Towards a System-Level Functional Language: Lithium

Master's thesis in Computer science and engineering

Sebastian Selander
Samuel Hammersberg

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Master's Thesis 2025

Towards a System-Level Functional Language: Lithium

Sebastian Selander
Samuel Hammersberg

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

Towards a System-Level Functional Language: Lithium

Sebastian Selander
Samuel Hammersberg

Supervisor: Jean-Philippe Bernardy, Computer Science and Engineering
Examiner: Patrik Jansson, Computer Science and Engineering

Master’s Thesis 2025
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 **Typst**
Gothenburg, Sweden 2025

Acknowledgments

We would like to thank our supervisor Jean-Philippe Bernardy for his thesis idea as well as his enthusiastic involvement, support, and guidance throughout the project.

Abstract

Functional programming has a rich and studied history. In functional programming, large problems can be described by the composition of smaller building blocks. Despite its benefits, functional programming has struggled to find its way into system-level programming. By leveraging the restrictions linear types impose, functional programming languages can be applied to system-level programming without sacrificing performance.

This thesis presents Lithium, a system-level functional programming language that is based on a variant of linear logic. The purpose of Lithium is as an intermediate compilation target. We give the typing and kinding rules for Lithium before describing a series of transformations to turn Lithium into a language that is easily translated into assembly code. Additionally, we present a compilation scheme, a mapping from types to memory, and the application binary interface (ABI).

Contents

Figures

1. Introduction

The concept of linearity in programming languages is not a newly discovered one. A language with linearity allows a developer to write software without the worries of mismanaging things such as memory and files. While there are benefits to linearity, not many languages have opted to implement it. This concept is most commonly found in functional programming, but one thing functional languages commonly do not support is system-level programming.

This thesis introduces a new system-level functional programming language, Lithium, which tries to fill this void.

1.1. Background

System-level programming is the act of developing software that interacts directly with a computer's hardware or provides foundational services to other software. We will define system-level programming as Wikipedia (Wikipedia, 2024) defines it:

- Programs can operate in resource-constrained environments
- Programs can be efficient with little runtime overhead, possibly having either a small runtime library or none at all
- Programs may use direct and "raw" control over memory access and control flow
- The programmer may write parts of the program directly in assembly language

For system-level programming today developers mostly choose between procedural programming languages, for instance: C, C++, or Rust. In C memory management is manual, and in C++ there are options for automatic memory management, but the compile time guarantees are few and weak. Rust has mostly solved the issue of memory management by introducing an ownership model and a borrow checker (Matsakis & Klock, 2014).

Although Rust offers many functional aspects, many are still missing, for instance: *purity* and *referential transparency.*

1.2. Motivation

Functional programming tends to emphasize *referential transparency*, *higher-order functions*, *algebraic type systems*, and *strong type systems.* Although the merits of functional programming are evident (Hughes, 1989), it is underrepresented for system-level programming. Functional languages are rarely used in system-level programming due to their lack of predictable performance, which can be traced back the use of *immutable* data structures, as opposed to *mutable* data structures. The former requires copying, and subsequently a form of automatic memory management, at least for convenience, whereas the latter can be modified in place.

A popular and effective optimization for functional languages is *fusion* (Gill et al., 1993). Fusion aims to remove intermediate data structures, improving performance and reducing memory usage. However, it can be difficult to predict whether fusion is effective for arbitrary compositions. Linear types provide a framework for fusion where, rather than letting a compiler decide heuristically whether fusion ends up duplicating or sharing work, it is specified by the types of the composed functions (Bernardy et al., 2016).

Girard's linear logic (Girard, 1987) is a refinement of classical and intuitionistic logic, where, rather than propositions being truth statements, they represent *resources*, meaning propositions are objects that can be modified into other objects. Linear logic models the problems of shared and mutable data, both of which are of critical importance in system-level programming. In linear logic, the uses of weakening and contraction are carefully controlled, which in a programming language setting means variables must be used exactly once. Because the use of resources is carefully controlled, mutable data structures can be used safely.

The goal for Lithium is not only to be used as a system-level level language, but also as an intermediate compilation target for (linear) functional programming languages.

1.3. Related Work

A lot of research has gone into researching linear logic, and linear logic in programming languages. Not much research has been done in actually implementing functional system-level linear languages.

In the previous year (2024), Nordmark wrote his master's thesis "Towards a practical execution model for functional languages with linear types" (Nordmark, 2024). This work can be seen as a sort of predecessor of the work presented in this thesis, and is similar in nature to his thesis. Nordmark compiled a language similar to Lithium to an untyped byte code, which ran in a custom virtual machine. Even though using a virtual machine is a suitable first step, we deem that this is not low-level enough for a system-level language.

2. Background

The Curry-Howard correspondence is the direct relation between logic and computer programs (Howard & others, 1980). The correspondence is commonly called the proofs-as-programs and propositions-as-types, because proofs correspond to programs and propositions correspond to types. Where a logician may write a proposition, and a proof of that proposition, a programmer may write a type and a (type correct) program. We take the approach from the programmer's perspective, i.e. programs and types.

This chapter introduces the necessary context for Lithium. We start by introducing the calculus in the form of the typed lambda calculus and linear types. We continue by explaining continuation-passing style and its relevance in compilers. The chapter finishes by presenting different compilation targets and their upsides and downsides.

2.1. Lambda calculus and linear types

This section aims to remind the reader of the lambda calculus and some of its variants. We assume familiarity with the untyped lambda calculus and typing rules. We will introduce the simply typed lambda calculus, then extending it with polymorphic types, and end by introducing linear types.

2.1.1. Simply Typed Lambda Calculus

Simply typed lambda calculus was first introduced by Alonzo Church to avoid the paradoxical use of the untyped lambda calculus (Church, 1940). It consists of two separate syntactic categories: the category of types and the category of terms. The two categories correspond to types and computation, respectively. The syntax for simply typed lambda calculus is shown in Figure 1. The symbol T is used to denote base types.

$$\textit{Types } \sigma, \tau ::= \sigma \to \tau \mid T$$

$$\textit{Terms } e_1, e_2 ::= x \mid e_1 e_2 \mid \lambda x.e_1$$

Figure 1: Syntax for simply typed lambda calculus.

To ensure that terms in simply typed lambda calculus are well-typed, we define a relation between terms and types. The typing relation uses the syntax $\Gamma \ \vdash e : \sigma$, which says that in environment Γ, the term e has type σ. The environment Γ is a mapping from free variables to types. $\Gamma, x : \sigma$ is the environment that extends Γ by associating the variable x with σ. If a dot $(\cdot)$ is used in place of Γ then the environment is empty. The typing rules for simply typed lambda calculus are shown in Figure 2.

$$\frac{}{\Gamma, x : \sigma \ \vdash x : \sigma}\text{Var} \qquad \frac{\Gamma, x : \sigma \ \vdash e : \tau}{\Gamma \ \vdash \lambda x : \sigma.\ e : \sigma \to \tau}\text{Abs}$$

$$\frac{\Gamma \ \vdash e_1 : \sigma \to \tau \quad \Gamma \ \vdash e_2 : \sigma}{\Gamma \ \vdash e_1 e_2 : \tau}\text{App}$$

Figure 2: Typing rules for simply typed lambda calculus.

The first rule, Var, says that $x : \sigma$ is a term if the environment Γ contains the variable $x : \sigma$. The rule Abs, short for abstraction, also commonly called "lambda" says that if the term $e : \tau$ can be deduced in the environment Γ extended with $x : \sigma$, then in the environment Γ the term $(\lambda x : \sigma.\ e)$ has type $\sigma \to \tau$. The last rule, App, short for application says that if in the environment Γ we have $e_1 : \sigma \to \tau$ and $e_2 : \tau$ then in the environment Γ the term $e_1 e_2$ has type τ.

2.1.2. Polymorphic Lambda Calculus

System F is a typed lambda calculus that introduces universal quantification over types (Girard, 1972). It was independently discovered by logician Girard in 1972 (System F) and in 1974 by computer scientist Reynolds (Polymorphic lambda calculus).

We extend the simply typed lambda calculus grammar with type variables (α) and universal quantification $(\forall)$. The grammar of terms is also extended with type abstraction $(\Lambda\alpha.e)$ and type application $(e[A])$.

$$\begin{aligned} \textit{Types}\ A &::= A \to A \mid T \mid \alpha \mid \forall\alpha.A \\ \textit{Terms}\ e &::= x \mid ee \mid \lambda x.e \mid \Lambda\alpha.e \mid e[A] \end{aligned}$$

Where in simply typed lambda calculus variables range over terms and lambdas have binders for terms, System F additionally introduces variables ranging over types as well as binders for types. The environment Γ is no longer only a mapping from variables to types, it also includes type variables. Shown in Figure 3 are the rules for type abstraction and type application.

$$\frac{\Gamma, \alpha \ \vdash e : \sigma}{\Gamma \ \vdash \Lambda\alpha.\ e : \forall\alpha.\sigma}\textit{TAbs} \qquad \frac{\Gamma \ \vdash e : \forall\alpha.\sigma}{\Gamma \ \vdash e[\tau] : \sigma[\tau/\alpha]}\textit{TApp}$$

Figure 3: Type abstraction and type application rules in System F.

The syntax $\sigma[\tau/\alpha]$ means replace each occurrence of α with τ in σ. In the polymorphic lambda calculus we can implement the identity function. The proof for the identity function and the identity function applied to the variable y with type A can be seen in Figure 4 and Figure 5.

$$\dfrac{\dfrac{\dfrac{}{\Gamma, \alpha, x : \alpha \vdash x : \alpha}}{\Gamma, \alpha \vdash \lambda x : \alpha.\ x : \alpha \to \alpha}}{\Gamma \vdash \Lambda\alpha.\ \lambda x : \alpha.\ x : \forall \alpha.\ \alpha \to \alpha}$$

Figure 4: The identity function.

We will use the meta-symbol **id** to refer to the identity function constructed in Figure 4 to keep it concise.

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \mathbf{id} : \forall \alpha.\alpha \to \alpha}}{\Gamma \vdash \mathbf{id}[A] : A \to A} \quad \dfrac{y : A \in \Gamma}{\Gamma \vdash y : A}}{\Gamma \vdash \mathbf{id}[A]\ y : A}$$

Figure 5: Applying the identity function to the variable y with type A.

