UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

VALMIR PRETTO DO AMARAL JÚNIOR

Curios - a web of types

Thesis presented in partial fulfillment of the requirements for the degree of Master of Computer Science

Advisor: Prof. Dr. Álvaro Moreira
Coadvisor: Prof. Dr. Rodrigo Machado

Porto Alegre
September 2023

CIP — CATALOGING-IN-PUBLICATION

Pretto do Amaral Júnior, Valmir

Curios - a web of types / Valmir Pretto do Amaral Júnior. – Porto Alegre: PPGC da UFRGS, 2023.

87 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2023. Advisor: Álvaro Moreira; Coadvisor: Rodrigo Machado.

1. Dependent types. 2. Compilation. 3. WebAssembly. I. Moreira, Álvaro. II. Machado, Rodrigo. III. Título.

“Enlightenment leads to benightedness,

science entails nescience.”

— PHILIPPE VERDOUX

ABSTRACT

Ever since their inception lambda calculus and type theory have been such an influence in the design of modern programming languages so much so that even programming languages outside of the functional circle have adopted its practices. But even inside the functional circle, not many programming languages have dared to venture beyond the higher order polymorphic lambda calculus, towards the calculus of constructions: it can be troublesome to incorporate dependent types into a more traditional type system without constraining the calculus in some way, and these constraints usually lead to the calculus not being a good theoretical foundation for a programming language. However, programming languages like Idris and Agda have proven that there exists benefits in bringing full, unrestricted dependent types to the type system of functional programming languages.
Despite the advances that these programming languages have achieved in the research surrounding advanced type systems, they do not target an executable format and, for the purposes of execution, choose to transpile their programs into other programming languages. While transpilation has its advantages, it also has its drawbacks: the programming language now depends on the compiler toolchain of a second, separate programming language. This gives rise to a question: would it be desirable to compile a dependently typed programming language to an executable format without indirection?
In the present research project we explore the topic of compiling Curios, a dependently typed functional programming language to WebAssembly, a binary instruction format for a stack-based virtual machine. We describe Curios through practical examples showcasing its syntax and basic features, and how to represent more complex concepts by using its basic features as building blocks. The type system of Curios is formally specified next, and following that we give a detailed explanation on the inner workings of the compiler, from source code to executable. We conclude by presenting the bodies of work that have influenced Curios, what was achieved and what was left as topics for future research.

Keywords: Dependent types. Compilation. WebAssembly.

Curios - Uma Rede de Tipos

RESUMO

Desde o seu princípio o cálculo lambda e a teoria dos tipos têm sido uma influência tão grande no design de linguagens de programação modernas que mesmo linguagens de programação fora do círculo funcional adotaram as suas práticas. Mas mesmo dentro do círculo funcional, não existem muitas linguagens de programação que se atreveram a se aventurar além do cálculo lambda polimórfico de order maior, em direção ao cálculo das construções: pode ser problemático incorporar tipos dependentes em um sistema de tipos mais tradicional sem restringir o cálculo de alguma forma, e estas restrições normalmente fazem o cálculo não ser uma fundamentação teórica tão boa para uma linguagem de programação. No entanto, linguagens de programação como Idris e Agda provaram que existem benefícios em trazer tipos dependentes completos e irrestritos para o sistema de tipos de linguagens de programação funcionais.

Apesar dos avanços que estas linguagens de programação conquistaram na pesquisa em torno de sistemas de tipos avançados, elas não têm como alvo um formato executável e, com o objetivo de serem executadas, escolhem transpilar os seus programas para outras linguagens de programação. Embora a transpilação tenha as suas vantagens, ela também tem suas desvantagens: a linguagem de programação acaba por depender das ferramentas de compilação de uma segunda, distinta linguagem de programação. Isso levanta uma pergunta: seria desejável compilar uma linguagem de programação para um formato executável sem indireção?

No presente projeto de pesquisa exploramos o tópico de compilar Curios, uma linguagem de programação dependentemente tipada para WebAssembly, um formato de instruções binárias para uma máquina virtual baseada em pilhas. Descrevemos Curios atráves de exemplos práticos demonstrando a sua sintaxe e funcionalidades básicas, e como representar conceitos mais complexos usando as suas funcionalidades básicas como ingredientes modulares. O sistema de tipos de Curios é formalmente especificado por próximo, e em seguida damos uma descrição detalhada das operações internas do compilador, de código fonte até executável. Concluímos apresentando os trabalhos que influenciaram Curios, o que foi alcançado e o que foi deixado como tópico para pesquisa futura.

Palavras-chave: Tipos dependentes, Compilação, WebAssembly.

LIST OF ABBREVIATIONS AND ACRONYMS

Wasm	WebAssembly
HTML	Hyper Text Markup Language
JS	JavaScript
XML	Extensible Markup Language
JSX	JavaScript XML
PHP	Hypertext Preprocessor
ISA	Instruction Set Architecture
UTF-8	Unicode Transformation Format – 8-bit
DSL	Domain Specific Language
API	Application Programming Interface
S-expr	Symbolic expression
CIC	Calculus of Inductive Constructions
GADT	Generalized Algebraic Data Type

LIST OF FIGURES

CONTENTS

1 INTRODUCTION

In recent years, functional programming languages have enjoyed a surge in popularity, with many software programmers discovering the benefits that the paradigm can offer. Functional idioms such as pattern matching and first-class functions have even begun to not only be adopted by well-established, market staple programming languages but also influence the design of every programming language to come.

Functional programming is known for its strong roots in lambda calculus and type theory. Languages such as Haskell and OCaml can have their advanced user facing features reduced to a concise core language based on the higher order polymorphic lambda calculus, also known as System Fω (PIERCE, 2002). To briefly recapitulate, System Fω is a type system that allows terms to depend on terms (functions), terms to depend on types (polymorphism), types to depend on types (type constructors), but it does **not** allow types to depend on terms (dependent types).

Under the Curry-Howard correspondence (SØRENSEN; URZYCZYN, 2006), the problem of verifying the correctness of a proof can be reduced to the problem of checking whether a program inhabits a type: a proposition is equivalent to a type, and the proof of that proposition is equivalent to a program. Dependent types employ the Curry-Howard correspondence specially well because, by allowing terms to appear at the type level, dependent types can act as formulas of intuitionistic predicate logic (MARTIN-LÖF; SAMBIN, 1984) and, as a consequence, they became a feature most commonly associated with proof assistants.

Dependent types do not come without their downsides though: full dependent types (i.e. allowing terms to occur without restriction in types) leads to undecidable type inference. In short, type inference for full dependent types can be reduced to a problem called *semi-unification* where solving the set of type constraints collected from the expressions of the program can lead to non-termination (KFOURY; TIURYN; URZYCZYN, 1990; DOWEK, 1993). The Calculus of Inductive Constructions (PAULIN-MOHRING, 2015) (which is the formalism behind the Coq[1] and Lean[2] proof assistants) sidesteps this issue by defining a set of schemes for the formation of types that restrict their recursive occurrences in such a way that termination is obtained.

In functional programming languages, this restriction is considered to be too heavy-handed because it rules out a number of popular idioms involving general recursion, which

[1]<https://coq.inria.fr/>
[2]<https://leanprover.github.io/>

led to dependent types being a feature relinquished to proof assistants. Even though dependency of types on terms is not a novel topic when the subject is programming languages[3], in most instances they were accompanied by a whole host of limitations.

Notwithstanding the challenge of incorporating full dependent types into general purpose programming, Idris (BRADY, 2013) demonstrates the feasibility of dependently typed idioms as tools for writing better software. For example, session types (MUIJNCK-HUGHES; BRADY; VANDERBAUWHEDE, 2019) allow communication protocols to be specified entirely at the type level, potentially allowing web apps to verify that their communication with a server is statically checked by the type system. Another example is the `Control.ST` module (BRADY, 2016) which offers a monad for dependently typed effects, which has the potential to give more precise types for effectful idioms such as centralized state containers.

The current landscape regarding execution of programs written in dependently typed languages has a bias towards transpilation: Idris targets C and Javascript (IDRIS CONTRIBUTORS, 2020); Agda targets Haskell and JavaScript (AGDA CONTRIBUTORS, 2022); Idris2, the next iteration of Idris, targets Chez Scheme (IDRIS2 CONTRIBUTORS, 2022). A significant downside of transpilation is that a transpiled programming language ends up indirectly depending on the entire compiler toolchain of a second language before its programs can be executed. With that in mind, we raise a question: would it be desirable to target an executable format (or at the very least, a low-level intermediate representation) with a dependently typed language?

WebAssembly (ROSSBERG, 2022), a binary instruction format for web apps that is **fast**, **safe** and **portable**, has taken the web programming landscape by storm. JavaScript used to be the only option both as a programming language and as a transpilation target for other programming languages, but WebAssembly now offers the opportunity for compiled languages such as C++ and Rust, which are seen as low-level programming languages, to be executed on the web browser. Just as important as bringing existing languages to the web programming landscape, WebAssembly has the potential to nurture new programming languages targeting its instruction set specifically. With that in mind, we refine our original question: can WebAssembly serve as the medium for executing dependently typed programs on the web browser?

In the present work, we explore how one might approach the task of compiling a dependently typed programming language by introducing the dependently typed func-

[3]As evidenced by Pascal which, in 1970, allowed the type of an array to be indexed by its size.

tional programming language Curios. The main selling points of Curios are:

- **It is general purpose.** Curios proposes a method for reconciling general recursion with dependent types, two notions that can be viewed as contradictory. Its type system offers an alternative to the Calculus of Inductive Constructions that has the potential to lend itself well for programming languages;
- **It targets WebAssembly.** The benefits of WebAssembly as a compilation target are perceptible. Curios seeks to validate the aptitude of WebAssembly as a compilation target for a dependently typed programming language.

The report on the research project surrounding Curios begins in Chapter 2 where the concepts necessary to understand the project as a whole are introduced. The main contributions are presented in three major chapters: in Chapter 3, the Curios programming language is introduced through practical examples showcasing its syntax and features; in Chapter 4 the specification of Curios' syntax, type system rules and operational semantics are formally introduced; in Chapter 5 a detailed explanation is given on the internals of Curios' compiler. Works related to Curios are given in Chapter 6 along with the extent of their influence in the research and development of Curios. The report is finished in Chapter 7 with a discussion on what Curios has and has not achieved along with the plans for the future.

2 BACKGROUND

Curios relies on many distinct pieces, each with its intricacies. In this section, we will introduce the reader to the most important components of the entire system through a summary of each of them.

Section 2.1 has a summary of what dependent types are, their advantages, disadvantages and how they can be used. Section 2.1.1 has an overview of the Coq proof assistant, and in Section 2.1.2 an overview of the Idris programming language is given. Both of them feature dependent types as part of their type systems in some way.

Section 2.2 has a brief history on the subject of programming for the web, along with the expected future trend. What follows, in Section 2.3, is an introduction on WebAssembly, its main allure and what it aims to achieve. Section 2.3.1 presents an overview of the main characteristics of WebAssembly, as well as details about what constitutes a WebAssembly program. To finish the WebAssembly section, examples of programs written using WebAssembly's text format are shown in Section 2.3.2.

2.1 Dependent and inductive types

Dependent types are a feature of the type system of some programming languages that allows the type of an expression to depend on terms. This allows complex and nuanced relationships between terms and types to be expressed directly at the type level that would otherwise be difficult or impossible to express in traditional, non-dependent type systems.

