frac{A : \infty \qquad B : \infty}{A \oplus B : \infty} \qquad \frac{A : \infty}{\$A : 1} \qquad \frac{A : n}{\sim A : \infty} \qquad \frac{A : n}{\%A : 2}$$

Note that there are two rules for $\otimes$ and $\oplus$, this is a reflection of their two forms of data representation. Precisely, those cases always lead to valid layouts, whereas, for example $A \otimes B$ would not be valid if $A : \infty$.

The perhaps most important cause for the kind system is to make formal the calling convention of the different negations. From the rules it is clear that $\sim$ and % only accepts arguments of fixed layout, while \$ has to be called with a stack.

3.4 LLLL3: Into primitive pieces

For LLLL3 the focus shifts into primitive representations. Concretely, all functions become top level by explicitly passing environments to the \$ negation. Further, care is put into memory handling by introducing boxing and explicit (de)allocations.

3.4.1 Primitive pieces

Concretely, LLLL3 decomposes the negations $\sim$ and % with the introduction of the following types, terms and commands:

$$A ::= \ldots \mid \bullet \mid \Box A \mid \exists x.A \qquad a ::= \ldots \mid \text{allocStack} \mid \text{box } a \mid \text{deBox } x \mid \langle B, a \rangle$$

$$c ::= \ldots \mid \text{deAlloc}; c \mid \langle A, x \rangle = z; c$$

As for intuition guidance: $\bullet$ is an empty stack, $\Box A$ is a pointer to A and $\exists x.A$ is the – logically – familiar existential quantifier. Precisely how these are used to decompose $\sim$ and % will be explained in section 3.4.2.

The typing rules for the new values and commands are as follows:

$$\frac{}{\vdash \text{allocStack} : \bullet} \qquad \frac{\Gamma \vdash a : A[B/x]}{\Gamma \vdash \langle B, a \rangle : \exists x.A} \qquad \frac{\Gamma \vdash a : A}{\Gamma \vdash \text{box } a : \Box A} \qquad \frac{\Gamma \vdash a : \Box A}{\Gamma \vdash \text{deBox } a : A}$$

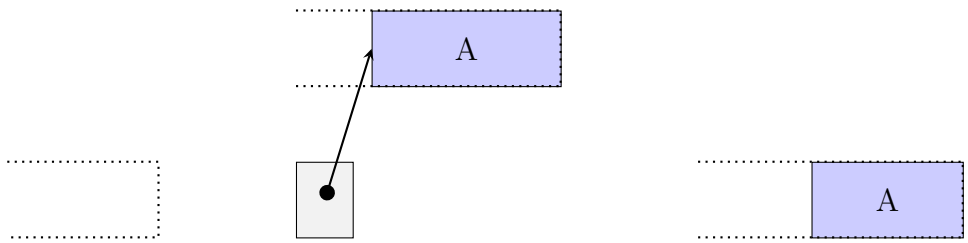
$$\frac{\Gamma \vdash c}{\Gamma, x : \bullet \vdash \text{deAlloc } x; c} \qquad \frac{\Gamma, y : A[B/x] \vdash c}{\Gamma, z : \exists x.A \vdash \langle B, y \rangle; c}$$

We note that $\Box A$ is the only connective with a right elimination rule. This breaks our otherwise completely maintained symmetry. The reasons for this deviation concerns the compilation into the Linear virtual Machine, see section 4.3, for which it is required that only one stack is active at a time.

Further, the semantics of redexes is extended to incorporates these constructs in a natural way:

$$\begin{aligned} \langle \sigma, x \mapsto \text{allocStack} \mid \text{deAlloc} x; c \rangle &\longrightarrow \langle \sigma \mid c \rangle \\ \langle \sigma, z \mapsto \langle B, a \rangle \mid \langle X, x \rangle = z; c \rangle &\longrightarrow \langle \sigma, x \mapsto a \mid c \rangle \\ \langle \sigma, z \mapsto \lambda x.c \mid z\ a \rangle &\longrightarrow \langle x \mapsto a[\sigma] \mid c \rangle \end{aligned}$$

Lastly we give the data representations, in order of $\bullet$, $\square$, $\exists$:



One sees that $\bullet$ is the empty stack, $\square A$ is a pointer to a stack A and $\exists x.A$ simply represents A without any modifications.