2.1.3. Linear types

The core idea of a linear type system is that variables must be used *exactly once*. This means the typing relation $\Gamma \vdash e : \sigma$ no longer only requires that the set of variables in e are a subset of Γ, but rather that the set of variables in e is Γ. This means the typing rules App and Var in Figure 2 are no longer valid.

The typing rules for a linear type system are shown in Figure 6. Note how the environments for e_1 and e_2 in App are disjoint, i.e. Γ and Δ must not share any variables. Similarly, the rule for Var, differs from its simply typed variant, which now requires that the environment contains only the variable $x : A$. The arrow $\multimap$ is used instead of $\to$ to denote linearity.

$$\dfrac{\Gamma \vdash e_1 : \sigma \multimap \tau \quad \Delta \vdash e_2 : \sigma}{\Gamma, \Delta \vdash e_1 e_2 : \tau}\text{App} \qquad \dfrac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \lambda x.e : \sigma \multimap \tau}\text{Abs}$$

$$\dfrac{}{\cdot, x : \sigma \vdash x : \sigma}\text{Var}$$

Figure 6: Typing rules for App, Abs, and Var in a linear type system.

How would we derive terms that use a variable twice, or perhaps a term that does not use a variable? Linear logic, and in turn linear types, solves this using *exponentials*. Exponentials introduce an explicit way to duplicate and discard variables. The rules for exponentials are shown in Figure 7.

$$\dfrac{\Gamma, x : A \ \vdash e : B}{\Gamma, x : !A \ \vdash e : B}\text{Derelict} \qquad \dfrac{\Gamma \ \vdash e : B}{\Gamma, x : !A \ \vdash e : B}\text{Discard}$$

$$\dfrac{\Gamma, x : !A, y : !A \ \vdash e : B}{\Gamma, x : !A \ \vdash e : B}\text{Duplicate} \qquad \dfrac{!\Gamma \ \vdash e : B}{!\Gamma \ \vdash e : !B}\text{Promote}$$

Figure 7: Context and term rules for exponentials.

The function !_ : Environment $\to$ Environment, called "bang", is defined by:

$$!(\Gamma, x : \sigma) = (!\Gamma, x : !\sigma)$$
$$!(\cdot) = \cdot$$

Note that "bang" is both a type constructor ($!A$) and a function on environments ($!\Gamma$). Because Derelict, Discard, and Duplicate manipulate the left-side of the turnstyle, they are read bottom-to-top. The Promote rule manipulates the right-side of the turnstyle, and is read top-to-bottom. Syntactically we leave the terms for derelict, discard, and duplicate silent, i.e. no explicit syntax is used to derelict, discard, and duplicate variables. Now we can create a derivation for the term $\lambda x.\lambda y.y : !\tau \multimap \sigma \multimap \sigma$ that discards the variable x. The derivation is show in Figure 8.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\cdot, y : A \ \vdash y : A}\text{Var}}{\cdot \ \vdash \lambda y.y : A \multimap A}\text{Abs}}{\cdot, x : !B \ \vdash \lambda y.y : A \multimap A}\text{Discard}}{\cdot \ \vdash \lambda x.\lambda y.y : !B \multimap A \multimap A}\text{Abs}$$

Figure 8: Derivation of a linearly typed term that discards the variable x.

Linear types does not entirely prohibit the duplication and discarding of variables, but rather make it explicit.

2.2. Continuation-passing Style

Continuation-passing style (CPS) is a programming style where control is passed explicitly via continuation functions rather than returning values. Intuitively, we can think of continuations as functions that capture “the rest of the program”.

Continuations have been successfully used in many compilers for strict languages, for example by: Appel (Appel, 2007), Fradet and Le Métayer (Fradet & Le Métayer, 1991), and Kelsey and Hudak (Kelsey & Hudak, 1989). Additionally, in the Spineless Tagless G-Machine, the abstract machine of the Glasgow Haskell Compiler (GHC), continuations are used to manage thunks, function application, and case analysis.

Continuation-passing style is easiest explained by example. The identity function written in normal style would be:

$$\begin{aligned}&\text{id} : \forall a.a \to a\\&\text{id} = \lambda x.\ x\end{aligned}$$

Contrast it to the CPS version, where we use $a \to \bot$ to denote a function that takes a as argument and terminates with no value.

$$\begin{aligned}&\text{id} : \forall a.\ a \to (a \to \bot) \to \bot\\&\text{id} = \lambda x.\lambda k.\ k(n)\end{aligned}$$

A natural question that comes to mind is why we want continuation-passing style? An immediate benefit of CPS is that every function call is a tail call, which ensures that tail call optimization (see Section 4.3) is always possible. Another benefit of CPS is that the order of evaluation is made explicit by the syntax. If we consider the following program written in normal style:

$$\begin{aligned}&\text{let } y = \text{bar}(x) \text{ in}\\&\text{let } z = \text{baz}(x) \text{ in}\\&y + z\end{aligned}$$

Is y evaluated first or is z evaluated first? The choice would be up to the specification. If we take at the same function in CPS we will see that the evaluation order is determined by the order of the function calls.

1. $\lambda k.\ \text{bar}(x, \lambda y.\ \text{foo}(x, \lambda z.k(y + z)))$
2. $\lambda k.\ \text{foo}(x, \lambda z.\ \text{bar}(x, \lambda y.k(y + z)))$

In (1), bar is called first in the closure binding k, and all other function calls are in subsequent closures. Only a single function call can be made per closure. This can been seen in the grammar of commands in Lithium (see Section 3.1).

2.3. Compilation Targets

When you are compiling your language you have to pick a target to compile it to. Unless you are directly targeting machine code, most of the time you want a higher level compilation target.

There are a lot of different choices for this task, some commons examples are: LLVM IR (Lattner & Adve, 2004), Cranelift (*Cranelift*, n.d.) and GNU's GIMPLE (*Gimple*, n.d.). These languages are what is known as intermediate representations (IR), and they are all targeted by different compilers. They remove the need for the compilers to directly target CPU specific machine code or assembly. Most of the time these IRs are also cross platform, making portability easier to achieve. Because the intermediate representations are higher level than assembly languages are, they trade-off explicit control in favor of abstractions.

Depending on the source language you are compiling, an IR is not necessarily the most fitting option. Many IRs are modeled for procedural languages, and expect the source language to follow a traditional stack frame based calling convention, which might not always be desired. If this is the case, targeting an assembly language directly can prove more advantageous.

An assembly language is often as low as you can go without directly targeting machine code. They are usually made to resemble machine code in text form, and are tailored after CPU specific instruction sets. Personal computers and servers commonly use the instruction set x86-64, which is an extension that was created in 2000 based on the already popular instruction set x86 (Place & Cleveland, n.d.).

When you directly target assembly languages, portability suffers. You not only have to target different assembly languages for different CPU architectures, you will also have to cater to the operating system you are targeting. For instance, on a unix-like operating system, you can almost always rely on an implementation of the C Standard Library (libc), or system calls if direct communication with the operating system is needed. This does not apply to operating systems such as Windows however, where you instead have to depend on the provided system libraries to interact with the rest of the system.

On Windows libc availability is not a guarantee, and because they do not have a stable API, it is *strongly* recommended to avoid using system calls directly. Due to reasons such as this, the simple act of printing to stdout may look wildly different depending on the operating system, even though they may use the same assembly language. This is something you have to consider with most IRs as well, but it can be alleviated with sufficient abstractions.

3. Lithium

The intended use of Lithium is as a compiler intermediate representation for functional languages, similar to that of GHC Core (Jones et al., 1993). Lithium differs from GHC Core and most functional language intermediate representations in that it prioritizes finer control over resources. This is achieved by departing from the lambda calculus and its natural deduction root, rather taking inspiration from linear types and intuitionistic linear logic (Lafont, 1988).

3.1. Grammar

Before going into details on Lithium it can be helpful to get an overview of how the language looks. The grammar of Lithium is depicted in Figure 9.

Value

$v, v' \ :=\ x \mid () \mid \text{newstack} \mid \lambda p.\ c \mid \mathit{inl}\ v \mid \mathit{inr}\ v \mid \square v \mid (v, v') \mid (@t, v)$

Command

$c, c' \ :=\ z(v) \mid \text{case } v \text{ of } \{\text{inl } x \to c; \text{inr } y \to c'\} \mid \text{let } p = v; c$

Pattern

$p \ :=\ () \mid x \mid @t, y \mid p, p' \mid \square p$

Type

$t, t' \ :=\ \mathbf{1} \mid \mathbf{0} \mid \bigcirc \mid x \mid \neg t \mid *\, t \mid \sim t \mid \square t \mid t \otimes t' \mid t \oplus t' \mid \exists x.t$

Definition

$d \ :=\ x : t = v$

Module

$m \ :=\ \varepsilon \mid d; m$

Figure 9: Grammar of Lithium.

Take this top-level function (*Definition*) as an example:

$$\begin{gathered}\text{swap} : *\,((B \otimes A) \otimes \sim (A \otimes B)) \\ = \lambda((x, y), k) \to k((y, x));\end{gathered}$$

swap here can be broken up into three parts:

- The name: swap
- The type: $*\,((B \otimes A) \otimes \sim (A \otimes B))$

- The value: $\lambda((x, y), k) \to k((y, x))$

swap takes two arguments: a tuple $(B \otimes A)$ and a continuation function $\sim (A \otimes B)$. The first argument, the tuple, is pattern matched on, exposing the variables x and y. The second argument, k, is the continuation function.

A module consists of a list of definitions, where a definition is a top-level function. A definition consists of a name, a type, and a value. The distinction between values and commands is the most interesting aspect. Commands come into play in the bodies of lambdas. Commands consist of let-bindings, case-expressions, or function calls. Note that the only way to terminate a sequence of commands is by a function call $z(v)$. This means Lithium programs are written in continuation-passing style.

3.2. Kinds & types

Lithium is based on a variant of polarised linear logic (Laurent, 2002). It is essentially Lafont's linear logic (Lafont, 1988), where $A \multimap B$ is replaced by $\neg A$. Intuitively, we can think of $\neg A$ as $A \multimap \bot$, or from a programmer's perspective: a function that takes A as argument and terminates with no value, like in Section 2.2.