The main advantage of dependent types is that they can be used to ensure that certain properties hold for specific terms. For example, a function that sorts a list can return, along with the sorted list, a proof that the list is sorted, and functions that require sorted lists can require a proof that the list is sorted. This allows functions to express their dependency on the property that the list is sorted directly on their types instead of, for example, spending precious time sorting the list by themselves or by expecting the programmer to informally uphold such a constraint[1].

[1]Relying on the programmer to uphold constraints about the data that they are manipulating is a frequent source of bugs. The most commonly exploited type of memory vulnerability expects the programmer to uphold the constraint that memory past an allocated region is never accessed irregularly. Another well known example of such an informally upheld constraint is to expect the programmer to never dereference null pointers. Tony Hoare calls his inclusion of null pointers in ALGOL W his "billion dollar mistake" (HOARE, 2009), as this preceded the inclusion of null pointers in many modern and still widely used

Besides requiring special treatment with regards to non-termination, dependent types have their own set of challenges: they require a more complex type system when compared to traditional type systems, which can make them harder to understand and reason about. Additionally, dependent types can introduce inefficiencies to the implementation of a programming language that are not as easy to optimize as it would be for a language that has a more traditional type system, which can make them less suitable for use in performance critical applications.

A concept that is closely related to dependent types is that of inductive types: data types that are defined in terms of their constructors, which are rules for building new elements of the type. They are similar to GADTs (generalized algebraic data types) found in Haskell but more powerful: whereas GADTs can be indexed only by types, inductive types are not restricted to being indexed only by types and can be indexed by both types and terms. Examples of inductive types include data structures (such as lists and trees) and mathematical objects (such as sets and relations).

Dependent types can be used in tandem with inductive types to reason about the type's properties at the type level. For example, in a language with dependent and inductive types, it is possible to express the concept of a matrix with a specific dimension, and ensure that operations on that matrix can only be performed if the dimension obeys some restrictions. This can help prevent certain types of errors, such as attempting to multiply two matrices whose columns from the first matrix do not match the rows from the second matrix.

In practice, dependent and inductive types are implemented in a variety of ways in different programming languages and proof assistants. Proof assistants like Coq (PAULIN-MOHRING, 2015), for example, are built specifically around dependent and inductive types and are used primarily for formal verification and proof assistance. Programming languages like Idris (BRADY, 2013), for example, include dependent and inductive types as an extension to a traditional functional type system. In the next sections, we elaborate on the usage of dependent and inductive types both in Coq and in Idris.

programming languages, which led to the believed amount of more than a billion dollars in damages to companies around the world related to errors caused by null pointers.

2.1.1 In proof assistants: the Coq example

Coq is a proof assistant that allows users to express mathematical definitions, functions and proofs in a precise and formal way. It is based on the Calculus of Inductive Constructions (CIC), a type theory that is expressive enough to encode a wide range of mathematical concepts, and that provides a powerful framework for reasoning about them. Coq's type system makes use of the Curry-Howard correspondence, which states that logical propositions and types in programming languages are equivalent. This means that propositions in Coq are represented as types, and proofs of these propositions are represented as programs. Coq has been used in a wide range of applications, including the verification of distributed algorithms (LESANI; BELL; CHLIPALA, 2016), programming languages (DUBOIS, 2000), security protocols (BARTHE; GRÉGOIRE; BÉGUELIN, 2009), and even the proof of the four color theorem (GONTHIER et al., 2008).

One of the key features of Coq is its ability to assist in the process of interactive proof development, which means that users can incrementally build their proofs, receiving feedback from the system as they go along. Not only this allows for a more natural and flexible proof development process, but also it provides the ability to verify the correctness of proofs mechanically. Its interactive proof environment is supported by a rich set of built-in tactics and automation tools that help users construct and check their proofs. These tactics can be used to perform basic proof steps, such as simplifying expressions or applying definitions, as well as more advanced proof techniques, such as induction and rewriting. The automation tools can help users find missing hypotheses and automatically generate proof scripts.

Like previously mentioned, Coq is based on the Calculus of Inductive Constructions, a type theory concerned with inductive types and their properties: it combines the calculus of constructions with an inductive definition mechanism, which allows for the definition of data types and (possibly recursive) functions that operate on top of said data types. For example, the natural numbers can be defined as an inductive type `nat` with two constructors: `zero` and `succ` (successor). `zero` is used to create the value `0`, and `succ` is used to create the value `n + 1` for any natural number `n`.

An important aspect of the Calculus of Inductive Constructions is the concept of **strict positivity** (also known as **well-foundedness**) which states that the arguments of a constructor can only refer to types in strictly positive occurrences in the sense that a constructor involves only strictly smaller values. A strictly positive occurrence of a type in a constructor argument is one where the type is only used in the form of a constructor application, with no negative occurrences such as being used in the argument of a function.

The `nat` type that we have discussed earlier is an example of a strictly positive type: the recursive occurrence of `nat` happens only in a positive position of the `succ` constructor, which ensures that the application of the `nat` argument to the `succ` constructor is strictly smaller than itself, making it so that a chain of `succ` constructors must always end with a `zero`. For example, trying to define a fixed-point operator, `infinite`, that evaluates to an infinite sequence of `succ` constructors results in the following error stating that the fixed-point operator does not recurse towards a smaller subterm of `nat`:

```
Fixpoint infinite (n: nat) : nat := 1 + infinite n.
```

```
Recursive definition of infinite is ill-formed.
In environment
infinite : nat -> nat
n : nat
Recursive call to infinite has principal argument equal to "n"
instead of a subterm of "n".
Recursive definition is: "fun n : nat => 1 + infinite n".
```

2.1.2 In programming languages: the Idris example

Idris is a general-purpose, pure, dependently typed functional programming language. Whereas Coq focuses on being a proof assistant, Idris shifts its focus completely to being a programming language. This means that Idris supports not only many functionalities commonly found in programming languages such as FFI (Foreign Function Interface, for interfacing with external code) and primitive, machine-mapped types, but also advanced features such as proof-carrying code due to dependent types being one of its capabilities.

As a general purpose programming language, Idris offers an interactive environment for type driven software development, but, just like Coq, it also has an interactive proof development environment. The interactive mode allows the programmer to write and test small parts of the program incrementally, which makes it easier to understand and fix errors.

The syntax of Idris is similar to that of Haskell. It uses a combination of expressions and patterns to define functions and data structures. For example, the following is a simple function that takes an integer and returns its double:

```
double : Int -> Int
double x = x * 2
```

Idris' type system is much more powerful than Haskell's, though, and allows for more ingenious relationships between types and terms to be expressed directly at the type level. For example, the length-indexed list, also known as `Vect`, can be defined in Idris as the following inductive type:

```
data Vect : Nat -> Type -> Type where
  Nil   : Vect Z a
  (::) : a -> Vect n a -> Vect (S n) a
```

The `Vect` type is a type constructor of arity 2, which means that before it can be considered a proper type, it needs to be applied to two arguments: a natural number `n` representing the length of the vector, and a type `a` representing the type of the elements in the vector. For example, the type of the list containing three boolean values can be written as `Vect 3 Bool`, and the type of the list containing four integer values can be written as `Vect (2 * 2) Int`. The `Vect` type has two constructors:

- `Nil`: which represents the empty vector and takes no elements;
- `(::)`: which takes an element of type `a` and a vector of length `n` and returns a vector of length `S n` (i.e. one larger than the original vector).

For example, the following is a `Vect` with three integers:

```
example : Vect 3 Int
example = 1 :: 2 :: 3 :: Nil
```

This is similar to the standard list type in Haskell, but Idris' `Vect` type has the length encoded in its type, which can be used for more powerful type checking and reasoning about the code. This can be used, for example, to ensure at compile time that the index passed to the `nth` function is within the range of the vector. Another example of a function that can take advantage of the vector's length being part of its type is `head`, a function that extracts the first element of a non-empty vector:

```
head : Vect (S n) a -> a
head (x :: xs) = x
```

In this function, `head` is defined to take a `Vect (S n) a` as an argument and to return an `a`. The type of the input vector is a `Vect` of length `S n`, where `n` is a natural number. By specifying `S n` in the type of the vector, we ensure that the function can only be applied to non-empty vectors (since `S n` will always be greater than `Z`). Here's an example of how we can use this function:

```
myList : Vect 3 Int
myList = 1 :: 2 :: 3 :: Nil

firstElement : Int
firstElement = head myList
```

In this case, `myList` is a vector of 3 integers and `firstElement` will be equal to the first element of the vector, which is `1`. If we try to use this function with an empty vector, the type checker will raise an error, since the type of the vector passed as argument is not compatible with the type of the function. This way, the function is type-safe: it will only work with non-empty vectors, and it cannot be misused.

```
myList2 : Vect Z Int
myList2 = Nil

-- This will raise a type error
-- during type checking
firstElement : Int
firstElement = head myList2
```

2.2 Programming for the web

The history of programming for the web dates back as far as the early days of the internet, when the World Wide Web was first proposed. The World Wide Web is a platform for sharing documents, and these documents can be linked to each other, allowing easy navigation between documents and access to a plethora of information.

The World Wide Web was accompanied by a markup language called HTML, also known as Hyper Text Markup Language, used to structure and format content on the web. HTML is used to specify not only the format of a document's content, such as lists, headings and paragraphs, but also the document's style, such as fonts, font sizes and colors. These HTML documents are downloaded and interpreted by a web browser, which would display the contents of the HTML document in a human-readable way.

In its inception, the World Wide Web was a tool meant to facilitate the sharing of knowledge, and thus, was predominantly geared towards scientists and researchers. As the annals of history attest, the World Wide Web progressively came to be adopted by a wider audience, and this eventually lead to its almost ubiquitous availability to the general public. The interconnected nature of the internet revealed its usefulness in not only disseminating information (which was its original purpose) but also reaching the population at large.

Such a hike in the popularity of the World Wide Web led to an increase in demand in interactivity from these documents. To meet these demands, PHP (also known as Hypertext Preprocessor) emerged. PHP is a language that allows a programmer to perform calculations, access databases and exchange data with the web browser, amongst other tasks. These capabalities allow PHP to dynamically generate HTML documents.

Another outcome from the demand for interactive HTML documents was JavaScript. While PHP was capable of dynamically generating HTML documents, interactivity still suffered due to the need for a server roundtrip before the HTML document could react to the user's input. Effectively, this meant that after a HTML document was dynamically generated by the server, once it was downloaded by the client, it became static, since the next request made by the client would replace the entire HTML document by a new one. To solve this deficiency, JavaScript was created and adopted by web browsers as a programming language on the client side.

JavaScript is a high-level, prototype-based object oriented programming language originally made to add scripting capabilities to otherwise static HTML documents. These scripting capabilities include reacting instantaneously to user input when validating user-submitted data, changing the appearance of the HTML document dynamically and updating isolated fragments of the HTML document. The level of interactivity offered by JavaScript revolutionized the landscape of web programming, as servers could now offload part of the processing tasks to the client, and the client could receive immediate feedback on its inputs.