3.4.2 Existence of closures

Our treatment of negations gave rise to different runtime representation for each of the continuation types. However, the capturing of free variables, i.e. variables bound by the environment, in lambdas is not as explicit as wanted at compilation. To address this, we further decompose lambdas of types $\sim$ and $\%$ – which capture their environment – into \$ – which does not capture its environment – by explicitly pairing the argument with the captured environment. For $\sim A$, the following conversion gives an intuitive picture of how this can be done for values and commands:

$$\lambda x.c \longrightarrow \lambda x'.(x,\gamma) = x'; \{split\ \gamma\}; c \qquad \text{call } x\ v \longrightarrow (f,\gamma) = x; \text{call } f\ (v,\gamma)$$

Two problems however arise from this translation once typing is considered. Firstly, what is the type of γ? Intuitively, a context $\Gamma = x_1 : A_1, x_2 : A_2, \ldots x_n; A_n$ can be represented as the type $A_1 \otimes A_2 \otimes \cdots \otimes A_n$. For simplicity, we will therefore allow ourselves to simply write Γ when it is clear that it acts as a type.

Secondly, simply assigning the type $\sim A = \$(A \otimes \Gamma) \otimes \Gamma$ loses generality as the environment type is now concrete. For example, in the presence in higher-order functions, this typing becomes a limiting factor since the function now has to specify some concrete type for the environment. The type of the environment should not matter, and is in general not known by the caller. We solve this with the use of existential quantifiers which capture the type of the environments. With this we reach a conversion of types:

$$\sim A = \exists\gamma.(\$(A \otimes \gamma) \otimes \gamma) \qquad \%A = \exists\gamma.(\$(A \otimes \gamma) \otimes \square\gamma)$$

The translation presented above is, however, only possible if the environment contains a stack – due to the calling convention of \$. In reality there are two cases that needs to be considered: When the captured environment is of variable size:

$\lambda x.c[\gamma : \Gamma]$	$\langle \Gamma, (\lambda z.(x,\gamma) = z; \ldots = \gamma; c, \gamma)\rangle$
$call\ z\ a$	$\langle \Gamma, z'\rangle = z; (f,\gamma) = z'; call\ f\ (a,\gamma)$

where $\ldots = \gamma; c$ is a syntactic shorthand for repeated – pairwise – splits of the environment.

When the captured environment is of fixed size we have to allocate a stack:

$\lambda x.c[\gamma : \Gamma]$	$\langle(\Gamma \otimes \bullet), (\lambda z.(x, z') = z; (\gamma, z'') = z'; deAlloc\ z''; \ldots = \gamma; c, (\gamma, allocStack)\rangle$
$call\ z\ a$	$\langle \Gamma, z'\rangle = z; (f, \gamma) = z'; call\ f\ (a, \gamma)$

Similar conversions are applied to %: In effect, the only change needed compared with $\sim$ is to box respectively unbox the environment.

3.4.3 Kind preservation

A correctness condition, for the purpose of modularity, of the closure conversion is that it preserves kinds. Before proving this statement, the new types are added to the kind system. Using the convention that m and n are fixed layouts, ∞ is the variable size layout and a could be either:

$$\frac{}{\bullet : \infty} \qquad \frac{A : a}{\Box A : 1} \qquad \frac{\begin{array}{c}[x : \infty] \\ \vdots \\ A : a\end{array}}{\exists x.A : a}$$

The proof follows the judgments. When its environment is of variable size one derives the following, assuming a finite argument type $A : n$:

$$\dfrac{\dfrac{\dfrac{\dfrac{A : n \qquad \Gamma : \infty}{A \otimes \Gamma : \infty}}{\$(A \otimes \Gamma) : 1} \qquad \Gamma : \infty}{\$(A \otimes \Gamma) \otimes \Gamma : \infty}}{\dfrac{\exists \Gamma.\$(A \otimes \Gamma) \otimes \Gamma : \infty}{\sim A : \infty}} \qquad \dfrac{\dfrac{\dfrac{\dfrac{A : n \qquad \Gamma : \infty}{A \otimes \Gamma : \infty}}{\$(A \otimes \Gamma) : 1} \qquad \dfrac{\Gamma : \infty}{\Box \Gamma : 1}}{\$(A \otimes \Gamma) \otimes \Box \Gamma : 2}}{\dfrac{\exists \Gamma.\$(A \otimes \Gamma) \otimes \Box \Gamma : 2}{\%A : 2}}$$

Since $\sim$ and % are the only propositions changed by the closure conversion, all other propositions will keep their kind.