There are four new constructs in Lithium that extend linear logic. These are: *empty stack* ($\bigcirc$), *linear pointer* ($\square$), *stack closure* ($\sim$), and *static function* ($*$). The latter two are variants of negation ($\neg$).

At the core of Lithium is the kind system. Where values have types, types have kinds. The two kinds in Lithium are *stack* (ω) and *known size.* ω represents a region of memory of unknown size, with extra space reserved to store known sized data.

$$\frac{}{\bigcirc : \omega}\ \textit{empty stack} \qquad \frac{}{\mathbf{0} : \omega}\ \textit{empty} \qquad \frac{}{\mathbf{0} : \text{❶}}\ \textit{empty}$$

$$\frac{}{\mathbf{1} : \text{❶}}\ \textit{unit} \qquad \frac{A : \text{❶} \quad B : \omega}{A \otimes B : \omega}\ \textit{product} \qquad \frac{A : \text{❶} \quad B : \text{❶}}{A \otimes B : \text{❶}}\ \textit{product}$$

$$\frac{A : \omega \quad B : \omega}{A \oplus B : \omega}\ \textit{sum} \qquad \frac{A : \text{❶} \quad B : \text{❶}}{A \oplus B : \text{❶}}\ \textit{sum} \qquad \frac{A : \text{❶}}{\sim A : \omega}\ \textit{stack closure}$$

$$\frac{A : \omega}{* A : \text{❶}}\ \textit{static function} \qquad \frac{A : \text{❶}}{\neg A : \text{❶}}\ \textit{linear closure} \qquad \frac{A : \omega}{\square A : \text{❶}}\ \textit{linear pointer}$$

$$\frac{A : \text{❶}}{\exists \alpha . A : \text{❶}}\ \exists\ \textit{intro} \qquad \frac{A : \omega}{\exists \alpha . A : \omega}\ \exists\ \textit{intro} \qquad \frac{}{\alpha : \omega}\ \textit{type var}$$

Figure 10: Kinding rules in Lithium.

The kinding rules in Figure 10 are mostly self-descriptive, but some things to keep in mind for the rules are:

- There is no sub-kinding; if a type with kind ω is expected, then a type with kind ❶ is not allowed, and vice versa.
- It is forbidden to construct a pair of two stacks
- The kinds in a sum type must match
- Type variables are always stacks, which means they cannot be used directly for Haskell-style polymorphism (see Section 4.1.3 why type variables must have kind ω).

Each negation enables one of three programming styles: goto ($*$) , procedural ($\sim$), and higher-order ($\neg$).

The goto style is the most primitive. It can be considered as a one-way transfer of control. Consider the function $f : *\,(A \otimes *\,B \otimes \bigcirc)$. From f we can call the continuation $*\,B$, which is just a static function pointer, and because it is only a static function pointer, it can not capture any state. The state that $*\,B$ manipulates is exactly the stack B, and it is passed by f.

The second style, procedural, enables exactly what its name suggests: procedures. The type signature $f : *\,(A \otimes \sim B)$ exactly corresponds to the C function signature $B\ f(A\ a)$. The type $\sim B$ corresponds to a stack that accepts B as a return value to continue with. This stack can store an arbitrary state of kind ω. Because of the kinding rules of $*$ and $\otimes$, only a single stack can be passed to a static function.

Finally we have higher-order programming, which is not possible with $*$ and $\sim$ alone. The type $*\,(A \otimes \sim B \otimes \sim C)$ is ill-kinded, and $*\,(A \otimes *\,B \otimes \sim C)$ would not work either because $*\,B$ can not capture state. To enable higher-order programming we introduce the *linear closure*. The linear closure can capture arbitrary state and produces a type with a known size. Now we can write the higher-order function: $*\,(A \otimes \neg B \otimes \sim C)$. In Section 4.1 we explain how closures are transformed to static functions and explicit stack environments.

3.3. Types & values

Lithium programs consist of two syntactic fragments:

- The *positive fragment* describes how values are created. When we talk about values we refer to the positive fragment.
- The *negative fragment*: describes how to consume environments of values. The negative fragment can also be referred to as commands.

The typing rules for values and commands are given the form $\Gamma \ \vdash v : A$ and $\Gamma \ \vdash c$, respectively. Values have a right-side type $(v : A)$ to symbolize construction. Conversely, commands do not have a right-side type, to symbolize consumption. Because the rules for values have a right-side type, they are read top-to-bottom. The rules for commands are read bottom-to-top.

Kinds are also introduced to the environment Γ. Figure 11 shows the rules for the environment.

$$\frac{}{\cdot : \text{❶}} \qquad \frac{\Gamma : \text{❶} \quad A : \omega}{(\Gamma, x : A) : \omega} \qquad \frac{\Gamma : \omega \quad A : \text{❶}}{(\Gamma, x : A) : \omega} \qquad \frac{\Gamma : \text{❶} \quad A : \text{❶}}{(\Gamma, x : A) : \text{❶}}$$

Figure 11: Kinding rules for environments.

The empty environment is of known size. If the environment is of known size, then extending it with a stack makes it have unknown size. The last two rules state that we are allowed to extend any environment with things of known size. The kinding rules allow at most one stack in the environment. This will be critical when we perform closure conversion in Section 4.1. If we omit the kind from the environment, then either kind is allowed.

The typing rules for the positive fragment are depicted in Figure 12, and Figure 13 shows the typing rules for the negative fragment.

$\dfrac{}{\cdot \ \vdash \text{newstack} : \bigcirc}$	*newstack*	Newstack is a primitive for creating an empty stack
$\dfrac{}{\cdot, x : A \ \vdash x : A}$	*var*	All variables must be used exactly once.
$\dfrac{}{\cdot \ \vdash () : \mathbf{1}}$	*unit*	The unit value. The environment must be empty
$\dfrac{\Gamma \ \vdash t : A \quad \Delta \ \vdash u : B}{\Gamma, \Delta \ \vdash (t, u) : A \otimes B}$	*pair*	Constructing a pair from the two values u and v. Note that the contexts Γ, Δ, must be disjoint
$\dfrac{\Gamma \ \vdash t : A}{\Gamma \ \vdash \text{inl}\ t : A \oplus B}$	*inj left*	Constructing the left value of a sum type

$\dfrac{\Gamma \ \vdash t : B}{\Gamma \ \vdash \text{inr } t : A \oplus B}$	*inj right*	Constructing the right value of a sum type
$\dfrac{\Gamma \ \vdash t : A}{\Gamma \ \vdash \Box t : \Box A}$	*linear pointer*	Making an indirection to a stack
$\dfrac{\Gamma, \alpha \ \vdash t : A}{\Gamma \ \vdash \langle A, t \rangle : \exists \alpha . A}$	*∃ intro*	Existentially quantifying the term $t : A$ with the type variable α
$\dfrac{\cdot, x : A \ \vdash c}{\cdot \ \vdash \lambda^{*} x.c : * A}$	*static function*	Create a static funcion. The environment must be empty, which means the static function can not capture any free variables.
$\dfrac{(\Gamma, x : A) : \omega \ \vdash c}{\Gamma \ \vdash \lambda^{\sim} x.c :\sim A}$	*stack closure*	Create a stack closure, which can capture free variables. The environment must be a stack.
$\dfrac{(\Gamma, x : A) : ❶ \ \vdash c}{\Gamma \ \vdash \lambda^{\neg} x.c : \neg A}$	*linear closure*	Create a linear closure, which can capture free variables, The environment must not contain a stack.

Figure 12: Typing rules for the positive fragment of Lithium.

$\dfrac{\Gamma \ \vdash c}{\Gamma, z : \bigcirc \ \vdash \text{freestack } z; c}$	*freestack*	Free the stack z. The variable z is removed from the environment.
$\dfrac{\Gamma \ \vdash c}{\Gamma, z : \mathbf{1} \ \vdash \text{let } () = z; c}$	*discard*	Discard the unit value. The variable z is removed from the environment.
$\dfrac{\Gamma, a : A, b : B \ \vdash c}{\Gamma, z : A \otimes B \ \vdash \text{let } a, b = z; c}$	*pair*	Destruct the pair z, introducing the variables a and b, and remove z from the environment.

$\dfrac{\Gamma, x_i : A_i \ \vdash c_i}{\Gamma, z : A_1 \oplus A_2 \ \vdash \text{case } z \text{ of inj}_i x_i \mapsto c_i}$	*case*	Pattern match on the sum, binding the value of z to x_i and continue with the continuation c_i. The subscript i is used to indicate that there are two possible injections.
$\dfrac{\Gamma, x : A \ \vdash c}{\Gamma, z : \Box A \ \vdash \text{let } \Box x = z; c}$	*follow*	Follow the indirection, binding the stack behind the indirection to x
$\dfrac{\Gamma, \alpha, x : A \ \vdash c}{\Gamma, z : \exists \alpha.A \ \vdash \text{let } \langle \alpha, x \rangle = z; c}$	$\exists$ *elim*	Match the existentially quantified variable z to access the value x
$\dfrac{\Gamma \ \vdash t : A}{\Gamma, z : *\,A \ \vdash \text{call}^* z(t)}$	*call**	Call the static function z with the value t as argument. Note that the environment does not need to be empty when calling static functions
$\dfrac{\Gamma \ \vdash t : A}{\Gamma, z :\sim A \ \vdash \text{call}^\sim z(t)}$	*call*$^\sim$	Call the stack closure z with the value t as argument
$\dfrac{\Gamma \ \vdash t : A}{\Gamma, z : \neg A \ \vdash \text{call}^\neg z(t)}$	*call*$^\neg$	Call the linear closure z with the value t as argument

Figure 13: Typing rules for the negative fragment of Lithium.

The juxtaposition of the negative and positive fragments create an elegant picture. For every value v, a corresponding command exists for how to destruct an environment of v. Variables are not explicitly destructed; they are consumed on use.

In Figure 14 we show how we can use the aforementioned rules to give the typing proof for the identity function specialised to the type $A :$ ❶ in Lithium.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\cdot, t : A \ \vdash t : A}\,\textit{var}}{\cdot, t : A, z :\sim A \ \vdash \text{call}^\sim z(t)}\,\textit{call}^\sim}{\cdot, x : (A \otimes \sim A) \ \vdash \text{let } t, z = x; \text{call}^\sim z(t)}\,\textit{pair}}{\cdot \ \vdash \lambda^* x.\ \text{let } t, z = x; \text{call}^\sim z(t) : *\,(A \otimes \sim A)}\,\textit{static function}$$

Figure 14: The typing proof for the identity function specialised to A in Lithium.