As of today, JavaScript remains one of the most popular programming languages to ever exist. Not only is it available in all major web browsers, it also made its way to both the server in the form of Node.js (a backend runtime environment), and the desktop in the form of Electron (a desktop application container). But being the only programming language available for scripting a HTML document on the web browser also meant that JavaScript held a chokehold on its niche. If code were to be executed on the web browser, then it had to be written in JavaScript. Some programming languages, TypeScript (BIERMAN; ABADI; TORGERSEN, 2014) and Elm (CZAPLICKI, 2012) being two prominent examples, circumvented this limitation by transpiling their source code to JavaScript, but it the end, the result was still JavaScript. This approach was adequate for one portion of programming languages, but paradigms and priorities collided for another portion. Transpiling C (a low-level, systems programming language designed to be fast and efficient) to JavaScript incurs a hefty penalty on performance, simply due to the clash between the static and dynamic typing disciplines of the programming languages respectively.

With the purpose of being a means through which code written in programming languages other than JavaScript could be executed in the web browser, WebAssembly (ROSSBERG, 2022) was created and is being actively developed as an open standard by the W3C WebAssembly Working Group. Its first version was released in 2019 and since then, WebAssembly quickly became a widely supported standard for executing code on web browsers with near-native performance. WebAssembly is slated to be the next technology to revolutionize the web programming landscape, specially due to its versatility: WebAssembly has already brought many existing programming languages to the web programming landscape and it also has the potential to foster an ecosystem of entirely new programming languages designed specifically to take advantage of its benefits.

2.3 WebAssembly

WebAssembly (ROSSBERG, 2022) (often abbreviated as Wasm) is a low-level, portable binary format for executable code that runs primarily in web browsers, but other environments are also supported. Like previously mentioned, WebAssembly is an open standard being developed as a team effort by the W3C WebAssembly Community Group.

Its objective is to provide a fast, safe and portable way to run compiled code in the browser. It is intended to be used as a target for compilers of high-level languages like C, C++ and Rust, which allows programmers to build complex applications that can run in the browser without being limited by the performance and capabilities of JavaScript. This does not mean that WebAssembly replaces JavaScript. Instead, it means that WebAssembly can work together with JavaScript for tasks in which it can perform better.

One of the main advantages of WebAssembly is its **performance**. It is compiled to machine code, which means it can execute much faster than JavaScript, which is interpreted at runtime. This makes it particularly useful for tasks that require a lot of processing power, such as video and audio encoding, image manipulation, and 3D graphics rendering.

In addition to its performance, WebAssembly is also **safe**. A WebAssembly module runs in a memory-safe, sandboxed environment. This means that WebAssembly has built-in safeguards to prevent common types of vulnerabilities that involve the manipulation of memory. These vulnerabilities can occur when a program tries to access memory that it is not allowed to access, or when it tries to write to memory in a way that corrupts other data. This memory safety is ensured through a combination of static type checking, bounds checking and structured control flow: static type checking means that the types of variables are checked at compile time, bounds checking means that the program is checked to ensure that it is not trying to access memory outside of the bounds of an allocated memory region, and structured control flow restricts how the flow of control can branch.

Last but not least, WebAssembly is **portable**. Even though WebAssembly is primarily meant to be executed in the web browser, code that has been compiled to WebAssembly can be run on a wide range of platforms and environments without requiring any changes or modifications. This is due to the fact that WebAssembly has been designed to be a low-level, platform-agnostic format that can be executed by a virtual machine that is embedded in web browsers, although this virtual machine can also be a standalone engine, such as Wasmtime. This virtual machine is responsible for abstracting away the differences between the different environments where WebAssembly code can run.

2.3.1 Overview

WebAssembly is a formally specified **virtual instruction set architecture** (ROSSBERG, 2022). An instruction set architecture, or ISA, is an abstract specification of a computer's hardware and software that specifies how a computer should behave. It defines the set of available instructions that a processor can execute, and the format in which those instructions are encoded. A virtual ISA implements the instruction set at the software level instead of at the hardware level through a virtual machine. Not only does this allow WebAssembly programs to be written in a platform-agnostic manner, but it also is of paramount importance to the implementation of the safety features that WebAssembly is based upon.

WebAssembly's execution model is **stack-based** in the sense that the virtual machine maintains a stack of values, and instructions operate on values by pushing them onto the stack, performing operations on them, and then popping the results off the stack. In this way, the stack serves as a temporary storage area for values that are being operated on by the WebAssembly instructions. It allows the instructions to be composed without the need to explicitly specify the locations of all the operands.

A WebAssembly program is organized into a **module** which is structured as a sequence of **sections**, and the module and its sections as a whole represent a collection of types, tables, memories, globals and functions. The WebAssembly module can declare imports and exports as well, and initialize tables and memories with element and data segments respectively. All sections are optional, and an omitted section is equivalent to an empty section.

2.3.2 Examples

WebAssembly is, first and foremost, a binary format. The binary format is what will be consumed by web browsers and by standalone engines. One fortunate consequence of being a binary format is that the sizes of executable bundles that need to be downloaded by netizens using web browsers are drastically reduced when compared to

JavaScript bundles. But one unfortunate consequence of being a binary format is that it is not meant to be readable by humans. Fortunately, the WebAssembly specification encompasses not only a binary format but also a text format that is meant to be readable by humans, and the binary and text formats are engineered to be as close as possible, reaching an almost one-to-one resemblance.

In this section, some examples of WebAssembly programs in their text format will be shown to demonstrate their mechanics. It is important to remember that the canonical format of a WebAssembly module is the binary format: the text format allows for more flexibility in terms of readability and ease of writing which is very useful when handcrafting illustrative examples, but it is not a standard format and it is not intended for use in production.

The WebAssembly text format is composed of s-expressions. To briefly recapitulate, an s-expression (or symbolic expression) is a notation for representing tree-like data structures in a simple and human-readable format. It is commonly used in programming languages such as Lisp, Scheme, and Racket, and it is based on the idea of a parenthesized list. An s-expression is typically composed of atoms (such as numbers or symbols) and lists, which are enclosed in parentheses and separated by whitespace. For example, the s-expression `(+ 2 (* 3 4))` represents the mathematical expression $2+(3*4)$.

To showcase the capabilities of WebAssembly, let us look at how a simple addition function can be represented. To recapitulate, WebAssembly operates on top of a stack: instructions push and pop values on a stack to perform operations with the data.

```
(module
  (func $add (param $a i32) (param $b i32) (result i32)
    local.get $a
    local.get $b
    i32.add
  )
)
```

The definition begins by naming the function as `$add`. The function is declared to receive two 32-bit integers, `$a` and `$b`, and return a single 32-bit integer. The body of the function begins with the `local.get` instruction that pushes the argument `$a` into the stack. The argument `$b` is pushed to the stack using, once again, the `local.get` instruction. The body of the function ends with the `i32.add` instruction that pops the two values that were pushed to the stack, calculates their sum, and pushes the result to the stack. At this point, a single value is on the stack, which is what will be returned by the function.

In WebAssembly, all branches must be structured by enclosing `block` or `loop` instructions. To exemplify the restrictions that WebAssembly imposes on its control flow, let us implement a function, `$from`, that sums all integers from 1 to `n`.

```
(module
  (func $from (param $n i32) (result i32)
    (local $result i32)

    i32.const 0
    local.set $result

    block $break
      loop $loop
        local.get $n
        i32.eqz
        br_if $break

        local.get $result
        local.get $n
        i32.add
        local.set $result

        local.get $n
        i32.const 1
        i32.sub
        local.set $n

        br $loop
      end
    end

    local.get $result
  )
)
```

The function is declared to receive one 32-bit integer, `$n`, and return a single 32-bit integer. The body of the function declares one local variable, `$result`, which is initialized to the integer 0 in lines 5 and 6. Lines 8 through 26 are the essence of the function, and correspond to a `for` loop, where the `block` instruction delimits a forward jump, and the `loop` instruction delimits a backward jump.

Lines 10 through 12 begin the loop by checking if `$n` is equal to 0. If it is, a forward jump towards outside of the loop is performed. If it is not, execution continues past the `br_if` instruction. Lines 14 through 17 add the current value of `$n` to `$result`, and lines 19 through 22 subtract `1` from `$n`. The `br` instruction, in line 24, jumps back to the beginning of the loop, where the process can repeat.

After the loop is finished, the function simply puts the value of `$result` in the stack, and since the end of the function has been reached, the value is returned.

3 THE CURIOS LANGUAGE

In this chapter, a short overview of the Curios language is given to illustrate its syntax and its basic features, along with some examples of programs written using the language. Next, examples of advanced features that the language supports through the combination of its basic features are exemplified. To finalize, a tutorial on the usage of the Curios compiler toolchain is available.

3.1 Overview

Curios is a statically typed functional programming language that follows a strict evaluation model, with its primary compilation target being WebAssembly. Curios allows general recursion both at the term and type level and since general recursion allows the definition of non-terminating programs, its type system is inconsistent when viewed as a logic. Curios also supports full dependent types: no restrictions are imposed on how terms can occur at the type level. Curios' type system has three main defining constructs:

1. **Dependent function types.** Curios supports dependent function types where the output type of the function can depend on the value of its input. In other words, the type of the output of the function is not fixed but varies depending on the input value. Dependent function types work as the types of functions but they are also capable of expressing more advanced concepts such as parametric polymorphism and universal quantification. Dependent function types can also degrade to their regular, non-dependent counterparts when the variable bound by the dependent function type is not free in its output;
2. **Dependent ordered pair types.** Along with dependent function types, Curios also supports dependent ordered pair types where the type of the second entry of the pair can depend on the value of the first entry. Apart from being the type of ordered pairs, they can also be used for existential quantification, and in Curios, they are used alongside finite enumerations to express inductive types, which we will be discussing in depth in Section 3.4. Dependent ordered pair types, like dependent function types, can also degrade to their regular, non-dependent counterparts when the variable bound by the dependent ordered pair type standing for the first entry is not free in the second entry;

3. **Finite enumerations.** Last but not least, Curios supports finite enumerations of labels which are types that represent a set of enumerable labels, and a term inhabiting the finite enumeration is equal to one of the labels that the set contains. The term inhabiting the finite enumeration can be eliminated through a pattern matching operator that inspects the term to determine which label it is equal to and chooses the appropriate branch to continue from. Finite enumerations and their types are used to express simple branching control flow such as conditionals, and alongside dependent ordered pair types they can be used to represent inductive types, which we will be discussing in depth in Section 3.4.

3.2 Basic examples

Curios programs are structured as a sequence of global declarations and definitions. Each name being defined must have been previously declared, and previously declared names cannot be declared again. The syntax of a declaration is `x : A;`, where `x` is an identifier and `A` is a type. The syntax of a definition is `x = a;`, where `x` is an identifier and `a` is a term. The entry point of a Curios program is the definition named `start`, declared to have type `Int32`. The `start` definition fills a role similar to the `main` function of a C program: returning `0` means that the program finished successfully while other values signify different errors. Taking all of that into consideration, the smallest possible Curios program looks like the following:

```
start : Int32;
start = 0;
```

Curios functions are mechanically similar to the functions found in other functional programming languages such as Haskell and Idris where functions are curried: no more and no less than a single argument is bound by each function, and to represent functions of multiple arguments, single argument functions are chained sequentially. The process of applying an argument to such a function returns a new function that expects the next argument to be applied. This process repeats until all available arguments are applied. Curios follows an extrinsic typing discipline (also known as Curry-style) where the function's argument does not accompany a type annotation: the type annotation happens in the declaration of the function.