3.5 A unified language

In this chapter, we have presented three iterations of LLLL with a clearly shared lineage. A key feature of our design is that all iterations may co-exist in the same language. Part of our idea for LLLL – or even for a language built on top of LLLL – is that the programmer should be able to write code at a chosen level of details, later relying on a compiler to decompose the whole program from that point into an equivalent program with only primitive components. Thus, if the programmers wants to make sure of a certain characteristic they can give that precise decomposition – for parts were such control is not wanted, they instead leave it to the compiler. We

believe that such a design would combine the best features of a highly compostable functional language with the precise control of system-level languages.

Figure 3.3 shows a topological graph over this imagined decomposition for a slightly higher-level language than LLLL.

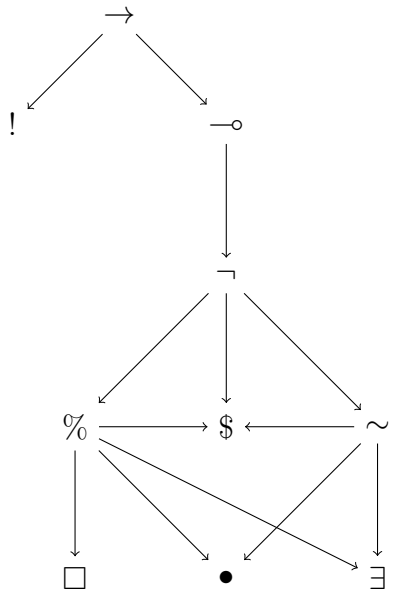


Figure 3.3: Decomposition of functions

4

The Linear Virtual Machine

This chapter presents the Linear Virtual Machine. The Linear Virtual Machine is a pragmatic target for the Low Level Linear Language, presented in chapter 3. While LLLL3 is low-level, it is still at its core a typed functional language, in the sense that it is specified using strong typing rules. To get past the final hurdle we need to separate the abstract concept of terms into concrete data and operations, which we achieve by defining the Linear Virtual Machine. With the translation of LLLL3 to LLVM the logical foundation, as well as the functional style, is lost. We argue however that the gap between them is relatively small and likewise is the gap from LLVM to machine code, and as such the VM can serve to attest to our goal of a C-like execution style.

As a first introduction, the VM re-states the values and commands from LLLL in an assembly like presentation. The Linear Virtual Machine is operationally close to a register-based machine. It has a predominantly sequential execution which operates immediately on memory regions, which it achieves by closely following the general architecture deployed by conventional computers. Notably there is the absence of traditional function calls with arguments, instead arguments are explicitly passed followed by a jump.

4.1 State

The state of the virtual machine consists of an arbitrary number of indexed registers of fixed size, accessed by $reg[i]$, and general memory in the form of a heap. Three registers serve special purposes in the machine, we denote these by `pc`, `sp` and `tsp`. `sp` and `pc` are the usual stack pointer and program counter.

`tsp` is the *target* stack pointer and acts as a complement to `sp`. The general principle is that `sp` is only used for popping and `tsp` in only for pushing. A typical function would start of by popping its arguments from `sp` into registers, perform its actions and lastly set up and push to `tsp` before making a call. After calling a function `sp` will be set to `tsp`, thus the code of a function can always expect its arguments on `sp`.

For the general registers it is sometimes helpful to think of them as part of a contiguous memory area. As we shall see shortly, many instructions operate over a specified range of registers. This use of multiple registers may be unconventional for a more traditional register machine, but it allows for a less cluttered compilation

scheme from LLLL3, see section 4.3. Of course, it should be straightforward to translation these operations to a more familiar instruction set.

4.2 Instructions

The instructions of the virtual machine is given in table 4.1. The influence of LLLL is reflected in the names of the instructions. Furthermore, the duality between values and commands of LLLL leaves its mark over the virtual machine's instructions as there almost exists a split between value instructions and command instructions. We spend some more time discussing this duality aspect in section 6.1.3.

#	OpCode	Arg1	Arg2	Description
0	Move	size	reg	deprecated
1	Inj,	tag	offset	
2	Lambda,	cmdAddr	_	addr to tsp
3	Push,	size	reg	push to tsp from reg
4	AllocS,	_	_	allocate stack, set tsp
5	SaveTsp,	reg	_	reg[arg1] = tsp
6	SetTsp,	reg	_	tsp = reg[arg1]
7	SyncS,	_	_	tsp = sp
8	Pop,	size	reg	pop from stack to reg
9	Case,	reg	_	
10	Jmp,	cmdAddr	_	
11	Call,	reg	_	Call register, swap
12	CallS,	cmdAddr	_	Call static, swap reg and tsp
13	FFCall,	cPtr	_	Call ff and swap
14	DeAllocS,	_	_	
15	SetSp,	reg	_	sp = reg[arg1]

Table 4.1: The Linear Virtual Machine's instructions

While the meaning of many instructions follow immediately from their related LLLL term, some are introduced to make the virtual machine usable as a concrete runtime.