In Figure 15 we derive the proof for the type, to ensure that the type is kind correct.

$$\cfrac{\cfrac{A : \text{❶} \qquad \cfrac{A : \text{❶}}{\sim A : \omega}}{(A \otimes \sim A) : \omega}}{* \, (A \otimes \sim A) : \text{❶}}$$

Figure 15: The kind proof for the type of the identity function.

The kinding rules and typing rules provide a structured way of constructing correct programs. When creating the compiler for Lithium we can leverage these rules to construct the kindchecker and typechecker.

4. Compiling Lithium

In this chapter we describe how Lithium can be translated to something that can be represented in an assembly language. We continue by giving a compilation scheme for the language, and describe how the output of the scheme translates to assembly. We then give a mapping of types to memory, and specify the application binary Interface (ABI).

4.1. Transformations

At this stage Lithium is still a calculus. How do we bridge the gap between calculus and machine? This section goes into the necessary transformations to turn Lithium into a language that can be transformed to an assembly language.

The first three phases of the Lithium compiler are: linear closure conversion, stack selection, and pointer closure conversion. The first step eliminates linear closures, the second step ensures that each closure contains at most one stack to execute on, and the third transformation, pointer closure conversion, replaces each stack closure by an explicit pair of static function and environment.

4.1.1. Linear closure conversion

It is critical for first-order programs to identify the call stack, i.e. where a procedure should return control when finishing execution. The first step in this process is making pointers to stacks explicit. We do this by transforming types and closure values in the following manner:

Source	Target
$\neg A$	$\square \sim A$
$\lambda^{\neg} x.c$	$\square\lambda^{\sim} x.c$

It is important not to forget the negative fragment as well. Before calling a function with type $\neg A$, which after conversion has type $\square \sim A$, we have to follow the indirection to access the closure.

Source	Target
$f(x)$	let $\square g = f; g(x)$

Because the type $\neg A$ is transformed to $\Box \sim A$, the type checker should allow $\Box \sim A$ where $\neg A$ is expected.

In the typing rules in Section 3.3, we have seen that linear closures require the environment Γ to have kind ❶, which means Γ does not contain a stack. However, because linear closures are transformed to stack closures behind linear pointers, the environment for stack closures end up being ill-kinded. In the next section (Section 4.1.2) we present a transformation that corrects this.

4.1.2. Stack Selection

It is important for every stack closure ($\sim A$) to identify a single unique stack that it can execute on. The stack selection phase selects a single unique stack for every closure if at least one stack exists, ensuring that every closure has *at most* one stack prepared. The reason we can not guarantee that there is *exactly one* stack prepared is because stacks have not been made explicit yet. In Section 4.1.3 we will show the necessary transformations to make stacks explicit, and how to introduce new stacks.

Consider the following program:

$$\lambda(f, k).\ k(\lambda y.\ f(y)) : *\,(\neg A \otimes \sim \neg A)$$

After making the pointers to stacks explicit we end up with the following program:

$$\lambda(f, k).\ k(\Box\lambda y.\ \text{let } \Box f' = f;\ f'(y)) : *\,(\Box \sim A \otimes \sim (\Box \sim A))$$

Because k has type $\sim (\Box \sim A)$, its environment must be a stack. The issue is that the only variable that is a stack is f', but it cannot be the chosen stack because bound variables are stored on the stack. The chosen stack must be a variable that is bound outside the closure, or an explicit newstack.

Stack selection moves the let $\Box f' = f$ out of the closure, making f' free in the closure, and selecting it as the stack.

The resulting program would end up being:

$$\lambda(f, k).\ \ \text{let } \Box f' = f;\ k(\lambda y.\ f'(y)) : *\,(\Box \sim A \otimes \sim (\Box \sim A))$$

Now $\lambda y.f'(y)$: contains exactly the stack f'.

4.1.3. Pointer Closure Conversion

The goal of the pointer closure conversion is to make the structure of stacks explicit, replacing a stack closure by an explicit pair of a static function pointer and an environment. At the assembly level the concept of procedures and closures do not exist, there are only jumps (gotos) and labels.

The representation for $* A$ is straightforward; it is a label. Calling a function of type $* A$ corresponds to jumping to the label. Because labels and jumps are the only thing available to us at the assembly level, we need to transform $\sim$ to $*$, and we need to make the closure explicit.

The pointer closure conversion phase transforms $\sim A$ to $\exists\gamma. * (A \otimes \gamma) \otimes \gamma$, eliminating both procedures and closures. The existential quantification is there because the structure of the environment is unknown for the callee. Now we can see why type variables must have kind ω; if they had kind ❶, then $* (A \otimes \gamma)$ would be ill-kinded, and we would not be able to represent the environment in the type.

Values and commands need to be transformed as well to ensure that they match the types. Stack closures ($\lambda^{\sim}$) are transformed in the following manner:

Source	Target
$\sim A$	$\exists\gamma. * (A \otimes \gamma) \otimes \gamma$
$\lambda^{\sim}.c$	$\langle \bigotimes \Gamma, ((\lambda^{*}(x, \rho).\ \text{unpairAll}(\rho); c), \text{pairvars}(\Gamma))\rangle$

Γ represents the free variables in the closure. $\bigotimes \Gamma$ is short for $A_1 \otimes A_2 \otimes \ldots \otimes A_n$ If Γ has kind ❶, then pairsvars must construct a newstack ($\bigcirc$). For example: if $\Gamma = \cdot, x : A :$ ❶$, y : B :$ ❶, then the output of pairsvars will be $(x, y, \mathit{newstack})$. unpairAll is a macro that inverts this procedure.

Because the closures are converted, the corresponding commands must also be transformed to match. Fortunately, the transformation is straightforward:

Source	Target
$z(a)$	$\text{let } \langle \alpha, z_1 \rangle = z; \text{let } z_2, \rho = z_1; z_2(a, \rho)$

The conversion can be easier to understand given an example. We will show two examples: one for each kind of the environment.

Take the resulting program from Section 4.1.2.

$$\lambda(f, k).\ \ \text{let } \Box f' = f;\ \ k(\lambda y.\ f'(y)) : * (\Box \sim A \otimes \sim (\Box \sim A))$$

Because f' is a stack, and a free variable in $\lambda y.f'(y)$, pairvars does not need to construct a newstack. Transforming the program would yield the following:

$$\begin{aligned}
\lambda^*(f,k).\ &\text{let } \Box f' = f;\\
&\text{let } \langle \alpha, k' \rangle = k;\\
&\text{let } g, \rho_1 = k';\\
&g(\Box\langle @\exists\gamma. *(A \otimes \gamma) \otimes \gamma, (\lambda^*(y, \rho_2).\ \text{let } \langle \beta, x\rangle = \rho_2;\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{let } h, \rho_4 = x;\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad h(y, \rho_4), f')\rangle, \rho_1)
\end{aligned}$$

Because k has type $\exists\gamma. *(A \otimes \gamma) \otimes \gamma$ after closure conversion, the second, and third rows are necessary to access the static function $g : *(A \otimes \gamma)$. The same process is repeated inside the argument of g. Also, note how f' is the environment inside g now.

Let us now consider an example where a stack closure does not contain a stack in the environment. Assume that the static function foo : $*\,A$ exists.

$$(\lambda^{\sim} x.\ \text{foo}(x)) : \ \sim A$$

There is no stack in the environment of the closure; the conversion phase needs to insert an explicit newstack to ensure that there is a stack to execute on. After conversion this closure would end up being:

$$\langle \bigcirc, (\lambda^*(x, \rho_1).\ \text{freestack } \rho_1; \text{foo}(x), \textit{newstack})\rangle$$

Now the environment is a new empty stack, and we free it before calling the static function foo()

4.2. Compilation Scheme

As can be seen in Section 3.3, every aspect of Lithium is based on a set of rules, split into positive and negative fragments.

These rules can by design be translated to x86-64 assembly in a straight-forward manner. They are first translated into "pseudo" instructions which can then be translated into x86-64.

The compilation scheme consists of three functions:

- $^{+}[\![\_]\!]^{\text{❶}}_{\rho}$: Value → Pseudo instruction
- $^{+}[\![\_]\!]^{\omega}_{\rho}$: Value → Pseudo instruction
- $^{-}[\![\_]\!]_{\rho}$: Command → Pseudo instruction

We prefix the functions with $^{+}$ and $^{-}$ to refer to the respective fragments. If we use α in-place of ω and ❶, then the definition exists for both kinds. The function ρ is a mapping from variables to a list of pseudo registers. The syntax $\rho, (x \mapsto s_n)$ means the context ρ is extended with x mapping to the list s_n. If we instead write $\rho, (x \mapsto [r_0, ..., r_n])$ then x maps to the list containing $r_0, ..., r_n$. Additionally, we will use $r_0 : rs$ to mean the non-empty list with r_0 as the head and rs as the tail, and $s_0 + +s_1$ means the concatenation of the lists s_0 and s_1. Lastly, the number of registers ρ maps the variable x to must be exactly $[\![\Gamma(x)]\!]^{\mathrm{R}}$, i.e $|\rho(x)| = [\![\Gamma(x)]\!]^{\mathrm{R}}$.

The function $[\![A]\!]^{\mathrm{R}}$ is a mapping from a type A to the number of registers needed to store a value v with type A. The definition for it can be found in Section 4.4.2.

We annotate a variable z^{ω} or $z^{\text{❶}}$ in the negative fragment to indicate the kind of the type of the variable.

A pseudo register is a physical register, or a location on the stack. Formally, ρ can be seen as a function $\rho : \Gamma \to \mathsf{List}(\mathrm{Reg})$. The range of ρ is a list of pseudo registers because not all values can be stored in one physical register.

In the compilation scheme, two explicit physical registers are used: the stack pointer (SP) and the stack save pointer (SSP). SP points to a stack on which we can pop and push, and SSP is used to temporarily back up SP.