In Curios, the syntax for function types is `A -> B` where `A` and `B` are types, and the syntax for functions is `x => b`, where `x` is an identifier and `b` is a term. Both function types and functions enjoy some amount of syntactic sugar: the syntax for functions nests at the right-hand side of the term as in `a => b => c`, which is equivalent to `a => (b => c)`, and the syntax for function types also nest to the right-hand side of the term as in `A -> B -> C`, which is equivalent to `A -> (B -> C)`. As a demonstration, a simple function exemplifying the operation of adding an `Int32` to another `Int32` looks like the following:

```
add : Int32 -> Int32 -> Int32;
add = a => b => +i a b;
```

Like previously mentioned, functions in Curios are curried, and consequently, so are applications: the term `add 2 3` will first apply `2` to `add`, and then it will apply `3` to the function that results from the previous application. Another consequence of functions being curried is that functions can be partially applied: the term `add 1` inhabits the type `Int32 -> Int32`, and expects another `Int32` to be applied before a fully formed `Int32` can be returned. The syntax for applications can also be considered to enjoy some syntactic sugar: each individual application nests at the left-hand side of the term, and a term like `add 2 3` is equivalent to `(add 2) 3`.

Since Curios is pure, recursion is the main way through which iteration and repetition are expressed. Functions take advantage of the global context to be able to mention names recursively: declaring a name makes it available when it is being defined, and no restriction is imposed in how names in recursive positions occur. In the example below, a function that calculates the factorial of a number is declared and defined.

```
factorial : Int32 -> Int32;
factorial = n => if (<=i n 1) (1) (*i n (factorial (-i n 1)));
```

Curios also supports parametric polymorphism: a dependent function type can be used to bind a variable standing for some unknown generic type and functions can manipulate terms inhabiting the generic type abstractly. This can, for example, be used to define data structures that work as generic containers and functions that manipulate the generic container without knowing the exact representation of the container's element. An example of a generic function is the identity function:

```
identity : (A: Type) -> A -> A;
identity = A => a => a;
```

```
int32_one : Int32;
int32_one = identity Int32 1;
```

On top of the aforementioned constructs, Curios also supports a handful of primitive types, values and operators that correspond directly to their underlying WebAssembly equivalents. The previous examples showcased the primitive type `Int32`, primitive values such as `1` and primitive operators such as `+i` which stand, respectively, for the type of 32-bit integers, a value that inhabits the type of 32-bit integers and the addition operation applied to their values.

In the Curios type system, the primitive types `Int32` and `Flt32` correspond to the WebAssembly types `i32` and `f32` respectively. Values inhabiting the primitive type `Int32` are written in decimal notation with no fractional part (`123` or `-123`, for example), and values inhabiting the primitive type `Flt32` are written in decimal notation with a mandatory dot before the fractional part (`123.123` or `-123.123`, for example). Values inhabiting these primitive types can be manipulated by the customarily expected numeric operators, and a portion of these operators (the comparison and boolean operators) return a boolean `Int32` value (as opposed to a value that inhabits a dedicated boolean type) per the WebAssembly specification. Boolean `Int32` values can be tested with a short-circuiting `if` operator that has syntax `if a b c`, where `a` is the scrutinee of the operator, `b` is the result of the operator if the scrutinee is true and `c` is the result of the operator if the scrutinee is false. Apart from the `if` operator, the following numeric operators are available:

- `Int32` **arithmetic** operators, with type `Int32 -> Int32 -> Int32`: `+i`, `-i`, `*i`, `/i`;
- `Int32` **boolean** operators, with type `Int32 -> Int32 -> Int32`: `&&i`, `||i`;
- `Int32` **comparison** operators, with type `Int32 -> Int32 -> Int32`: `==i`, `/=i`, `<i`, `<=i`, `>i`, `>=i`;
- `Flt32` **arithmetic** operators, with type `Flt32 -> Flt32 -> Flt32`: `+f`, `-f`, `*f`, `/f`;
- `Flt32` **comparison** operators, with type `Flt32 -> Flt32 -> Int32`:

`==f`, `=/f`, `<f`, `<=f`, `>f`, `>=f`.

It is worth mentioning that, even though the primitive numeric operators and their usage may resemble functions and function application, they are not regular functions and, therefore, are not curried, cannot be applied to other functions and cannot be partially applied like regular functions can.

Curios also supports ordered pairs whose main purpose is to create structures of values that are grouped together. The syntax for ordered pair types is `A * B`, where `A` and `B` are types. The syntax for ordered pairs is `(a, b)`, where `a` and `b` are terms. For example, the declaration and definition of a pair that holds, as its first entry, an `Int32` and, as its second entry, a `Flt32` looks like the following:

```
numbers_pair : Int32 * Flt32;
numbers_pair = (1, 1.0);
```

To be able to access the values contained within an ordered pair, Curios provides a syntax in the form of `let (x, y) = a; b` where `x` and `y` are variables and `a` and `b` are terms that eliminates the ordered pair by binding each of its entries to a name called a *split expression*. The reason the split expression is used in lieu of the more familiar `fst` and `snd` is that when checking the type of `b`, the context is augmented with the fact that `a` is equal to `(x, y)`. Fret not, as `fst` and `snd` can still be implemented in terms of the split expression. For example, accessing the first entry of the ordered pair in the `numbers_pair` with a split expression example looks like the following:

```
first_number : Int32;
first_number = let (a, b) = numbers_pair; a;
```

Much like the syntax for function types and functions, the syntax for ordered pair types and ordered pairs also enjoy a minute amount of syntactic sugar. The syntax for ordered pair types nests at the right-hand side of the term as in `A * B * C` which is equivalent to `A * (B * C)`, and the syntax for ordered pairs also nests at the right-hand side of the term as in `(a, b, c)` which is equivalent to `(a, (b, c))`. For example, the declaration and definition of a nested ordered pair that holds three `Int32` values looks like the following:

```
nested_pair : Int32 * Int32 * Int32;
nested_pair = (1, 2, 3);
```

The eliminator for ordered pairs also supports nested ordered pairs, and works by sequentially destructuring the second entry of the nested ordered pair. For example, accessing the third number in the `nested_pair` example looks like the following:

```
third_number : Int32;
third_number = let (a, b, c) = nested_pair; c;
```

It is worth mentioning that nested ordered pairs can also be partially destructured, and the remainder of the nested ordered pair will be available as the last name bound by the eliminator. In the example below, the name `a` binds with `1`, and the name `b` binds with the second entry of the ordered pair, which is the pair `(2, 3)`.

```
partial_destr : Int32 * Int32;
partial_destr = let (a, b) = nested_pair; b;
```

Curios supports finite enumerations which are types that describe a collection of distinct, countable elements called *labels*. A term inhabiting the finite enumeration is equal to one of the labels contained in the finite enumeration. The syntax for finite enumerations is `{:l0, :l1, ..., :ln}` where each `li` is an identifier and the syntax for labels is `:a` where `a` is an identifier. For example, `{:one, :two, :three}` is a finite enumeration and `:two` is a label inhabiting this finite enumeration. Labels are used in conjunction with the finite enumeration they inhabit to carry out two central tasks:

1. They are scrutinized by a pattern matching operator, also known as *match expression*, that chooses the appropriate branch to continue from. Effectively, this is used to manipulate the program's flow of control;
2. When employed in conjunction with (dependent) ordered pair types, they are used to represent inductive types.

In a first moment, let us focus only on how finite enumerations and their labels can be used alongside the pattern matching operator to represent branching control flow. In Section 3.4 we show the role they play in the representation of inductive types. As an example, the definitions and declarations of a `Bool` type and a function that scrutinizes a value inhabiting the `Bool` type look like the following:

```
Bool : Type;
Bool = {:false, :true};
```

```
from_bool : Bool -> Int32;
from_bool = x =>
  match x {
    :false = 0;
    :true = 1;
  };
```

```
// The `converted_bool` definition evaluates to `0`
converted_bool : Int32;
converted_bool = from_bool :false;
```

3.3 Dependently typed examples

So far, we have discussed all of Curios' more basic functionalities but Curios also boasts an advanced type system that supports full dependent types, where terms are allowed to occur at the type level without any kind of restriction. In this section, the dependently typed capabilities of Curios are introduced through examples illustrating how terms can be used at the type level.

With that in mind, is it possible to define a type constructor that varies according to whether the argument it receives evaluates to false or true? The answer is yes! A declaration, definition and utilization of such a type looks like the following:

```
Wow : Bool -> Type;
Wow = x =>
  match x {
    :true = Int32;
    :false = Flt32;
  };
```

```
such_dependent : Wow :true;
such_dependent = 0;
```

This example already demonstrates some of the flexibility of Curios' type system, but there is still more ground to cover. What if we wanted the `Wow` type to vary according to the boolean variable bound by the function, **but in its own function type**? Let us revisit our old friend, the function type. We are yet to unlock its full power.

So far, we have seen only one example of a dependent function type, the type of the identity function. All other examples of function types were represented as their non-dependent counterparts, but in Curios, non-dependent function types are actually a special case of dependent function types where the variable bound by the function is not free in the output type i.e. the output type does not depend on the variable by the function. Dependent function types enable complex types to be expressed by allowing the type of the output of the function to depend on the variable bound for the input value. The syntax for dependent function types is `(a: A) -> B`, where `a` is a variable, `A` is a type and `B` is a type where `a` may occur free. All of the syntactic sugar that is available for non-dependent function types is also available to dependent function types. As an example, a declaration and definition of a function that is capable of returning terms inhabiting two different types based on its argument looks like the following:

```
even_more_dependent : (x: Bool) ->
  match x {
    :true = Int32;
    :false = Flt32;
  };

even_more_dependent = x =>
  match x {
    :true = 1;
    :false = 2.3;
  };
```

This example showcases computations occurring at the type level to manipulate types in such a way that terms inhabiting different types can be returned based on the value of the argument. Trying to return `2.3` from the `:true` branch would violate the constraint expressed in `even_more_dependent`'s declaration stating that only `Int32` values may be returned from the `:true` branch, ultimately resulting in a type error.

Just like function types, ordered pair types were another type that we have only seen through the lens of their non-dependent counterparts. And once again, just like function types, the non-dependent counterpart of ordered pair types is just a special case of the dependent ordered pair type where the variable bound as the value of the first entry is not free in the type of the second entry i.e. the type of the second entry does not depend on the value of the first entry. The syntax for dependent ordered pair types is `(a: A) * B`, where `a` is a variable, `A` is a type and `B` is a type where `a` may occur free. All of the syntactic sugar that is available for non-dependent ordered pair types is also available to dependent ordered pair types. As an example, a declaration and definition of an ordered pair that is capable of storing values inhabiting two different types as its second entry based on the first entry looks like the following:

```
increasingly_dependent : (x: Bool) *
  match x {
    :true = Int32;
    :false = Flt32;
  };

increasingly_dependent = (:false, 2.3);
```

This example showcases the versatility of dependent ordered pair types in storing different kinds of data while also obeying a constraint expressed at the type level. It would not be possible to define `increasingly_dependent` as an ordered pair storing both `:true` and `2.3` because that would violate the constraint expressed in its declaration stating that only `Int32` values may be stored alongside a `:true`, ultimately resulting in a type error.

The somewhat artificial code snippets in this section only scratch the surface of all the possibilities that dependent types bring to the table. We have only seen the tip of the iceberg, go wild!