For Case to behave properly it should be directly followed by Jmp instructions, such that a case analyses on tag i leads to a jump to the ith Jmp instructions which then jumps into the code for the corresponding branch. The design of Case allows for matching beyond binary tags, this is chosen to scale better towards more realistic source languages.

While we have not defined a formal semantics for the Linear Virtual Machine, we instead specify its execution by means of a program defined in a C-like language. In fig. 4.1 the operation of the VM is specified by a fetch-decode-execute cycle. `memcpy(a, b, s)` copies s units of data starting at address b to address a.

```
loop {
  (opCode, arg1, arg2) = *(pc++)
  switch(opCode) {
      case Inj:
          tsp -= arg2      // skip offset
          *(--tsp) = arg1 // set tag
      case Lambda:
          *(--tsp) = arg1 // Push code pointer to stack
      case Push:
          tsp -= arg1
          memcpy(tsp, arg2, arg1)
      case AllocS:
          tsp = allocS()
      case SaveTsp:
          reg[arg1] = tsp
      case SetTsp:
          tsp = reg[arg1]
      case SyncS:
          tsp = sp
      case Pop:
          memcpy(arg2, sp, arg1)
          sp += arg1
      case Case:
          pc += reg[arg1] // Relative jump
      case Jmp:
          pc = arg1        // Long jump to labeled address
      case CallS:
          pc = arg1
          sp = tsp
      case Call:
          pc = reg[arg1]
          sp = tsp
      case FFCall:
          ffCall(arg1)
          sp = tsp
      case DeAllocS:
          deAllocS()
      case SetSp:
          sp = reg[arg1]
  }
}
```

Figure 4.1: The fetch-decode-execute cycle of the Linear Virtual Machine

4.3 Translation from LLLL3

The close relationship between LLLL3 and the virtual machine become evident during compilation. The compilation of well typed values and commands, as presented shortly, follow a simple recursive scheme with relatively little bookkeeping.

Before going over the compilation schemes we define some shared preliminaries. For any context Γ there exists a map ρ from variable to register index. Since some variables, say x, refers to data larger than one register we specify that $\rho(x)$ is the *first* consecutive index which contains the data of x.

The function $|\_|$ gives the size of values and variables. Note that, since we only consider well typed terms, this size can always be had through the kind of the terms type.

We let i denote a register index that is followed by arbitrarily many free registers – such that data can always be stored at index i. The introduction of a free register i is simply to avoids distraction from the main algorithm, in a compiler it is straightforward to keep track of free registers. Of course, if registers are of a limited quantity it would complicate matters, we touch slightly upon this difficulty in section 2.4.

An Invariant of the algorithms is that Γ can at most contain one type with kind ∞. If there exists such an element in Γ, lets name it x, then it is always the case that $\rho(x) = \texttt{sp}$. We note that this restriction is naturally upheld as a consequence of our kind system, in particular $\$A$ only allows $A : \infty$ and commands may only create an infinite type by decomposition of another infinite type.

4.3.1 Compiling commands

We define the compilation scheme $[\_](\rho)$, parametrised over the map $\rho : \Gamma$, for well typed commands as follows:

$[() = z; c](\rho) = [c](\rho)$

$[(x, y) = z; c](\rho, z \mapsto \texttt{sp}) = \texttt{Pop}\ |x|\ i;\ [c](\rho, x \mapsto i, y \mapsto \texttt{sp})$
$[(x, y) = z; c](\rho) = [c](\rho, x \mapsto \rho(z), y \mapsto \rho(z) + |x|)$

$[\text{case}\ z\ \text{of}((x_1 \to c_1, x_2 \to c_2)](\rho) = \texttt{Case}\ \rho(z);\ \texttt{Jmp}\ l_1;\ \texttt{Jmp}\ l_2$
where l_i is $[c_i](\rho, x_i \mapsto \rho(z) + o)$ for $o = \text{if}\ \rho(z) = \texttt{sp}\ \text{then}\ 0\ \text{else}\ |z| - |x_i|$

$[f\ a](\rho) = [a];\ \texttt{Call}\ \rho(f)$

$[\langle X, x\rangle = z; c](\rho) = [c](\rho, x \mapsto \rho(z))$

$[\text{deAlloc}\ z; c](\rho, z \mapsto \texttt{sp}) = \texttt{DeAllocS};\ [c](\rho)$

Note that splits of the finite pair have no VM instruction counterpart. Indeed, due to our flat data representation, the offset to a record's fields are statically determined during compilation and as such splits have no semantic meaning on this level. Splits of pairs of ∞ kind, on the other hand, gets interpreted as a pop.