The scheme also contains the meta instruction: "let $r = \mathrm{next}(\rho, t)$", where next has the type $\mathsf{List}(\mathrm{Reg}) \to \mathsf{Type} \to \mathsf{List}(\mathrm{Reg})$, which allocates the list r of fresh pseudo-registers. The pseudo-registers chosen depends on which pseudo registers are used in ρ, and the size of the type t. The meta instruction exists only at compile time.

To ensure correctness and consistency of the compilation scheme, we specify pre- and post-conditions for each compilation function:

Before calling $^{+}[\![v]\!]^{\omega}$, SP can be used freely. After the call, SP points to v. $^{+}[\![v]\!]^{\text{❶}}$ (note the change in kind) requires that SP points to a valid stack before being called. After being called, v is pushed on the stack pointed to by SP. Finally, we have $^{-}[\![v]\!]$. It has no additional pre-conditions, only the post-condition that the program is terminated. All three functions require that $\rho(x)$ is correctly loaded with values.

The translation between pseudo instructions and x86-64 assembly can be seen in Figure 16, and in Figure 17 we explain the operands used in the compilation scheme.

Pseudo instruction	x86-64 instructions

$push_R(\mathrm{OP}_n)$	`subq $n, R`
	`movq OP_1, 0(R)`
	`...`
	`movq OP_n, n(R)`

$\mathrm{OP1} = \mathrm{OP2}$	`movq OP2, OP1`

$pop_R(\mathrm{OP}_n)$	`movq n(R), OP_n`
	`...`
	`movq 0(R), OP_1`
	`addq $n, R`

`if izero(OP_1)`
`  then C1`
`  else C2`
	`movq OP_1, %R10`
	`cmp $0, %R10`
	`jnz lbl`
	`C1 # codeblock`
	`lbl:`
	`C2 # codeblock`

$jmp\ L$	`jmp L`
$L:$	`L: # a label`

$\mathrm{OP1}_1 \leftarrow \mathrm{malloc}(\mathrm{OP2}_1)$	`movq OP2_1, %RDI`
	`call malloc`
	`movqq %RAX, OP1_1`

$\mathrm{free}(\mathrm{OP}_1)$	`movq $0, %RAX`
	`movq OP_1, %RDI`
	`call free`

Figure 16: Translations between pseudo and x86-64 instructions and operands. Observe that the x86-64 instructions may differ in the compiler due to optimizations or the memory size of a variable.

Operand	x86-64 Operand
Numerical literal, e.g. `42`	Numerical literal prefixed with `$`, e.g. `$42`
$\rho(x)$	Appropriate list of pseudo registers for type of x
`SP`	`%R15`
`SSP`	`%R14`
`[VAL]`	`0(VAL)`

Figure 17: Translations between pseudo and x86 operands. `%R14` and `%R15` are physical x86-64 registers, and `0(VAL)` means that we are interacting with the memory address stored in `VAL`.

Stack tuple	Compile v_2 first to ensure that SP points to a valid stack, then compile v_1.	$^{+}[\![(v_1, v_2)]\!]^{\omega}_{\rho,\sigma} =$ $[\![v_2]\!]^{\omega}_{\rho}$ $[\![v_1]\!]^{\text{❶}}_{\sigma}$
Tuple	Compile v_2 first, to ensure conformity with the stack tuple.	$^{+}[\![(v_1, v_2)]\!]^{\text{❶}}_{\rho,\sigma} =$ $[\![v_2]\!]^{\text{❶}}_{\rho}$ $[\![v_1]\!]^{\text{❶}}_{\sigma}$
Existential introduction	Compile the value v_1. Because types do not exist at runtime it is just a recursive call.	$^{+}[\![(@t, v_1)]\!]^{\alpha}_{\rho} =$ $[\![v_1]\!]^{\alpha}_{\rho}$
Indirection to a stack	Create space on the stack for the stack pointer to v_1. Backup SSP and SP. Compile v_1, setting SP to the stack v_1. Write SP to the space created, then restore SP and SSP to their previous states.	$^{+}[\![\square v_1]\!]^{\text{❶}}_{\rho} =$ $sp = sp + 1$ $push_{sp}(ssp)$ $ssp = sp$ $[\![v_1]\!]^{\omega}_{\rho}$ $[ssp - 1] = sp$ $sp = ssp - 1$ $ssp = [ssp]$
Right injection	Compile v_1 and push the tag 0 on the stack. The tag must be pushed after the compilation of v_1 because when $\alpha = \omega$, SP might not be a valid stack.	$^{+}[\![\text{inl } v_1]\!]^{\alpha}_{\rho} =$ $[\![v_1]\!]^{\alpha}_{\rho}$ $push_{sp}(0)$
Left injection	Compile v_1 and push the tag 1 on the stack.	$^{+}[\![\text{inr } v_1]\!]^{\alpha}_{\rho} =$ $[\![v_1]\!]^{\alpha}_{\rho}$ $push_{sp}(1)$
Stack variable	Set SP to r_0, essentially switching stack.	$^{+}[\![x]\!]^{\omega}_{x \mapsto [r_0]} =$ $sp = r_0$

Variable	$^{+}[\![x]\!]^{\text{❶}}_{x\mapsto s_n} = \boxed{push_{sp}(s_n)}$
Push the variable on the stack.	
Unit	$^{+}[\![()]\!]^{\text{❶}}_{[]} = \boxed{}$
() does not exist at runtime.	
Static function	$^{+}[\![\lambda^{*}x.c]\!]^{\text{❶}}_{[]} = l_1\colon \boxed{\begin{array}{l} \text{"let } r = \text{next}([], ptr)\text{"} \\ r = sp \\ ^{-}[\![\text{c}]\!]_{x\mapsto r} \end{array}}$ $\boxed{push_{sp}(l_1)}$
Generate a unique label l_1. Under l_1, let r_1 be the next available pseudo register, then set it to sp and compile the command c. Finally, push l_1 on the stack. l_1 and the block can be thought of as creating a procedure in a C-like language.	
Newstack	$^{+}[\![newstack]\!]^{\omega}_{[]} = \boxed{\begin{array}{l} r_1 \leftarrow \text{malloc}(S) \\ sp = r_1 \end{array}}$
Allocates a new stack and switches to it. S is the size of the stack to be allocated. Some implementation specific details are omitted. Section 4.4.3 goes into more detail how a stack is allocated in Lithium.	

Figure 18: Compilation scheme for the positive fragment of Lithium.

Pop top of stack
$^{-}[\![\text{let } x, y = z^{\omega} : A \otimes B; \text{c}]\!]_{\rho, z\mapsto[r_0]} = \boxed{\begin{array}{l} \text{"let } r_1 = \text{next}(\rho, A)\text{"} \\ pop_{r_0}(r_1) \\ [\![\text{c}]\!]^{\omega}_{\rho, x\mapsto r_1, y\mapsto[r_0]} \end{array}}$
Pops the top value of the stack z and stores it in r_1. y is the rest of the stack.

Destruct tuple

$${}^{-}\llbracket \text{let } x, y = z^{\text{❶}} : A \otimes B; c \rrbracket_{\rho, z \mapsto s_0 +\!+ s_1} = \llbracket \text{c} \rrbracket^{\text{❶}}_{\rho, x \mapsto s_0, y \mapsto s_1}$$

such that: $|s_0| = \text{sizeof}(A); |s_1| = \text{sizeof}(B)$

Split the environment into two disjoint lists of pseudo registers. The environment x is the environment s_0 and y is s_1. The invariant states that the size of s_0 and s_1 are determined by the size of the types A and B

Unit elimination

$${}^{-}\llbracket \text{let } () = z^{\text{❶}}; c \rrbracket_{\rho, z \mapsto []} = \llbracket \text{c} \rrbracket_{\rho}$$

Because the unit value does not exist at runtime, the matching is a no-op.

Existential elimination

$${}^{-}\llbracket \text{let } @t, x = z^{\alpha}; c \rrbracket_{\rho, z \mapsto s_n} = \llbracket \text{c} \rrbracket_{\rho, x \mapsto s_n}$$

Types have no runtime representation.

Following an indirection

$${}^{-}\llbracket \text{let } \square x = z^{\text{❶}}; c \rrbracket_{\rho, z \mapsto [r_0]} = \llbracket \text{c} \rrbracket_{\rho, x \mapsto [r_0]}$$

Because r_0 is the pointer to the stack already, we just need to update the environment.

Stack deallocation

$${}^{-}\llbracket \text{freestack } z^{\omega}; c \rrbracket_{\rho, z \mapsto [r_0]} = \begin{array}{l} \text{free}(r_0) \\ \llbracket \text{c} \rrbracket_{\rho} \end{array}$$

Deallocate the stack r_0.

Case expression with variable which is a stack

$${}^{-}\llbracket \begin{array}{l} \text{case } z^{\omega} \text{ of } \{ \\ \qquad \text{inl } x \mapsto c_1; \\ \qquad \text{inr } y \mapsto c_2; \} \end{array} \rrbracket_{\rho, z \mapsto [r_0]} = \begin{array}{l} \text{"let } r_1 = \text{next}(\rho, \text{int})\text{"} \\ \mathit{pop}_{r_0}(r_1) \\ \text{if iszero}(r_1) \\ \quad \text{then } \llbracket c_1 \rrbracket_{\rho, x \mapsto [r_0]} \\ \quad \text{else } \llbracket c_2 \rrbracket_{\rho, y \mapsto [r_0]} \end{array}$$

Pop the tag from the stack r_0. Generate the appropriate branching instructions using `iszero`. Because the tag has been popped from r_0, the value under the injection is at the top of the stack.

Case expression with variable
${}^{-}[\![\text{case } z^{❶} \text{ of } \{\text{inl } x \mapsto c_1; \text{inr } y \mapsto c_2; \}]\!]_{\rho, z \mapsto r_1 : s_n} = \begin{array}{l} \text{if iszero}(r_1) \\ \quad \text{then } [\![c_1]\!]_{\rho, x \mapsto s_n} \\ \quad \text{else } [\![c_2]\!]_{\rho, y \mapsto s_n} \end{array}$
Because z is not a stack, it corresponds to a non-empty list of pseudo registers where the head is the tag, and the remaining pseudo registers correspond to the value under the injection.
Function call
${}^{-}[\![\text{call } z^{❶}(v)]\!]_{\rho, z \mapsto [r_0]} = \begin{array}{l} [\![v]\!]^{\omega}_{\rho} \\ \textit{jmp } r_0 \end{array}$
Compile v, preparing the stack, and then jump to the label r_0.