3.4 Inductive types

Functional programming languages usually offer data types as one of the native features of their type systems. Languages like Haskell and Idris introduce a new data type to their type systems by asking the user to describe the data that the type represents in the form of constructor applications, and this description is what allows the language to construct values of the type while also allowing the type to participate in other mechanisms of the language such as pattern matching. Curios takes a different approach: instead of offering native data types, Curios uses a combination of its more elementary components to introduce new data types to the system as **encodings**.

The most noticeable difference between encoded data types and native data types is the syntax: encoded data types tend to be much more verbose than native data types. This is because the individual components that compose an encoded data type each have a dedicated syntax (since each component has a specific purpose and can be used separately) and require the full syntax to be spelled out to achieve the same effect of the more concise native data type. The main advantage that encoded data types have over native data types is simplicity: it is much easier to understand each individual component of the encoded data type than it is to understand the entirety of a native data type system, and this situation is similar with regards to the implementation of a type checker for a type system that offers encoded data types versus one that offers native data types.

In a first moment, it was chosen for Curios to offer only encoded data types given the fact that they are simpler to understand and easier to implement. This does not mean that Curios' type system is any less expressive though, as we will see in this and the following sections that many high-level functional programming language features can be expressed through some combination of Curios' features. One of the more important plans for Curios' future is the development of a high-level syntax that desugars into the encoding, which would allow Curios to have the best of both worlds: a high-level, concise syntax for introducing new data types to the type system while also retaining the original orthogonality of the individual components.

3.4.1 Simple inductive types

Inductive types in Curios are represented through a dependent ordered pair type where the type of the first entry is a finite enumeration and the type of the second entry varies according to the value stored in the first entry. With that in mind, let us take a closer look into how we can use the tools that Curios offers to encode the type standing for the natural numbers, and a simple addition operation on top of this type.

```
Unit : Type;
Unit = {:unit};

unit : Unit;
unit = :unit;

Nat : Type;
Nat =
  (l: {:zero, :succ}) * match l {
    :zero = Unit;
    :succ = Nat;
  };

zero : Nat;
zero = (:zero, unit);

succ : Nat -> Nat;
succ = a => (:succ, a);

add : Nat -> Nat -> Nat;
add = a => b =>
  let (al, a_) = a;

  match al {
    :zero = b;
    :succ = succ (add a_ b);
  };
```

Whoa, that looks complicated. Let us look at the example line by line, starting with `Unit : Type;`. This is a **declaration** stating that `Unit` is a `Type`. The very next line, `Unit = {:unit};`, **defines** `Unit` to be a **finite enumeration** containing a single label, `:unit`. A helper constructor is also declared (`unit : Unit;`) and defined (`unit = :unit;`). So far, so good.

Moving on, we have the **declaration** of `Nat` in line 7. Lines 8 through 12 have the definition of `Nat` as a dependent ordered pair type. The left side of the dependent ordered pair type introduces the variable `l` as inhabiting the finite enumeration `{:zero, :succ}`. The type in the right side of the dependent ordered pair type depends on the variable `l`: if `l` matches with `:zero`, this type is `Unit`, and if it matches with `:succ`, this type is `Nat`. Just like our `Unit` type, we also declare and define the helper constructors `zero` and `succ` for our `Nat` type in lines 14 through 18.

Before we move on to `add`, let us first take a closer look at the inner workings of our `zero` and `succ` constructors. If we check `zero`, which is the ordered pair `(:zero, unit)`, against `Nat`, which is a dependent ordered pair type composed of the finite enumeration `{:zero, :succ}` and the match expression, it is possible to see that the label `:zero` indeed inhabits the finite enumeration `{:zero, :succ}`. But how does `unit` inhabit the match expression at the right side of the dependent ordered pair type? Because the first entry of the ordered pair is `:zero`, the type of the second entry of the ordered pair (the match expression) is able to reduce further, exposing the fact that the second entry of the ordered pair should have type `Unit`. And it does! Mission accomplished. In our `succ` constructor, defined in lines 17 and 18 to be the ordered pair `(:succ, n)` the same concept applies: since the first entry of the ordered pair is `:succ`, the match expression in the type of the second entry of the ordered pair reduces to `Nat`.

But how do we work with data encoded this way? Let us look at `add`. We start with its declaration in line 20, followed by its definition in lines 21 through 27. Line 22 of the definition binds a name for each of the ordered pair's entries, and in lines 24 through 27 the first entry of the pair is matched against its possible values.

Notice how the pattern matching operation is performed on `al` and `a_` is only mentioned when `al` matches with `:succ`. This is because if `al` matched with `:zero`, then `a_` would have type `Unit`, as its (dependent ordered pair) type suggests. This is the key insight to how Curios encodes its inductive types: when performing pattern matching, the context is augmented with the constraint that `al` is equal to `:zero` and `:succ` in each respective branch, and the type rules exploit this information to assign the correct type to `a_`.

3.4.2 Indexed inductive types

The `Nat` type is a very simple inductive type that is not indexed by any types or terms, but Curios' inductive types can be indexed by both types and terms. For example, let us consider how to represent the length-indexed list in Curios, which is indexed by both a term and a type.

```
1 Vect : Nat -> Type -> Type;
2 Vect = length => T =>
3   let (ll, l_) = length;
4
5   match ll {
6     :zero = Unit;
7     :succ = T * Vect l_ T;
8   };
```

The `Vect` type is the poster child of dependently typed languages, and it remains so in Curios. This type represents a list whose length is part of its type, which means that it becomes possible to reason about the length of the list directly at the type level. Despite being a fairly straightforward type, it demonstrates a lot of the power that dependently typed languages can wield.

Let's take a closer look at the `Vect` type. Its declaration, in line 1, states that it depends on a `Nat`, standing for the length of the list, and a `Type`, standing for the type of the element that the list holds. Its definition spans lines 2 through 8 and begins in line 3 by binding a variable for each of the length's entries. The definition proceeds, in line 5, to match on the label of the length. Line 6 states that if the label of the length matches with `:zero`, then the definition is `Unit`, representing the fact that the list holds no elements. If the label of the length matches with `:succ`, then the definition is an ordered pair type holding the element at the head of the list and the rest of the list.

```
1 head : (length: Nat) -> (T: Type) -> Vect (succ length) T -> T;
2 head = length => T => vect =>
3   let (x, xs) = vect;
4   x;
```

The `head` operation gives an idea of how we can use the length of the list at the type level to extract the element at the head of a non empty list. Most importantly, how is it possible to determine that the list is not empty exclusively through its type? Consider the type of the list, `Vect (succ length) T`. More specifically, consider its length, `succ length`. This length statically ensures that the `Vect` must hold one or more elements, which means that it is statically safe to extract the element at the head. The definition of `head` does exactly that: since the length of the `Vect` statically ensures that the `Vect` holds one or more elements, it is possible to bind one variable for the element at the head of the list and one variable for the rest of the list. At this point, the element at the head is simply returned.

The careful reader might have spotted that there seems to be a disconnect between the way the length-indexed list is represented in Curios and how it is represented in other dependently typed languages such as Idris. This observation is correct: in this example, Curios' length-indexed list is represented through induction over its length.

`Vect` is an example of a type that can be encoded this way, but this encoding is not enough in the general case, since many inductive types cannot easily be encoded this way. For example, if we want to define an inductive type `IsEven` standing for the proposition that a natural number is even as the remainder of the division of the natural number by two being equal to zero, an explicit equality constraint is necessary. To remedy this shortcoming, we will be looking into how to define a propositional equality type in Curios, and how an equality type can be used to represent both the `IsEven` proposition and the `Vect` type through equality constraints.

3.5 Equality

Curios also has the ability to perform large eliminations (ROUX, 2023a; ROUX, 2023b) i.e. to construct types by eliminating terms. This ability will be used alongside a procedure for determining if two values are structurally equal to define a propositional equality type, which will allow us to equip Curios' inductive types with equality constraints. To more easily illustrate the concept, we will specialize our example to `Nat`, the type standing for natural numbers.

```
eq : Nat -> Nat -> Bool;
eq = a => b =>
```

```
  let (al, a_) = a;
  let (bl, b_) = b;

  match al {
    :zero = match bl {
      :zero = :true;
      :succ = :false;
    };
    :succ = match bl {
      :zero = :false;
      :succ = eq a_ b_;
    };
  };
```

```
Empty : Type;
Empty = {};
```

```
Eq : Nat -> Nat -> Type;
Eq = a = > b = >
  match eq a b {
    :true = Unit;
    :false = Empty;
  };
```

The equality type `Eq` uses the structural equality function `eq` to lift the comparison to the type level. If the values are structurally equal, then `Eq` is equivalent to `Unit`, expressing the fact that the equality holds and can be instantiated with `unit`. But if the values are not structually equal, then `Eq` is equivalent to `Empty`, expressing the fact that the equality does not hold and cannot be instantiated. Assuming that `rem` is a function that calculates the remainder of dividing a natural number by another natural number, we are now able to define `IsEven`.

```
two : Nat;
two = succ (succ zero);
```

```
IsEven : Nat -> Type;
IsEven = a = Eq (rem a two) zero;
```

Even though we can represent the `Vect` type through induction over its length, the `Vect` type can be rewritten to use equality constraints instead which allows us to reason about its structure using equalities as opposed to pattern matching on its length directly.

```
Vect : Nat -> Type -> Type;
Vect = len => T =>
  (l: {:null, :cons}) * match l {
    :null = Eq len zero;
    :cons = (vlen: Nat) * T * Vect vlen T * Eq len (succ vlen);
  };
```

The `head` function also needs to be rewritten accordingly, since it needs to deal with an absurd case: pattern matching on the structure of the vector reveals a case where the list is empty even though its type clearly suggests that it should not be, but we can readily dismiss that case through the usage of `absurd` in conjunction with the (absurd) proof that the length of the list is equal to zero in `v_`.

If the label of the list matches with `:null`, then `v_` is a proof that the list's length is equal to zero, which is false and can be dismissed since the type of the list that the `head` function accepts expresses the fact that the list contains at least one element. The case where the label of the list matches with `:cons` is what the `head` function is interested in, within which `v_` is a nested ordered pair that includes the element at the head which is promptly returned.

```
absurd : Empty -> (A: Type) -> A;
absurd = e => A => match e {};
```

```
head : (len: Nat) -> (T: Type) -> Vect (succ len) T -> T;
head = len => T => vect =>
  let (vl, v_) = vect;

  match vl {
    :null = absurd v_ T;
    :cons = let (vlen, x, xs, p) = v_; x;
  };
```

3.5.1 Proofs

We can use the `Eq` type to represent equalities between terms at the type level, but we can also reason abstractly about the equality itself. For example, we can show that equality is reflexive by induction on the natural number.

```
refl : (a: Nat) -> Eq a a;
refl = a =>
  let (al, a_) = a;

  match al {
    :zero = unit;
    :succ = refl a_;
  };
```

After having shown that equality is reflexive with `refl`, we can show that equality is also substitutive with `subst`. This proof employs `absurd`: even though we have an instance of `Eq a b` proving that the natural numbers `a` and `b` are equal, the process of pattern matching on the structure of `a` and `b` reveals the fact that there may be some cases where they are not equal, but we can readily dismiss those cases through the usage of `absurd` in conjunction with the proof of `Eq a b`.

```
subst : (a: Nat) -> (b: Nat) -> Eq a b
  -> (P: Nat -> Type) -> P a -> P b;

subst = a => b => eq_a_b => P => p_a =>
  let (al, a_) = a;
  let (bl, b_) = b;

  match al {
    :zero = match bl {
      :zero = match a_ { :unit => match b_ { :unit => p_a; }; };
      :succ = absurd eq_a_b (P b);
    };
    :succ = match bl {
      :zero = absurd eq_a_b (P b);
      :succ = subst a_ b_ eq_a_b (x => P (succ x)) p_a;
    };
  };
```

After having shown that equality is both reflexive and substitutive, we can also show that equality is both symmetric and transitive through its reflexive and substitutive properties with `sym` and `trans`. These proofs simply use the proof of substitutivity to rearrange the equalities.

```
sym : (a: Nat) -> (b: Nat) -> Eq a b -> Eq b a;
sym = a => b => eq_a_b =>
  subst a b eq_a_b (x => Eq x a) (refl a);
```

```
trans : (a: Nat) -> (b: Nat) -> (c: Nat) ->
  Eq a b -> Eq b c -> Eq a c;

trans = a => b => c => eq_a_b => eq_b_c =>
  subst b c eq_b_c (x => Eq a x) eq_a_b;
```

It is important for the reader to remember that, while Curios can potentially support theorem proving, Curios is **partial**: not only is general recursion allowed but also the type of `Type` is `Type`, and both of them lead to inconsistency. In these small examples, it is easy to see that these proofs are total, but this might not be the case in larger proofs. Viewer discretion is advised. A separate termination checker with a cumulative hierarchy of universes for the subset of Curios programs that obey well-founded recursion is envisioned to be developed in the future.