4.3.2 Compiling values

The intuitive reading of $\Gamma \vdash a : A$ is that a is a recipe on how to construct a value of type A by consuming Γ. As such, the generated code for a are the instructions which build the data representation of a. If $A : n$, the data representation gets built on top of `tsp` while if $A : \infty$ we point `tsp` to its result.

We extend the compilation scheme $[\_](\rho)$ for well typed values. As a syntax note, we will mostly write $[\_]$, this is to remove clutter since ρ is never modified while compiling values.

$[()] = \_$
where _ denotes no operation

$[x](\rho, x \mapsto \texttt{sp}) = \texttt{SyncS}$
$[x](\rho) = \texttt{Push}\ |x|\ \rho(x)$

$[(a, b)] = [b]; [a]$

$[\text{inj}_i\ a] = [a]; \texttt{Inj}\ i\ o$
where $o =$ if $|\text{inj}_i\ a| = \infty$ then 0 else $|\text{inj}_i\ a| - 1 - |A_i|$

$[\lambda x.c] = \texttt{Lambda}\ l$
where l is a label to the code segment $[c](x \mapsto \texttt{sp})$

$[\langle A, a \rangle] = [a]$

[allocStack] = `AllocS`

[box a] = `SaveTsp` i; $[a]$; `SaveTsp` j; `SetTsp` i; `Push` 1 j

[deBox x] = `SetTsp` $\rho(x)$

We note a few things: In compiling (a, b), we compile the right element first, this is because our stack grows to the left and the last element pushed thus becomes the leftmost at the stack. When compiling a variable $x : A$ there are two cases to consider: $A : n$ results in pushing to `tsp` and $x : \infty$r results in a *SyncS*, that is, setting `tsp` to `sp`. The consequence of this is that we never move a stack.

5

Usage, examples and analysis

We have implemented a compiler for LLLL as well as a runtime for the Linear Virtual Machine. The compiler does not cover all of LLLL, but a version close to that of LLLL2. It contains the negations \$, $\sim$ and %, an integer type, Int, with the primitive functions add, sub, mul, div : \$($Int \otimes Int \otimes \sim Int$) and print : \$($Int, \sim 1$).

Special care is also given to recursion. While we have not given any treatment of recursion in the theoretical language, it is crucial in a practical language. Our solution is rather primitive; allow all top-level declarations to live in a separate, non-linear, scope. This in turn is justified since all top-level declarations are declared with an empty environment. A variable can then either refer to a linear variable in the current environment or to a top level declaration.

5.1 Swap

The swap example from section 3.1.3 can be written as a function in our concrete language as:

```
swap : $((A * B) * ~(B * A))
     = \z.
       (z', ret) = z;
       (x, y) = z';
       call ret (y, x);
```

From this code, the compiler performs closure conversion, as described in section 3.4. The resulting program is shown below:

```
swap : $((A * B) * Exist x. $((B * A) * x) * x)
     = \z.
       (z', ret) = z;
       (x, y) = z';
       <Env, ret> = ret;
       (k, env) = ret;
       call k ((y, x) * env);
```

Lastly, given the sizes 3 and 4 for the types A and B, this can be compiled, following the scheme outlined in section 4.3, to the following VM code:

```
swap:
  Pop 7 0
  Pop 1 7
  SyncS
  Push 3 0
  Push 4 3
  Call 7
```

5.2 Fibonacci sequence

As a slightly more extensive example, we define a program printing the Fibonacci sequence:

```
main : $(~1)
    = \ret.
      call fib ((0, 1), ret);

fib : $((Int * Int) * ~1)
    = \z.
      (z', ret) = z;
      (y, x) = z';
      x' = copy x;
      call print (x', \k.
      discard k;
      y' = copy y;
      call add ((x,y'), \t.
      call fib ((y,t), ret)));
```

Here we are utelising the primitive functions add and print as well the ability to refer to top-level declarations. Compiling the LLLL program yields the VM code below:

```
main:
  SyncS
  Inj 1 0
  Inj 0 0
  CallS "fib"

fib:
  Pop 2 0
  SyncS
  Push 1 1
  Push 1 0
  Lambda fib_2
  Push 1 0
  FFCall Print
fib_2:
  Pop 1 0
```

```
  Pop 1 1
  SyncS
  Push 1 1
  Lambda fib_3
  Push 1 1
  Push 1 0
  FFCall Add
fib_3:
  Pop 1 0
  Pop 1 1
  SyncS
  Push 1 0
  Push 1 1
  CallS fib
```

6

Conclusion

We have shown that it is possible to construct a language with a strong logical foundation while simultaneously being explicit about low-level details such as calling convention and data representation. We have achieved this goal by refining the language in small successive steps that transform higher level features into more primitive objects closer to that found in conventional machines.