Figure 19: Compilation scheme for the negative fragment of Lithium.

4.3. Compilation Target

When picking a compilation target there are always a lot of options, and for Lithium, x86-64 was picked. While choices like LLVM IR provide a lot of benefits to the developer in terms of development speed and convenience, one ultimately sacrifices some control over things like the calling convention and memory allocation. Due to Lithium's CPS nature, tail call optimization is a must. Languages like LLVM IR, provide tools and syntax for this, but a developer can not guarantee how the stack is handled when functions are called, nor how arguments are passed to these functions. For this explicit need of control x86-64 was determined to be a fitting choice.

Utilizing the flexibility given by x86-64, Lithium gains a lot of control over how the calling convention is implemented and how the stack, registers, and memory in general is used. In other words, it gives us the ability to have complete control over the language's application binary Interface (ABI). See Section 4.4 for details about the ABI.

Similarly to other languages Lithium uses stack frames for function calls, but unlike other languages, Lithium only uses one stack frame during normal execution. This is possible due to the fine grained control x86-64 gives a developer and the fact that Lithium is written in CPS. Every function call can be tail-call optimized, because the sequence of commands in a function is terminated by a function call.

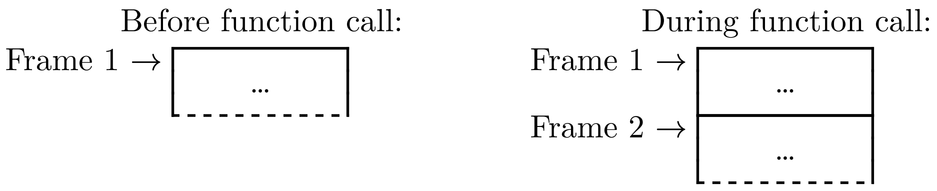


Figure 20: Function call without tail call optimization

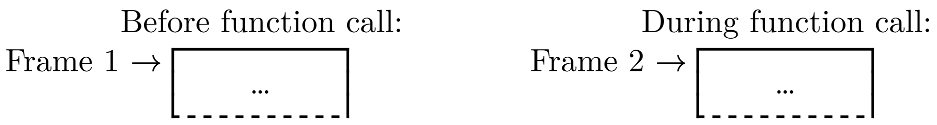


Figure 21: Function call with tail call optimization

As can be seen in Figure 20 and Figure 21, when using tail-call optimization the last stack frame is simply replaced by the new frame. This optimization is however not guaranteed when calling functions using Foreign Function Interfaces (FFI) calls, because the function that is being called might allocate stack frames. While FFI is not exposed to the user, it is still used internally at the time of writing, because libc is used for printing and allocating memory on the heap.

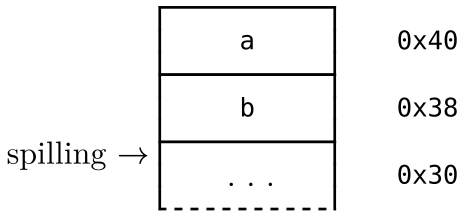


Figure 22: A stack frame containing the variables `a` and `b`, and any register spilling will occur in address space `0x30` and below.

The single stack frame is used for variable storage and register spilling[1], and its size is determined at compile time and can vary between functions. This size is based on the amount of variables used by the function, and does not account for register spilling because pushing and popping updates stack frame size dynamically.

4.4. Language ABI

As with any language, one should define an application binary Interface (ABI), and in turn a calling convention. As said before in Section 4.3, Lithium only uses one stack frame during normal execution which is used for

[1]Register spilling means backing up a register on the stack, freeing the register temporarily.

variable storage and register spilling. This stack frame is located on the stack given by the operating system, which will be referred to as the system stack henceforth, and outside of this system stack, Lithium makes heavy use of other stacks. These stacks are used for passing variables and all calculations. This stack usage is similar in nature to the Java Virtual Machine (*Oracle*, n.d.) or WebAssembly (Haas et al., 2017), which also makes heavy use of stacks and virtual registers stored on the system stack.

4.4.1. Function calls

When functions are called these are the pre-conditions that must be fulfilled:

- Register `R15`is set to an address which points to a valid stack.
- The address in `R15` must be a multiple of 8 (4 on a 32-bit system).
- The stack size should be big enough for all variables in the function.
- All expected arguments exist on the stack.
- All arguments are properly aligned.
- The bottom of the stack contains a pointer to the start of the stack.
- The stack grows downwards.

The reason stacks in Lithium grow downwards is because system stacks on most architectures also grow downwards. This allows an implementation of the language to allocate a stack on the system stack if need be, and in turn use the built in pop and push instructions, which exists in most instruction sets.

When allocating new stacks, the first value on the stack must be the pointer which points to the start of the stack. Because stacks grow downward their pointers need to be offset by their size on allocation, but this creates a problem since we can not deallocate using this updated pointer. To alleviate this, the original pointer is placed on the stack, which can then be popped when the stack is empty and `freestack` is called. This also opens up for stacks to be allocated at differing sizes, as the process deallocating a stack does not need to know the size of said stack.

3c	3b	3a	39	38	37	36	35	34	33	32	31	30	2f	2e	2d
start pointer								Free space...							

Figure 23: An empty stack only containing the start pointer.

All functions in Lithium are called through jump(with some discrepancies for FFI). This is implemented by passing around all functions as values on the stack, and popping them and jumping to them when applicable. Top-level functions work differently however, and act more like constants do in most other languages.

A top-level function is not actually generated as a function, and it is instead a static constant which contains a code pointer to the actual function. This code pointer can then be copied and pushed onto the stack, and called when need be.

When FFI calls occur, such as calling a libc function like `printf`, this function will allocate a stack frame on top of the single stack frame, and execute like it would normally do. The result will then be written into fitting pseudo-registers, or it will be pushed onto the current stack. This and along with top-level functions, are the only time Lithium strays from the strict continuation based style.

4.4.2. Mapping types to memory

An important part of any ABI is specifying how types are represented. The following chapter specifies the memory representation of types in Lithium, and the number of physical registers needed to store them.

In the table `Word` represents 8 bytes, and ∞ is a memory section of unknown size, and it is used to represent a stack (see Section 4.4.3 for a more detailed explanation). In general a `Word` depends on the architecture of the CPU, but on a 64-bit CPU a `Word` is often considered to be 8 bytes.

Type	**Registers** $[\![A]\!]^R = \mathbb{N}$	**Memory** $[\![A]\!]^M = \mathbb{N}$
Product-type	$[\![A \otimes B : \omega]\!]^{\mathrm{R}} = 1$	$[\![A \otimes B : \omega]\!]^{\mathrm{M}} = [\![A]\!]^{\mathrm{M}} + \infty$
Product-type	$[\![A \otimes B : ❶]\!]^{\mathrm{R}} = [\![A]\!]^{\mathrm{R}} + [\![B]\!]^{\mathrm{R}}$	$[\![A \otimes B : ❶]\!]^{\mathrm{M}} = [\![A]\!]^{\mathrm{M}} + [\![B]\!]^{\mathrm{M}}$
Sum-type	$[\![A \oplus B : \omega]\!]^{\mathrm{R}} = 1$	$[\![A \oplus B : \omega]\!]^{\mathrm{M}} = \texttt{Word} + \infty$
Sum-type	$[\![A \oplus B : ❶]\!]^{\mathrm{R}} = 1 + \max([\![A]\!]^{\mathrm{R}}, [\![B]\!]^{\mathrm{R}})$	$[\![A \oplus B : ❶]\!]^{\mathrm{M}} = \texttt{Word} + \max([\![A]\!]^{\mathrm{M}}, [\![B]\!]^{\mathrm{M}})$
Static function	$[\![* A : ❶]\!]^{\mathrm{R}} = 1$	$[\![* A : ❶]\!]^{\mathrm{M}} = \texttt{Word}$
Linear pointer	$[\![\square A : ❶]\!]^{\mathrm{R}} = 1$	$[\![\square A : ❶]\!]^{\mathrm{M}} = \texttt{Word}$

Empty stack	$[\![\bigcirc : \omega]\!]^{\mathrm{R}} = 1$	$[\![\bigcirc : \omega]\!]^{\mathrm{M}} = \infty$
Unit	$[\![\mathbf{1} : \text{❶}]\!]^{\mathrm{R}} = 0$	$[\![\mathbf{1} : \text{❶}]\!]^{\mathrm{M}} = 0$
Empty	$[\![\mathbf{0} : \text{❶}]\!]^{\mathrm{R}} = 0$	$[\![\mathbf{0} : \text{❶}]\!]^{\mathrm{M}} = 0$
$\exists$ intro	$[\![\exists\alpha.A : \text{❶}]\!]^{\mathrm{R}} = [\![A]\!]^{\mathrm{R}}$	$[\![\exists\alpha.A : \text{❶}]\!]^{\mathrm{M}} = [\![A]\!]^{\mathrm{M}}$
$\exists$ intro	$[\![\exists\alpha.A : \omega]\!]^{\mathrm{R}} = [\![A]\!]^{\mathrm{R}}$	$[\![\exists\alpha.A : \omega]\!]^{\mathrm{M}} = [\![A]\!]^{\mathrm{M}}$
Type variable	$[\![\alpha : \omega]\!]^{\mathrm{R}} = 1$	$[\![\alpha : \omega]\!]^{\mathrm{M}} = \infty$

Outside of these types, Lithium contains an auxiliary type: a word sized integer. The memory specification of integer is the following:

Integer	$[\![int]\!]^{\mathrm{R}} = 1$	$[\![int]\!]^{\mathrm{M}} =$ `Word`

4.4.3. Memory alignment

At the time of writing, Lithium does not contain that many different types, and currently it is limited to integers, function pointers, stack pointers, and product- and sum-types.

Memory wise, the simplest here are function pointers and stack pointers. Both of these are simply the size of a word, and they can always fit in a register, and thus never need to be split up across multiple registers when working with them.

Integers are currently also simple, because they are also the size of a word. This may however change in the future because it is useful to have access to integers of different sizes, especially when working in a system-level context. When integers of different sizes are introduced, memory alignment needs to be taken into consideration.