4 THE CURIOS SPECIFICATION

In this chapter, we introduce Curios' formal abstract syntax, its type rules and its operational semantics. The focus is given mainly to dependent function types, dependent ordered pair types and finite enumerations: we refer the reader to the WebAssembly specification (ROSSBERG, 2022) for the type rules and semantics of the WebAssembly constructs that have counterparts in Curios.

4.1 Abstract syntax

The abstract syntax of Curios terms is given in Figure 4.1. The specification of the abstract syntax begins with Variables and Labels which are two distinct sets of identifiers that play specific roles within the abstract syntax: variables stand for unknown terms in the bodies of function types, functions, pair types and split expressions while labels are inhabitants of finite enumerations and are scrutinized by match expressions.

Next, we have Terms, which encompass variables, the type of all types, dependent function types, functions, applications, dependent ordered pair types, ordered pairs, split expressions, finite enumerations, labels and match expressions. The syntax specified for some terms differs slightly from what is usually expected ($Type$ instead of $*$, $(x : A) \to B$ instead of $\Pi\, x : A.\ B$, $x \Rightarrow b$ instead of $\lambda x.\ b$ and $(x : A) \times B$ instead of $\Sigma x : A.\ B$) and the motivation for this alternative syntax is that it more closely resembles the concrete syntax that is given in Chapter 3. The overline in finite enumerations and match expressions stands for zero or more occurrences.

One import detail to note is that a different metavariable is used when a term plays the role of a type (i.e. a versus A), but there is no distinction between terms and types: they both belong to the same syntactic category. Additionally, $(x : A) \to B$ can be abbreviated

Figure 4.1 – Abstract syntax of Curios terms.

$$x,\ y \in \mathit{Variable} \qquad l \in \mathit{Label}$$

$$\begin{array}{llcllllll}
\mathit{Term} & a,\ b,\ f,\ A,\ B & ::= & x & \mid & \mathit{Type} & & \\
 & & \mid & (x : A) \to B & \mid & x \Rightarrow b & \mid & f\ a \\
 & & \mid & (x : A) \times B & \mid & (a,\ b) & \mid & \mathit{let}\ (x,\ y) = a;\ b \\
 & & \mid & \{\overline{:l}\} & \mid & :l_i & \mid & \mathit{match}\ a\ \{\overline{:l = b;}\}
\end{array}$$

Figure 4.2 – Abstract syntax of Curios typing contexts.

$$x \in \mathit{Variable} \quad a,\, A \in \mathit{Term}$$

$$\begin{array}{llll} \mathit{Context} & \Gamma & ::= & \varnothing \\ & & \mid & \Gamma;\ x : A \\ & & \mid & \Gamma;\ x = a \end{array}$$

Figure 4.3 – Context declaration lookup – $x : A \ \in \ \Gamma$.

$$\frac{}{x : A \ \in \ (\Gamma;\ x : A)} \qquad \frac{x : A \ \in \ \Gamma \quad x \neq y}{x : A \ \in \ (\Gamma;\ y : B)} \qquad \frac{x : A \ \in \ \Gamma \quad x \neq y}{x : A \ \in \ (\Gamma;\ y = b)}$$

to $A \to B$ when x is not free in B, and $(x : A) \times B$ can be abbreviated to $A \times B$ when x is not free in B.

4.2 Typing rules

In Figure 4.2 we introduce the syntax for Curios' typing Contexts which can either be empty, or a context extended with a new declaration of a name associated to a type, or a context extended with a new definition of a name associated to a term. Contexts are used as the representation of a Curios program and they are also used in tandem with Curios' type rules to keep track of the declarations and definitions of variables introduced by dependent function types, functions, dependent pair types, split expressions and match expressions.

Before we can define Curios' type system, we need to first define $x : A \ \in \ \Gamma$, which stands for the lookup of a declaration in the context. This derivation works through the weakening of the context: we can *throw out* declarations and definitions until we find the declaration we are looking for. The rules are available in Figure 4.3.

With the definition for context declaration lookup, we can proceed to define $\Gamma \vdash a : A$ which stands for a typing judgement stating that the term a inhabits the type A in the context Γ. The rules for the typing judgements are available in Figure 4.4. Some rules (namely, the *app* and *pair* rules) mention a substitution operation which will be detailed in Section 4.3. The first rule, named *conv*, applies to all terms and employs the β-equality relation (which will be detailed in Section 4.4) to allow evaluation at the type

Figure 4.4 – Typing rules for typing judgements – $\Gamma \vdash a : A$.

$$\frac{\begin{array}{c}\Gamma \vdash a : A \\ \Gamma \vdash A =_\beta B\end{array}}{\Gamma \vdash a : B}\ \mathit{conv} \qquad \frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ \mathit{var} \qquad \frac{}{\Gamma \vdash \mathit{Type} : \mathit{Type}}\ \mathit{type}$$

$$\frac{\begin{array}{c}\Gamma \vdash A : \mathit{Type} \\ \Gamma;\ x : A \vdash B : \mathit{Type}\end{array}}{\Gamma \vdash (x : A) \to B : \mathit{Type}}\ \mathit{pi} \qquad \frac{\begin{array}{c}\Gamma \vdash (x : A) \to B : \mathit{Type} \\ \Gamma;\ x : A \vdash b : B\end{array}}{\Gamma \vdash x \Rightarrow b : (x : A) \to B}\ \mathit{abs}$$

$$\frac{\begin{array}{c}\Gamma \vdash (x : A) \to B : \mathit{Type} \\ \Gamma \vdash f : (x : A) \to B \\ \Gamma \vdash a : A\end{array}}{\Gamma \vdash f\ a : B\,[x := a]}\ \mathit{app} \qquad \frac{\begin{array}{c}\Gamma \vdash A : \mathit{Type} \\ \Gamma;\ x : A \vdash B : \mathit{Type}\end{array}}{\Gamma \vdash (x : A) \times B : \mathit{Type}}\ \mathit{sigma}$$

$$\frac{\Gamma \vdash (x : A) \times B : \mathit{Type} \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : B\,[x := a]}{\Gamma \vdash (a,\ b) : (x : A) \times B}\ \mathit{pair}$$

$$\frac{\begin{array}{c}\Gamma \vdash (x : A) \times B : \mathit{Type} \\ \Gamma \vdash a : (x : A) \times B \\ \Gamma \vdash T : \mathit{Type}\end{array} \qquad \begin{array}{c}\Gamma \vdash a =_\beta z \\ \Gamma;\ x : A;\ y : B;\ z = (x,\ y) \vdash b : T\end{array}}{\Gamma \vdash \mathit{let}\ (x,\ y) = a;\ b : T}\ \mathit{split}$$

$$\frac{\begin{array}{c}\Gamma \vdash (x : A) \times B : \mathit{Type} \\ \Gamma \vdash a : (x : A) \times B\end{array} \qquad \begin{array}{c}\Gamma \vdash T : \mathit{Type} \\ \Gamma;\ x : A;\ y : B \vdash b : T\end{array}}{\Gamma \vdash \mathit{let}\ (x,\ y) = a;\ b : T}\ \mathit{split}^{\mathit{alt}}$$

$$\frac{}{\Gamma \vdash \{\overline{:l}\} : \mathit{Type}}\ \mathit{enum} \qquad \frac{\Gamma \vdash \{\overline{:l}\} : \mathit{Type} \qquad :m \in \overline{:l}}{\Gamma \vdash :m : \{\overline{:l}\}}\ \mathit{label}$$

$$\frac{\begin{array}{c}\Gamma \vdash \{\overline{:l}\} : \mathit{Type} \\ \Gamma \vdash a : \{\overline{:l}\}\end{array} \qquad \begin{array}{c}\Gamma \vdash T : \mathit{Type} \\ \Gamma \vdash a =_\beta x\end{array} \qquad (\Gamma;\ x = :l_i \vdash b_i : T)_i}{\Gamma \vdash \mathit{match}\ a\ \{\overline{:l = b;}\} : T}\ \mathit{match}$$

$$\frac{\Gamma \vdash \{\overline{:l}\} : \mathit{Type} \qquad \Gamma \vdash a : \{\overline{:l}\} \qquad \Gamma \vdash T : \mathit{Type} \qquad (\Gamma \vdash b_i : T)_i}{\Gamma \vdash \mathit{match}\ a\ \{\overline{:l = b;}\} : T}\ \mathit{match}^{\mathit{alt}}$$

level. Following the *conv* rule, there is one rule for each of the syntactic constructs of the abstract syntax introduced in Figure 4.1:

- The *var* rule ascribes the type A to a variable x by looking its declaration up in the context;
- The *type* rule states that the type of $Type$ is itself;
- The *pi* rule states that a dependent function type $(x : A) \to B$ is a type if its input type A is a type (first premise) and if its output type B is a type in the context extended with a variable x declared to inhabit the input type A (second premise);
- The *abs* states that a function $x \Rightarrow b$ inhabits a dependent function type $(x : A) \to B$ if the dependent function type is a type (first premise) and if the output b of the function can be typed by the output type B of the dependent function type in the context extended with a declaration of the variable x inhabiting the input type A of the dependent function type (second premise);
- The *app* rules states that an application $f\ a$ inhabits the type B with its free variable x substituted for the application's argument a if the dependent function type $(x : A) \to B$ is a type (first premise), if the application's function f inhabits the dependent function type (second premise) and if the application's argument a inhabits the input type A of the dependent function type (third premise);
- The *sigma* rule states that a dependent ordered pair type $(x : A) \times B$ is a type if the first entry type A is a type (first premise) and if the second entry type B is a type in the context extended with a variable x declared to inhabit the first entry type (second premise);
- The *pair* rule states that an ordered pair $(a,\ b)$ inhabits a dependent ordered pair type $(x : A) \times B$ if the dependent ordered pair type is a type (first premise), if the first entry a matches the first entry type A (second premise) and if the second entry b matches the second entry type B with its free variable x substituted for the first entry (third premise);
- The *split* rule states that a split expression $let\ (x,\ y) = a;\ b$ inhabits a type T if the dependent ordered pair type $(x : A) \times B$ is a type (first premise), if the scrutinee a of the split expression inhabits the dependent ordered pair type (second premise), if the type T is a type (third premise), if the scrutinee of the split expression reduces to a variable z (fourth premise) and if the output b of the split expression inhabits the type T in a context extended with the declarations of x and y as inhabiting A

and B respectively and the definition of z as being equal to the ordered pair (fifth premise);

- The $split^{alt}$ rule is a version of the $split$ rule without dependent elimination;
- The *enum* rule states that a finite enumeration $\{\overline{:l}\}$ is a type;
- The *label* rule states that a label $:l_i$ inhabits a finite enumeration $\{\overline{:l}\}$ if the finite enumeration is a type (first premise) and if the label is an element of the finite enumeration (second premise);
- The $match$ rule states that a match expression $match\ a\ \{\overline{:l = b;}\}$ inhabits a type T if the finite enumeration $\{\overline{:l}\}$ is a type (first premise), if the split expression's scrutinee a inhabits the finite enumeration (second premise), if the type T is a type (third premise), if the split expression's scrutinee reduces to a variable x (fourth premise) and if every branch b_i of the match expression inhabits the type T in a context extended with the definition of x as being equal to the label $:l_i$ of each branch (fifth premise);
- The $match^{alt}$ rule is a version of the $match$ rule without dependent elimination.