6.1 Discussion

During the course of this thesis a lot of interesting ideas arose that did not make in into this report. In this section we will discuss some of these ideas and observations.

6.1.1 A syntactic division of types

To handle runtime representations we make use of kinds which capture the size of each type. This is central part of our design but it is not without downside. Since we allow multiple kind rules per type, see the rules for $\otimes$ and $\oplus$, types are not enough to determine the run-time representation. Furthermore we are somewhat inconsistent in what should be handled through typing or kinding; $\sim$ and % are separated by type but their defining difference lies in their representation which could be expressed by kinding. Another such case, where the typing but not representation is identical, is that of 1 and $\bullet$.

An alternative could be to divide types into two syntax categories; fixed and variable. Fixed types would always have an exact size whereas variable types would always have a variable size. Naturally, some variable types would be build on top of finite types and vice versa to allow for interplay. This split is akin to some work on polarity, see section 2.1.3.1, in which polarity *shifts* are explicit through syntax [7]–[9]. This change could make it easier to express intent directly by types as we do not have to consider another layers of indirection in the form of kinds. Of course, the downside is that it would make the syntax more verbose, especially if term syntax can not be shared by multiple types. We leave this choice open for future work.

6.1.2 Copying and discarding

In this thesis we have not paid significant attention to copying. We have followed the original formulation of linear logic were weakening and contracting are limited

to !-types [13]. A "manual" form of copying is possible for concrete types, such as $1 \otimes 1$, due to first decomposing them and then rebuilding twice. This can however become costly. Copying $1 \oplus 1$ in this manner compiles down to a case analysis and an indirect jump, clearly, it would be more efficient to simply copy the (one bit!) memory region.

From the logical point of view, contraction and weakening have been explored through polarity. Girard [18] proposed a unified system of logic which incorporates both linear and non-linear propositions. This system revolves around sequents of the form $\Gamma; \Gamma' \vdash \Delta'; \Delta$ in which Γ and Δ are linear and Γ and Δ are non-linear, i.e. allows weakening and contraction. Naturally, any non-linear proposition can be converted to a linear one, but he goes further on to specify rules that allows moving from the linear context to the non-linear context. We showcase this action by the left rules:

$$\frac{\Gamma; A, \Gamma' \vdash \Delta'; \Delta}{\Gamma, !A; \Gamma' \vdash \Delta'; \Delta} \qquad \frac{\Gamma; P, \Gamma' \vdash \Delta'; \Delta}{\Gamma, P; \Gamma' \vdash \Delta'; \Delta}$$

where P has *positive* polarity and A has any polarity.

This system could be a promising basis for a slightly more lenient language that still can enforce linearity when needed.

6.1.3 Duality of the Linear Virtual Machine

Similarly to the division between values and commands in LLLL, the Linear Virtual Machine had for a long time during its development a split instruction set and two operation modes. The two instruction sets later got merged due to difficulties during development, but we believe that the split could be reintroduced. Perhaps the question is weather this is even a wanted aspect.

An immediate benefit to the split instruction set is that the opcode tag could be reduced and allows the two instruction sets to have overlapping byte representations. An immediate downside is the extra complicity and overhead of changing between operation modes.

6.2 Future work

The contributions of this thesis lays in the in-between, that is, we neither specify a high level language nor do we compile all the way down to machine code. As such we feel that there are two clear directions to continue upon this work.

Firstly, to build a compiler from a high-level functional language that targets LLLL. Proof transformations between intuitionistic logic and linear logic exists [13] and we believe that a similar conversion should be possible from a typical lambda calculus based IR into LLLL. From a more pragmatic point of view, the transformation would essentially be a typed CPS translation. One question that arises is how the manual memory management aspects of LLLL could be expressed in a more conventional functional language. One theoretically interesting possibility would be if this could simply be annotated through typing, but we leave this to continued research.

Secondly, to bypass the virtual machine runtime by compiling directly into machine code. Due to the assembly-like nature of the Linear Virtual Machine, we believe that there should exist a relatively straightforward approach for this. The main difficulty, we suspect, is register allocation. That said, the subject of register allocation and spillage have an extensive literature [3], [33], [34].