When pushing values of different sizes to the stack, alignment needs to be considered. Take this stack that just contains a 16-bit integer with the value `42`.

3c	3b	3a	39	38	37	36	35	34	33	32	31	30	2f	2e	2d
🟩 $42		Free space...													

Now if we want to push another value, say a 32-bit integer with the value `777`, we need to pad it so the value is placed on an address which is divisible by its size.

3c	3b	3a	39	38	37	36	35	34	33	32	31	30	2f	2e	2d
$42		padding						$777				padding			

As can be seen we are padding by 6. Theoretically we only need to pad by 2 bytes here for a 4 byte integer, because 4 is divisible by 2. The reason this is done is to both to simplify the compilation process, and to simplify the needed code for any given pop and push.

A pop/push should always be able to expect that the pointer to the stack is currently correctly aligned, and this is done to minimize the amount of instructions needed when interacting with the stack. If a function is called through FFI for instance, it has no way of knowing if the stack pointer is currently aligned, unless it explicitly checks the current stack pointer and manually aligns. This would be a waste of computation time, and for this reason, all pushes pad to make sure that the next stack location is in an address divisible by 8 (would be 4 on a 32-bit platform).

The reason padding is needed is simply because a lot of computer architectures assume that memory is stored in an aligned way. If a value consists of 8 bytes, it needs to be stored on an address divisible by 8, if its 4 bytes, the address needs to be divisible by 4, and so on and so forth. While x86-64 allows unaligned memory interactions for some instructions, this is often heavily discouraged because it can potentially harm performance, because it can require more clock cycles, and hurt the cache friendliness of the values being interacted with.

The memory layout for product- and sum-types is also relatively simple. When we put a sum-type value such as `inl 42` on the stack it looks like this:

3c	3b	3a	39	38	37	36	35	34	33	32	31	30	2f	2e	2d
$42								0							

and similarly if we put the value `inr 777` on the stack, it will look like this:

3c	3b	3a	39	38	37	36	35	34	33	32	31	30	2f	2e	2d
$777								1							

Observe here that the tag is put on the right-most position, i.e. at the top of the stack, and that the tag is 8 bytes.

Almost the same goes for product-types, so for a value such as `(1, 2)` we simply put them after one another:

3c	3b	3a	39	38	37	36	35	34	33	32	31	30	2f	2e	2d
$1								$2							

For a more complex value such as `(1,(2,3))` it would look like this:

3c	3b	3a	39	38	37	36	35	34	33	32	31	30	2f	2e	2d
$1								$2							
2c	2b	2a	29	28	27	26	25	24	23	22	21	20	1f	1e	1d
$3								Free space...							

As can be seen no information outside of the actual values is stored.

Similar alignment should be done when storing variables on the system stack, but this is not enforced by this ABI, because the system stack is not used when passing variables.

In Section 4.4.2 we have seen more complicated memory mappings of types, specifically mappings involving ∞. Take $A \otimes B : \omega$ for instance, whose memory usage is calculated as: $[\![A]\!]^{\mathrm{M}} + \infty$. Here A is a type with kind ❶ and B has kind ω, hence it is a stack. Visually this would be represented like this (if $[\![A]\!]^{\mathrm{M}} =$ `Word`):

...								34	33	32	31	30	2f	2e	2d
∞								A							

This means that B is a stack of unknown size, but we know that *on top* of B there are 8 bytes dedicated to a value of type A. Without A, B could potentially be empty, or it could be huge, but we cannot know from the types alone.

5. Discussion

This chapter presents a simple benchmark of Lithium using the recursive fibonacci function. We also compare it to the equivalent C program. Furthermore, we also present and suggest future work for the language and the compiler.

5.1. Usage

Fibonacci is not the most interesting program, but from the standpoint of linearity it is *somewhat* interesting, because it requires the reuse of variables:

$$\begin{aligned} F(0) &= 0 \\ F(1) &= 1 \\ F(n) &= F(n-1) + F(n-2) \end{aligned}$$

Figure 24 contains two different implementations of the fibonacci function: one in Lithium, and a recursive one in C.

The Lithium version uses the compiler defined functions `__dup__` and `__eq__`. `__dup__` has the type signature $*\,(\mathit{int} \otimes \sim (\mathit{int} \otimes \mathit{int}))$, which means it takes a tuple containing the integer to duplicate, and a continuation that takes the two new integers as argument. `__eq__` has type signature $*\,(\mathit{int} \otimes \mathit{int} \otimes \sim (\mathbf{1} \oplus \mathbf{1}))$. This function takes two integers to check for equality, and a continuation that takes the result as argument. The value `inl ()` represents true and the value `inr ()` represents false.

A Lithium version:

```
fib : *(int ⊗ ~int) =
  \n,k ->
    __dup__(n,       \n,n'    ->
    __eq__((n', 0), \is_zero ->
    case is_zero of {
      inl () ->
        __dup__(n,       \n,n'   ->
        __eq__((n', 1), \is_one ->
        case is_one of {
          inl () ->
            __dup__(n, \n,m ->
            fib((n-1), \n   ->
            fib((m-2), \m   ->
            k(n+m))));
          inr () -> k(n);
        }));
      inr () -> k(n);
    }));
```

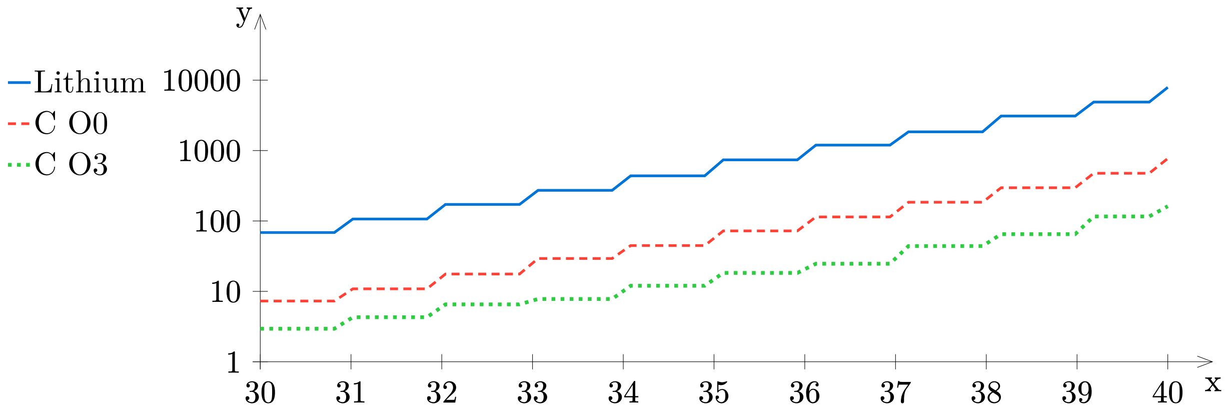
A recursive version made in C:

```
long fib(long m) {
  if (m == 0 || m == 1) {
    return m;
  }
  long a = fib(m - 1);
  long b = fib(m - 2);
  return a + b;
}
```

Figure 24: Fibonacci in Lithium.

5.1.1. Benchmark

The y-axis in the benchmark represents time measured in milliseconds, and the x-axis represents the input to the fibonacci function. Note that the y-axis grows logarithmically.



,

Figure 25: Benchmark comparing the average time needed to calculate the fibonacci numbers 30 to 40 in the two different implementations.

As can be seen in the benchmark in Figure 25 there is quite a large gap between the version written in Lithium and the version written in C. Each input was measured 100 times. The C version was compiled using GCC with O0 and O3. However, while the Lithium version is around one order of magnitude slower than the unoptimized C version, the growth in execution time is a almost constant factor between all three programs.

This can be observed even more clearly if we overlap the results over each other. We can see that the unoptimized C version grows the same way as the Lithium version, while the optimized C version is slightly more efficient.

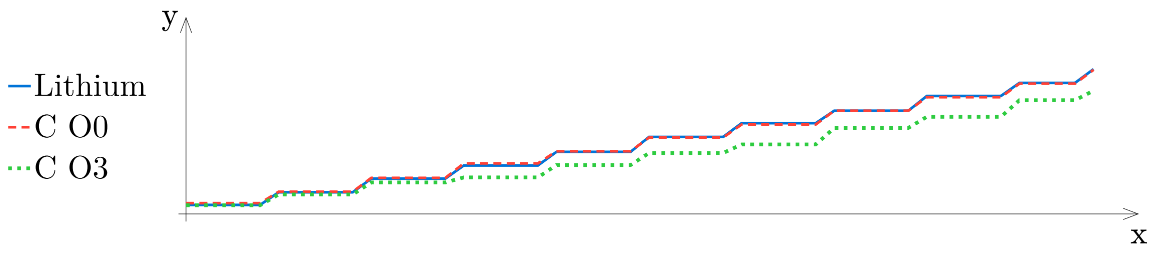


Figure 26: Benchmark from Figure 25 with results overlapped.

The performance difference we can see Figure 25 can be attributed to several factors but the two most significant ones are most likely that Lithium is currently not optimized at all, and it generates more code than is likely necessary. Also worth to note that the calling convention is less performant than System V (Matz et al., 2013) which C uses. System V prioritizes passing arguments using registers while Lithium strictly passes arguments on stacks.

5.2. Future Work

Developing a language and a compiler is a continuous process of refinement and discovery, seldom with a clear finish line in sight. In this chapter we discuss some features to work on in the immediate future. The proposed future work is listed in no particular order.

5.2.1. Compiler interface for creating stacks

In the current implementation of Lithium, all stacks are heap-allocated. This can lead to unnecessary overhead, and can be a hindrance for using the language on more restricted platforms such as embedded devices. To mitigate this, giving the developer the option to allocate stacks on the system stack would work (maybe as a compiler flag or as a linked library).

To implement this the language would need to include a runtime which can keep track of which memory regions on the system stack are currently in use, to make sure that stacks are not overlapping. It is also important to allocate these dynamic stacks *above* the programs singular stack frame (remember: the stack grows downwards). This is important because accessing memory below the stack pointer is considered undefined behavior on a lot of platforms because hardware or an OS may potentially use that memory for things such as interrupts or exceptions. This is similar in concept to GCC's split stacks, which lets threads share a fragmented stack, leading to less allocations.