Except for the unorthodox syntax and for the $split$ and $match$ rules, all of the aforementioned rules are fairly standard in type theory (specially dependent function types and dependent ordered pair types, which are more commonly known as Π-types and Σ-types respectively in type theory literature). More specifically, the premise that the scrutinees of the $split$ and $match$ rules should reduce to a variable can be seen as unusual but they are very important: by extending the context with a definition equating the relevant scrutinee to its corresponding pattern in the output of split expressions and in each branch of match expressions, the system is capable of performing the dependent elimination (GOGUEN; MCBRIDE; MCKINNA, 2006; COQUAND, 1992; ROUX, 2023b) of ordered pairs and labels (as opposed to the non-dependent elimination performed by primitive recursors where the context does not store knowledge regarding the pattern being eliminated). To better understand why those premises add significant expressiveness to Curios' type system, let us look at two examples that would not type check without those premises.

The first example involves existential quantification: to be able to retrieve the witness of an existential proposition at the term level, we need to bring into action a split expression at the type level whose responsibility is to eliminate the existential quantification and retrieve the proposition that it contains. At the term level, the split expression binds a name for the proposition and the witness while also augmenting the context with equality information which allows the split expression at the type level to reduce and substantiate the fact that that the witness inhabits its proposition.

```
exists : (x: (P: Type) * P) -> let (P, p) = x; P;
exists = x => let (P, p) = x; p;
```

The `exists` function would not type check if it was not for the premise that the scrutinee of the split expression `x` is equated to its corresponding pattern `(P, p)` in the output of the split expression because it would not be possible to reduce the split expression at the type level and see that the witness inhabits its proposition.

The second example involves disjoint unions: we illustrate the problem with the type `Foo` which is defined as a dependent ordered pair type where the type of the first entry is `Bool` and the type of the second entry varies according to the `Bool` stored in the first entry.

```
Foo : Type;
Foo =
  (x: Bool) * match x {
    :true = Int32;
    :false = Flt32;
  };
```

What happens if we try to eliminate an ordered pair inhabiting `Foo` and use its second entry in some way? Let us write a function that eliminates terms inhabiting `Foo` and checks whether their second entries are greater than zero.

```
greater_than_zero : Foo -> Bool;
greater_than_zero = a =>
  let (al, a_) = a;

  match al {
    :true = if (>i 0 a_) :true :false;
    :false = if (>f 0.0 a_) :true :false;
  };
```

The `greater_than_zero` function would not type check if it was not for the premise that the scrutinee of the match expression `al` is equated to the patterns `:true`

Figure 4.5 – Context formation – $\Gamma\ ok$.

$$\frac{}{\varnothing\ ok} \qquad \frac{x \notin dom(\Gamma) \qquad \Gamma \vdash A : Type}{\Gamma;\ x : A\ ok} \qquad \frac{\Gamma \vdash x : A \qquad \Gamma \vdash a : A}{\Gamma;\ x = a\ ok}$$

Figure 4.6 – Substitution operation – $m\ [z := n]$.

$$\begin{array}{rcl}
(x)\ [z := n] & = & if\ z == x\ then\ n\ else\ x \\
(Type)\ [z := n] & = & Type \\
((x : A) \to B)\ [z := n] & = & (x : A\ [z := n]) \to B\ [z := n] \\
(x \Rightarrow b)\ [z := n] & = & x \Rightarrow (b\ [z := n]) \\
(f\ a)\ [z := n] & = & (f\ [z := n])\ (a\ [z := n]) \\
((x : A) \times B)\ [z := n] & = & (x : A\ [z := n]) \times B\ [z := n] \\
(a,\ b)\ [z := n] & = & (a\ [z := n],\ b\ [z := n]) \\
(let\ (x,\ y) = a;\ b)\ [z := n] & = & let\ (x,\ y) = a\ [z := n];\ b\ [z := n] \\
(\{\overline{:l}\})\ [z := n] & = & \{\overline{:l}\} \\
(:l_i)\ [z := n] & = & :l_i \\
(match\ a\ \{\overline{:l = b;}\})\ [z := n] & = & match\ a\ \{\overline{:l = b\ [z := n];}\}
\end{array}$$

and `:false` in each corresponding branch of the match expression because it would not be possible to reduce the match expression in the type of `a_` and see that, in the `:true` branch, the type of `a_` reduces to `Int32` and that, in the `:false` branch, the type of `a_` reduces to `Flt32`.

To finalize, we introduce, in Figure 4.5, the rules for context formation: Γ *ok* means that the context Γ is valid. In summary, the rules state that extending a context with a declaration requires the type to be well-formed, and extending the context with a definition additionally requires the name to be previously declared.

4.3 Substitution

This section defines, in Figure 4.6, the substitution operation. It is written $m\ [z := n]$ and stands for replacing all free occurrences of the variable z within the term m with the term n. Substitution is an essential operation for type systems that include dependent types in view of the fact that types may mention variables standing for arbitrary terms. Substitution is also at the heart of the β-equality relation (which will be detailed in Section 4.4) that specifies when two possibly distinct terms can be considered equivalent. The

Figure 4.7 – Context definition lookup – $x = a \in \Gamma$.

$$\frac{}{x = a \in (\Gamma;\, x = a)} \qquad \frac{x = a \in \Gamma \quad x \neq y}{x = a \in (\Gamma;\, y : B)} \qquad \frac{x = a \in \Gamma \quad x \neq y}{x = a \in (\Gamma;\, y = b)}$$

substitution operation is assumed to be capture-avoiding.

4.4 β-equality

This section defines the notion of β-equality, which is an essential component of the *conv* rule introduced in Figure 4.4. β-equality (also known as β-conversion) is the mechanism through which a simplified term can be obtained from a term that contains redexes. We begin by defining $x = a \in \Gamma$ which stands for the lookup of a definition in the context. It is derivable if the definition $x = a$ is visible in the context Γ. This rule works very similarly to how $x : A \in \Gamma$ works: the context is weakened until we have found the relevant definition. The rules are available in Figure 4.7.

Concluding the specification of Curios' type system, we define the notion of β-equality as a congruence relation equating variables to their definitions and equating the redexes formed by the constructors and eliminators of functions, ordered pairs and labels to their contractum. The rules are available in Figure 4.8 and can be summarized as follows:

- Rules (1) through (3) represent the reflexivity, symmetry and transitivity properties of the relation;
- Rules (4) through (10) propagate the β-equality relation towards subterms of the relevant terms;
- Rule (11) states that any two α-equal terms are also β-equal;
- Rule (12) states that a variable x is β-equal to a term a if the definition of x as being equal to a is visible in the context;
- Rule (13) states that the application of a term $x \Rightarrow b$ to a term a is β-equal to the term b with its free variable x substituted for the term a;
- Rule (14) states that a match expression scrutinizing a label $:l_i$ and containing the branches $\overline{:l = b;}$ is β-equal to the branch b_i of the match expression;
- Rule (15) states that a split expression binding the variables x and y for the ordered

Figure 4.8 – β-equality – $\Gamma \vdash a =_\beta b$.

$$\frac{}{\Gamma \vdash a =_\beta a}\ (1) \qquad \frac{\Gamma \vdash a =_\beta b}{\Gamma \vdash b =_\beta a}\ (2)$$

$$\frac{\Gamma \vdash a =_\beta b \qquad \Gamma \vdash b =_\beta c}{\Gamma \vdash a =_\beta c}\ (3)$$

$$\frac{\Gamma \vdash a =_\beta a'}{\Gamma \vdash x \Rightarrow a =_\beta x \Rightarrow a'}\ (4) \qquad \frac{\Gamma \vdash f =_\beta f'}{\Gamma \vdash f\ a =_\beta f'\ a}\ (5) \qquad \frac{\Gamma \vdash a =_\beta a'}{\Gamma \vdash f\ a =_\beta f\ a'}\ (6)$$

$$\frac{\Gamma \vdash a =_\beta a'}{\Gamma \vdash match\ a\ \{\overline{:l = b;}\} =_\beta match\ a'\ \{\overline{:l = b;}\}}\ (7)$$

$$\frac{\Gamma \vdash \overline{b =_\beta b'}}{\Gamma \vdash match\ a\ \{\overline{:l = b;}\} =_\beta match\ a\ \{\overline{:l = b';}\}}\ (8)$$

$$\frac{\Gamma \vdash a =_\beta a'}{\Gamma \vdash let\ (x,\ y) = a;\ b =_\beta let\ (x,\ y) = a';\ b}\ (9)$$

$$\frac{\Gamma \vdash b =_\beta b'}{\Gamma \vdash let\ (x,\ y) = a;\ b =_\beta let\ (x,\ y) = a;\ b'}\ (10)$$

$$\frac{a =_\alpha b}{\Gamma \vdash a =_\beta b}\ (11) \qquad \frac{x = a\ \in\ \Gamma}{\Gamma \vdash x =_\beta a}\ (12)$$

$$\frac{}{\Gamma \vdash (x \Rightarrow b)\ a =_\beta b\ [x := a]}\ (13) \qquad \frac{}{\Gamma \vdash match\ :l_i\ \{\overline{:l = b;}\} =_\beta b_i}\ (14)$$

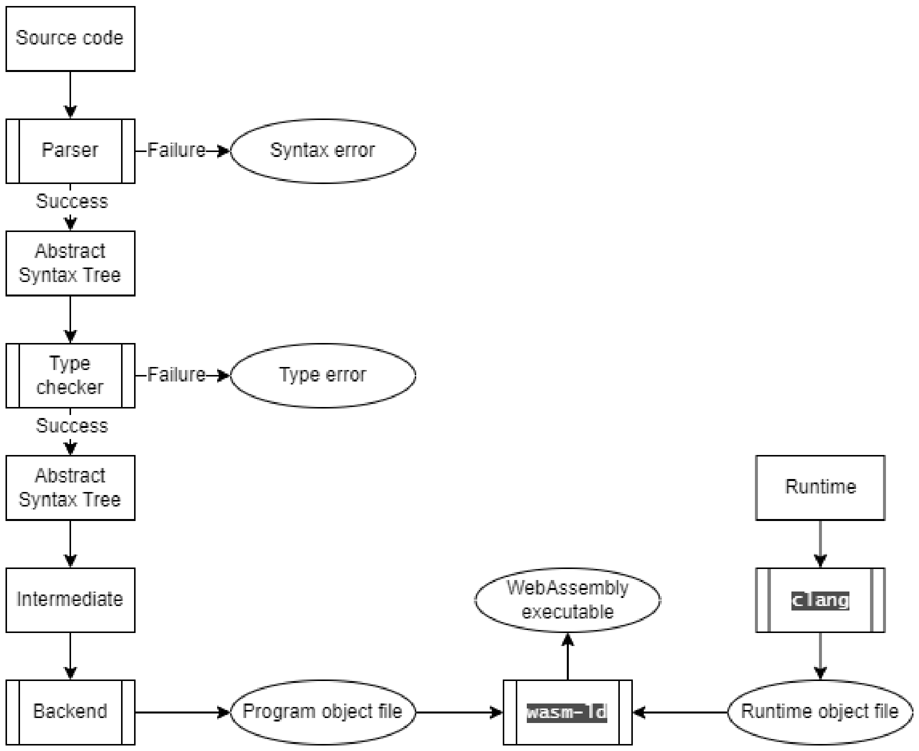
$$\frac{}{\Gamma \vdash let\ (x,\ y) = (a_1,\ a_2);\ b =_\beta b\ [x := a_1,\ y := a_2]}\ (15)$$

pair $(a_1,\ a_2)$ and outputting a term b is β-equal to the term b with its free variables x and y substituted for the terms a_1 and a_2 respectively.