An empirical evaluation of our design can only be done given the presence of the above. Until that point, evaluation has to be based on other means such as coherence of design and extensibility for practical implementations.

Disregarding the fact that our intermediate representation lacks a context in which in acts as an intermediate, there are some concrete aspects in which LLLL could be improved upon. We discuss some of these in the upcoming sections.

6.2.1 Calling convention

Current implementation forces all arguments to be passed on the stack and they may in general only be used once they are pushed out of the stack. This makes loops inefficient as we would like arguments to stay in their assigned registers. A more sophisticated analysis could detect cases like these, but ultimately a more systematic approach is preferred.

An alternative to sophisticated analysis – going along the main theme of this thesis – is to integrate calling convention with typing. As it stands now, for a function $f : \$A$, $\$A$ essentially gives the calling convention; A represents the exact shape of the stack that f expects. The problem, however, is that the language is not expressive enough to denote arguments outside of this stack. Optimally, if the typing was more refined, it should be possible to specify register moves directly.

Section 6.2.2 builds upon the ideas presented here but frames it as a more applied setting by considering integration with existing languages – predominantly C.

6.2.2 Linking with C

An initial motivation for our design was to be able to integrate with C in a very direct way. Many languages offer a foreign function interface, which often follows the C calling convention. A C interface for LLLL could be achieved with minimal overhead if the memory layout of our programs matched that of C's and that they share a common calling convention.

We give a quick overview of how this would be possible. Given a C function such as

```
int foo(x :  char, y :  double) { ... }
```

we would represent it in LLLL2 as something akin to

$$foo : \$(\text{char} \otimes \text{double} \otimes \sim \text{int})$$

The key idea here is to capture the return value in the continuation $\sim$ Int whose representation should exactly match that of the C call stack. By doing this, it would be possible to make a direct call to a C function without any overhead.

A more thorough investigation of how this could be realised is an interesting continuation of our work. Some potential research questions in this direction could be: Can all C types be converted to types in our system? What is the limit when converting LLLL types to C types?

6.2.3 Wider support for functional constructs

There are some noteworthy omissions in our functional language that are essential in a more mature functional language. Most pressingly, we have not developed a theory for how to support recursive data types. This is something that needs to be addressed in future works.

6.3 Conclusion

We have shown that linear logic can be leverage to design a functional programming language following a more C-like execution model. In particular, we emphasise that linearity helps in two main aspects: Firstly, it reduces the memory management bookkeeping typically required by functional languages, instead memory management is explicit in the language – whilst remaining safe due to typing. Secondly, linearity encourages a predominantly flat data representation – as is tradition for system languages – since sharing is of lesser prevalence.

An interesting consequence of our goal, is that it made the control stack an explicit part of the language. In this sense, we go beyond what is typically expressible in a C-like language, but this was necessary in our language to allow for higher order functions – which are vital for functional languages.

As explained in section 6.2, there exists much needed work before our design can be fully evaluated. In its current state, we judge LLLL based on its coherence and control capabilities. Coherence is displayed in the close relation to a linear logic proof system, which has guided the design of the language throughout. Control is reflected in part due to the lack of runtime system and part due to the minimal compilation scheme. To end, we feel positively about the potential of LLLL and hope to see continued work on it and the paradigm shift it represents.

Bibliography

[1] J. Hughes, "Why functional programming matters," *The computer journal*, vol. 32, no. 2, pp. 98–107, 1989.

[2] S. L. Peyton Jones, *The implementation of functional programming languages (prentice-hall international series in computer science)*. Prentice-Hall, Inc., 1987.

[3] A. W. Appel, *Compiling with continuations*. Cambridge university press, 2007.

[4] W. A. Howard *et al.*, "The formulae-as-types notion of construction," *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, vol. 44, pp. 479–490, 1980.

[5] G. Gentzen, "Untersuchungen über das logische schließen. i.," *Mathematische zeitschrift*, vol. 35, 1935.

[6] J.-Y. Girard, P. Taylor, and Y. Lafont, *Proofs and types*. Cambridge university press Cambridge, 1989, vol. 7.

[7] P. Downen and Z. M. Ariola, "Duality in action (invited talk)," in *6th International Conference on Formal Structures for Computation and Deduction (FSCD 2021)*, Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2021.

[8] N. Zeilberger, "On the unity of duality," *Annals of pure and applied logic*, vol. 153, no. 1-3, pp. 66–96, 2008.

[9] A. Spiwack, "A dissection of l," *Unpublished draft*, 2014.