Another feature that can be added is to add an option allowing discontiguous stacks, similarly to GCC's split stack option. Newly allocated stacks start out as small, and as the stacks start being filled, a new stack is allocated (either on the heap, or on the system stack) and linked to the old stack. This would allow stacks to be small, but grow in size dynamically if need be. Implementing this would place some requirements on the push and pop push semantics however, because we would need to keep track of the stack size while the program is running, and allocate more memory if need be. This bookkeeping could have an impact on performance, and should ideally be as fast as possible. Overall, this option would effectively trade a little bit of performance for potentially more effective memory usage.

5.2.2. Register allocation

When optimizing the generated code, one important technique is utilizing the physical registers for variable allocation. If one were to implement this naively, it would suffice to put the first few variables of a function into registers, and the rest on the system stack. This would however not necessarily suffice for functions where there are more variables than available physical registers. In this case it would be more efficient to analyze the usage of variables, and prioritize them based on the order they are used.

Doing this efficiently can prove difficult because the compiler has to minimize the amount of register spilling needed, and this has in fact been proven to be NP-complete (Bouchez et al., 2006). This was not implemented for Lithium because it was deemed an optimization that, while interesting, was not something we would have time to implement properly.

5.2.3. System Calls

Due to Lithium being a system-level language, it may be useful to have the ability to directly make system calls to interact with the operating system. This allows for the possibility of not relying on, for example, libc, because the behavior one would need from libc could be re-implemented in the language. This would in theory make the language more portable, because it would be easier to port it to systems where libc might not be available. System calls are used under the hood in the language, but it is not something that is exposed to a user of the language. One simple approach would be to implement a compiler function, similar to `__dup__` and `__eq__` called `__sys__`. This function would have the type signature $*\,(\mathit{int} \otimes \texttt{"}(\mathit{int} + \mathbf{1}) \cdot 6\texttt{"} \sim \mathit{int})$, where the meta syntax $\texttt{"}(\mathit{int} + \mathbf{1}) \cdot 6\texttt{"}$ stands for the 6-tuple of $\mathit{int} + \mathbf{1}$, which is used for optionality. The first argument would be the number of the system call, and the 6-tuple represents the arguments passed to it.

5.2.4. Foreign Function Interface

Similarly to system calls, proper support for Foreign Function Interfaces (FFI) would permit the language to interact with software outside. A prime example of this would be libraries written in other languages. Just like system calls, FFI is currently used under the hood for printing and the likes using libc, but this is not exposed to a user of the language.

5.2.5. Exponentials

While linearity in a programming language is useful for managing resources such as memory, sometimes one needs to use values more than once. Consider the fibonacci function where we do not use `__dup__`:

```
fib : *(int ⊗ ~int)
  = \(n,k) -> __eq__((n,0), \res -> case res of {
    case res of {
        inl () -> k(0);
        inr () ->
          __eq__((n,0), \res -> case res of {
            inl () -> k(1);
            inr () ->
              fib((n-1, \r1 ->
              fib((n-2, \r2 -> k(r1 + r2)))))
          })
      }
  })
```

This function does not compile sadly, because the variable `n` is used 4 times, and due to linearity one may only use it once! To address this issue we would want to introduce exponentials.

Exponentials would let a user reuse a value multiple times opening up for some much needed expressiveness. Take Fibonacci again with syntax for duplicating exponential variables and reading exponential variables. The variable z has type $!A$

- let $a + b = z$: the environment is extended with $a : !A$ and $b : !A$
- let $!z_1 = z$: the environment is extended with $z_1 : A$.

The first extension corresponds to the Duplicate rule in Figure 6, and the second one corresponds to Derelict in the same figure. We would also need syntax for Promote and Discard, but we leave that to the reader's imagination.

```
fib : *(!int ⊗ ~int)
  = \(n,k) ->
      let n1 + z1 = n; -- duplicate as a pattern, not addition
      let !z2 = z1;
      __eq__((z2, 0), \res -> case res of {
        inl () ->
          let n2 + o1 = n1;
          let !o2 = o1;
          __eq__((o2, 1), \res -> {
            inl () ->
              let n3 + p = n2;
              let !n3 = n3;
              let !n4 = p;
              fib((n3 - 1, \r1 ->
              fib((n4 - 2, \r2 ->
              k(r1 + r2)))));
            inr () -> k(1);
          });
        inr () -> k(0);
      });
```

As can be seen here, we can now re-use the value in `n`, allowing us to write the fibonacci function without compiler defined functions.

Although the function is now possible, it is still noisy. If we were to introduce some syntax sugar, for example a `*` operator that combines both Duplicate and Derelict. we could simplify the term: `let a + b = n; let !n1 = a; let !n2 = b; k(n1 + n2)` to: `k(*n + *n)`. Rewriting the fibonacci function with this operator would result in:

```
fib : *(!int ⊗ ~int)
  = \(n,k) ->
    __eq__((*n, 0), \res -> case res of {
      inl () ->
        __eq__((*n, 1), \res -> case res of {
          inl () ->
            fib((*n - 1, \r1 ->
            fib((*n - 2, \r2 ->
            k(r1 + r2)))));
          inr () -> k(1);
        });
      inr () -> k(0);
    });
```

Because exponentials create more than one reference to memory, we would need automatic deallocation, or we risk leaking memory. This could be solved using reference counting, garbage collection or any other alternative.

5.2.6. Data Types

While the language currently contains sum and product types ($a \oplus b$ and $a \otimes b$ respectively), having types with named fields or constructors would be useful for more complex types. It would also be required for recursive types such as trees or linked lists. Having access to contiguous data types such as the array is also a very useful thing when contiguous memory is wanted.

In other immutable languages such as Haskell, data is copied when you operate on it. Take this Haskell function for example:

```haskell
plus1 :: [Int] -> [Int]
plus1 xs = map (+1) xs
```

In Haskell any list you input into this function will be duplicated, returning a new list while keeping the original, in case it is used again. In a linear language the original list would no longer be usable, and this opens up room for optimizations. Instead of working on a copy of the old list, a linear language could mutate the original list in place.

As long as the size of the type contained by the list is not changed, a function such as map can simply mutate the content, and thus just replace the original data. This can lead to major performance benefits in programs where data is pipelined in such a way where there is not a lot for need for duplication.

5.2.7. Using Lithium as a compilation target

While just writing Lithium on its own works, a good benchmark and milestone of the language's capability would be another language using Lithium as a compilation target.

If the work presented in this chapter were implemented, then using Lithium as a compilation target would be feasible. A future project could be to compile a subset of Linear Haskell (Bernardy et al., 2017) (or other linear functional languages) to Lithium. This would make it easy to see what features or areas Lithium is lacking in, making future goals clearer.

6. Conclusion

This thesis presented the system-level functional programming language Lithium. Lithium takes advantage of linear types to provide a safe and reliable intermediate language. By applying a series of transformations to Lithium, the language is made assembly compatible. The transformations turn higher-order closure and procedures into labels and jump, with environments being translated into stacks, and pointers to stacks are made explicit.

The thesis continued by providing a compilation scheme that turns the lowered language into a set of pseudo assembly instructions, which in turn were easily translated into assembly language. Additionally, the thesis presented the application binary interface (ABI) for interoperability. The chapter detailed the pre-conditions for function calls and how memory is structured.

Finally, the thesis ended by presenting an example of a program written in Lithium, a simple benchmark for context, and proposed several future extensions and optimizations to refine and improve the language.

References

Appel, A. W. (2007). *Compiling with continuations*. Cambridge university press.

Bernardy, J.-P., Boespflug, M., Newton, R. R., Peyton Jones, S., & Spiwack, A. (2017). Linear Haskell: practical linearity in a higher-order polymorphic language. *Proceedings of the ACM on Programming Languages*, *2*(POPL), 1–29.

Bernardy, J.-P., Juan, V. L., & Svenningsson, J. (2016). Composable efficient array computations using linear types. *Unpublished Draft*.

Bouchez, F., Darte, A., Guillon, C., & Rastello, F. (2006). Register allocation: What does the NP-completeness proof of Chaitin et al. really prove? or revisiting register allocation: Why and how. *International Workshop on Languages and Compilers for Parallel Computing*, 283–298.

Church, A. (1940). A formulation of the simple theory of types. *The Journal of Symbolic Logic*, *5*(2), 56–68.

Cranelift. https://cranelift.dev/

Fradet, P., & Le Métayer, D. (1991). Compilation of functional languages by program transformation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *13*(1), 21–51.

Gill, A., Launchbury, J., & Peyton Jones, S. L. (1993). A short cut to deforestation. *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, 223–232.

Gimple. https://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html

Girard, J.-Y. (1972). *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*.

Girard, J.-Y. (1987). Linear logic. *Theoretical Computer Science*, *50*(1), 1–101.

Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., Wagner, L., Zakai, A., & Bastien, J. (2017). Bringing the web up to speed with WebAssembly. *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 185–200.

Howard, W. A., & others. (1980). The formulae-as-types notion of construction. *To HB Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, *44*, 479–490.

Hughes, J. (1989). Why functional programming matters. *The Computer Journal*, *32*(2), 98–107.

Jones, S. P., Hall, C., Hammond, K., Partain, W., & Wadler, P. (1993). The Glasgow Haskell compiler: a technical overview. *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, *93*, 13.

Kelsey, R., & Hudak, P. (1989). Realistic compilation by program transformation (detailed summary). *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 281–292.

Lafont, Y. (1988). The linear abstract machine: Theoretical computer science 59 (1988) 157–180. *Theoretical Computer Science*, *62*(3), 327–328.

Lattner, C., & Adve, V. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 75–86.

Laurent, O. (2002). *Etude de la polarisation en logique.*

Matsakis, N. D., & Klock, F. S. (2014). The Rust language. *ACM Sigada Ada Letters*, *34*(3), 103–104.

Matz, M., Hubicka, J., Jaeger, A., & Mitchell, M. (2013). System v application binary interface. *Amd64 Architecture Processor Supplement, Draft V0*, *99*(2013), 57.

Nordmark, F. (2024). *Towards a Practical Execution Model for Functional Languages with Linear Types.*

Oracle. https://www.oracle.com/java/

Place, O. A., & Cleveland, S. *x86-64 TM Technology White Paper.*

Wikipedia. (2024,). *Systems programming — Wikipedia, The Free Encyclopedia.*