[10] P. Downen and Z. M. Ariola, "A tutorial on computational classical logic and the sequent calculus," *Journal of Functional Programming*, vol. 28, e3, 2018.

[11] P. Downen, L. Maurer, Z. M. Ariola, and S. Peyton Jones, "Sequent calculus as a compiler intermediate language," in *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, 2016, pp. 74–88.

[12] P.-L. Curien and H. Herbelin, "The duality of computation," *ACM sigplan notices*, vol. 35, no. 9, pp. 233–243, 2000.

[13] J.-Y. Girard, "Linear logic," *Theoretical computer science*, vol. 50, no. 1, pp. 1–101, 1987.

[14] J.-P. Bernardy, M. Boespflug, R. R. Newton, S. Peyton Jones, and A. Spiwack, "Linear haskell: Practical linearity in a higher-order polymorphic language," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, pp. 1–29, 2017.

[15] P. Wadler, "Linear types can change the world!" In *Programming concepts and methods*, Citeseer, vol. 3, 1990, p. 5.

[16] V. L. Juan, "Efficient functional programming using linear types: The array fragment," 2015.

[17] J.-M. Andreoli, "Logic programming with focusing proofs in linear logic," *Journal of logic and computation*, vol. 2, no. 3, pp. 297–347, 1992.

[18] J.-Y. Girard, "On the unity of logic," *Annals of pure and applied logic*, vol. 59, no. 3, pp. 201–217, 1993.

[19] S. L. P. Jones, "Implementing lazy functional languages on stock hardware: The spineless tagless g-machine," *Journal of functional programming*, vol. 2, no. 2, pp. 127–202, 1992.

[20] S. Marlow and S. P. Jones, "Making a fast curry: Push/enter vs. eval/apply for higher-order languages," *ACM SIGPLAN Notices*, vol. 39, no. 9, pp. 4–15, 2004.

[21] Y. Lafont, "The linear abstract machine," *Theoretical computer science*, vol. 59, no. 1-2, pp. 157–180, 1988.

[22] S. Abramsky, "Computational interpretations of linear logic," *Theoretical computer science*, vol. 111, no. 1-2, pp. 3–57, 1993.

[23] I. Mackie, "Lilac: A functional programming language based on linear logic," *Journal of Functional Programming*, vol. 4, no. 4, pp. 395–433, 1994.

[24] R. Milner, "A theory of type polymorphism in programming," *Journal of computer and system sciences*, vol. 17, no. 3, pp. 348–375, 1978.

[25] J. S. Hodas, K. M. Watkins, N. Tamura, and K.-S. Kang, "Efficient implementation of a linear logic programming language.," in *IJCSLP*, 1998, pp. 145–159.

[26] J. S. Hodas and D. Miller, "Logic programming in a fragment of intuitionistic linear logic," *Information and computation*, vol. 110, no. 2, pp. 327–365, 1994.

[27] M. Hofmann, "A type system for bounded space and functional in-place update," in *Programming Languages and Systems: 9th European Symposium on Programming, ESOP 2000 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2000 Berlin, Germany, March 25–April 2, 2000 Proceedings 9*, Springer, 2000, pp. 165–179.

[28] A. Ohori, "The logical abstract machine: A curry-howard isomorphism for machine code," in *International Symposium on Functional and Logic Programming*, Springer, 1999, pp. 300–318.

[29] C. Bruggeman, O. Waddell, and R. K. Dybvig, "Representing control in the presence of one-shot continuations," in *Proceedings of the ACM SIGPLAN 1996 conference on Programming Language Design and Implementation*, 1996, pp. 99–107.

[30] A. Filinski, "Linear continuations," in *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1992, pp. 27–38.

[31] S. L. P. Jones and J. Launchbury, "Unboxed values as first class citizens in a non-strict functional language," in *Functional Programming Languages and Computer Architecture: 5th ACM Conference Cambridge, MA, USA, August 26–30, 1991 Proceedings 5*, Springer, 1991, pp. 636–666.

[32] P. Downen, Z. M. Ariola, S. Peyton Jones, and R. A. Eisenberg, "Kinds are calling conventions," *Proceedings of the ACM on Programming Languages*, vol. 4, no. ICFP, pp. 1–29, 2020.

[33] G. J. Chaitin, "Register allocation & spilling via graph coloring," *ACM Sigplan Notices*, vol. 17, no. 6, pp. 98–101, 1982.

[34] M. Poletto and V. Sarkar, "Linear scan register allocation," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 21, no. 5, pp. 895–913, 1999.