5 THE CURIOS COMPILER

In this chapter, the compilation pipeline of Curios is introduced through detailed explanations on the two representations that a Curios program assumes and an explanation of each of the four components of the compilation pipeline. An overview of Curios' compilation pipeline is available in Figure 5.1.

Figure 5.1 – Overview of Curios' compilation pipeline.



As a Curios program progresses through the compilation pipeline, its representation varies as different compiler pipeline components are employed. Curios programs begin their lifetimes as **source code**, where they are nothing more than text. At this point, the **parser** component is employed, and the Curios program goes from source code to **abstract syntax tree**. As an abstract syntax tree, the Curios program has its types checked by the **type checker** component. When the Curios program is guaranteed to be type correct, the Curios program goes from an abstract syntax tree to an **intermediate representation**, which serves as a bridge between the abstract syntax tree and the WebAssembly binary instructions. In its intermediate representation, important low-level characteristics such as control flow branches, allocations and evaluation order are explicit. To conclude, the intermediate representation is compiled to a WebAssembly object file through the **backend** component and linked to the **runtime**.

The Curios compiler toolchain depends on a number of programming languages and tools to emit an executable WebAssembly module:

- Three of the four of Curios compiler components, the parser, the type checker and the backend, are implemented in the Haskell programming language whose code and dependencies are managed by the Stack dependency manager;
- The fourth and last component of Curios' compiler pipeline, the runtime, is implemented in the C programming language and is compiled to an object file through LLVM's C compiler, `clang`;
- Linking the object file resulting from Curios' backend with the object file resulting from compiling the runtime is performed by `wasm-ld`, LLVM's WebAssembly linker.

Executing the resulting WebAssembly module involves a number of small but nonetheless important components:

- A standalone server, implemented in the Haskell programming language, responsible for handling the HTTP protocol;
- A HTML document, served through a HTTP server, responsible for bootstrapping the execution;
- A JavaScript program, contained within an inline script in the HTML document's header, responsible for downloading, instantiating and invoking the WebAssembly module's start function.

Throughout this chapter, a passing familiarity with the Haskell programming language is assumed from the reader, but the concepts introduced should be not be difficult to understand with a basic background on functional programming languages. The reader might wish to read this chapter alongside Curios' compiler toolchain source code (PRETTO, 2023), in which case a stronger familiarity with the Haskell programming language is expected.

5.1 Parser

The parser takes input in the form source code, checks whether the source code conforms to the rules of Curios' grammar and outputs an abstract syntax tree. It was developed using Haskell's `megaparsec` library (KARPOV, 2023), which is a parsing library based on the concept of combinatory parsing. At the heart of combinatory parsing are parser combinators, which are composable higher order functions that perform the parsing task when applied to a textual input. Parser combinators offer an immediate advantage over tools like `lex` and `yacc` for C, or `alex` for Haskell, because instead of writing a parser through a separate specification, the specification of the parser is the code that performs the parsing itself.

The implementation of `megaparsec` is based on monads and monad transformers from the `mtl` library, which gives them another big advantage: parser combinators compose with each other through monad transformer stacks, which consequently means that augmenting the parser with complex extensions is done by simply adding another monad transformer or monad to the parser's monad transformer stack.

5.2 Abstract syntax tree

The abstract syntax tree is the first of two representations that a Curios program assumes during its progress through the compiler pipeline, and it is the output of the parser component and the input of the type checker component. The essence of abstract syntax tree is the `Term` data structure. It represents only the information that is essential both for type checking and for generating the intermediate representation. The nodes of the tree correspond to the different elements of Curios: the types, constructors and eliminators of functions, ordered pairs, finite enumerations and WebAssembly primitives.

The noteworthy characteristic of the `Term` data structure is how it chooses to deal with variables: the `Scope` data structure indicates subtrees of `Term` where an additional variable is bound. The variables themselves use a mix of names for free variables and De Bruijn indices for bound variables, which avoids the pitfalls of using either of them exclusively while still reaping the benefits of both of them (MCBRIDE; MCKINNA, 2004). An expanded view on the details of the implementation of variables in the abstract syntax tree is given in Appendix B.

5.3 Type checker

The type checker is the second of four components of the Curios compiler pipeline. It analyses the abstract syntax tree and verifies that the types of variables and terms are used correctly throughout the program. The type checker will flag errors such as using a variable of one type where a different type is expected or calling a function with arguments of the wrong type. After the analysis is complete, the abstract syntax tree is translated into the intermediate representation.

Checking the types of terms is performed with the help of a monad called `Check` that is responsible for general bookkeeping: generating fresh names while also managing the global and local contexts. The generation of fresh names is necessary due to Curios' implementation of variables, which is explained in greater detail in Appendix B. During the type checking process, the global context is only ever read from, while the local context is mutated according to the term that is being checked. For example, if the term being checked is a function, it becomes necessary to first modify the local context with a declaration standing for the variable that the function binds before the body of the function can be checked.

A reduction algorithm is employed by the type checker when two terms need to be compared for β-equality, or when it becomes necessary to match against a specific constructor of `Term`. This reduction only goes as far as the weak head-normal form of the term. Due to the fact that Curios allows general recursion, if the reduction algorithm is invoked against a term that does not have a weak head-normal form, it will loop infinitely. The following is an example of such a program: even though `0` does not inhabit `Bad`, the reduction algorithm will diverge trying to reduce `Bad` to its (non-existent) weak head-normal form as part of type checking.

```
Bad : Type;
Bad = Bad;

example : Bad;
example = 0;
```

Every programming language including dependent types in its type system needs to be mindful of programs existing at the type level, which instigates the necessity of a β-equality algorithm. For strongly normalizing languages, determining whether two terms are β-equal is decidable, but since Curios supports general recursion, β-equality is undecidable. To avoid most cases of divergence, Curios' β-equality algorithm keeps track of pairs of terms standing for equations it has already tried to compare. If the algorithm winds up trying to compare a pair of terms inside an equation that it has already seen, then it assumes that that pair of terms is equirecursive, and returns that the two terms in the equation are β-equal up to equirecursion. It is worth reiterating that there are still cases where the algorithm will diverge (such as in the previous example), since general recursion makes ascertaining β-equality undecidable and only an approximation is possible.

The implementation of the type checker follows a bidirectional discipline (PIERCE; TURNER, 2000), but the rules given in Figure 4.4 follow a type-assignment discipline and are not meant for the direct implementation of a type checker for a programming language. The authors intend on publishing the bidirectional type rules used in the implementation of the Curios type checker at a later date.

After the program has had its types checked and it does not contain type errors, it resumes its progress through the compilation pipeline. At this point, the program is translated into an intermediate representation.

5.4 Intermediate representation

A program can be thought of as a sequence of instructions that operate on data. These instructions are typically written in a high-level language that is designed to be easy for humans to read and write. However, machines cannot directly execute high-level instructions, so they must first be translated into a low-level language that the machine can understand. The intermediate representation is the final of two representations that the Curios program assumes and it is essential in the process of converting the high-level program into the low-level program: whereas the abstract syntax tree focuses on the logical aspects of the program, the intermediate representation focuses on the computational aspects of the program.

After a program is verified to be type correct, the Curios program in its core representation is translated into its intermediate representation. The translation procedure has three major responsibilities:

1. Throw away unnecessary type annotations;
2. Convert functions into data structures representing closures (ADAMS et al., 1986);
3. Flatten terms into sequences of expressions.

Even though most type annotations are discarded, the small amount of type information that remains in the intermediate representation serves a practical purpose for the

Figure 5.2 – Intermediate representation blocks and closures.

$$x,\ y \in Name \quad bx \in BlockName \quad cx \in ClosureName \quad s \in Sequence$$

$$\begin{array}{lll}
Block\ b & ::= & block\ bx\ [x_0,\ x_1\ ,\ ...,\ x_n]\ do \\
 & & \quad s \\
 & & end \\
 & & \\
Closure\ c & ::= & closure\ cx\ \{x_0,\ x_1,\ ...,\ x_n\}\ [y_0,\ y_1,\ ...,\ y_n]\ do \\
 & & \quad s \\
 & & end \\
 & & \\
Program\ p & ::= & \varnothing \\
 & | & p\ b \\
 & | & p\ c
\end{array}$$

Figure 5.3 – Intermediate representation sequences of expressions.

$$x \in Name \quad e \in Expression$$

$$\begin{array}{lll}
Sequence\ s & ::= & x \leftarrow e;\ s \\
 & | & e
\end{array}$$

compilation process: in broad strokes, it describes the layout of Curios objects in memory. Apart from such type information, the intermediate representation also exposes many important low level details such as lifetimes, memory allocation and control flow branches, which are important for the process of emitting WebAssembly instructions.

A program in its intermediate representation is a collection of blocks and closures, as defined in Figure 5.2. Blocks are parameterized by a list of arguments while closures are parameterized by both an environment and by a list of arguments. Both blocks and closures contain, as their bodies, a sequence of expressions to be executed from top to bottom. The environment of closures is represented by a list of variables that the closure has closed over. While blocks can be jumped to directly, closures must first be allocated and subsequently entered.

Sequences of expressions, defined in Figure 5.3, exist in the body of a block and can be thought of as an expression and an optional continuation, which is itself a sequence of expressions. The result of executing the expression is bound to a name and if a continuation exists, the expression is a **named expression**: the name is substituted in the

Figure 5.4 – Intermediate representation expressions.

$$x \in Name \quad bx \in BlockName \quad cx \in ClosureName$$
$$i \in Index \quad n \in Integer \quad m \in Floating$$

$$
\begin{array}{lll}
Atom\ a & ::= & x \\
 & | & null \\
 & & \\
Expression\ e & ::= & Pure\ a \\
 & | & Jump\ bx\ [a_0,\ a_1,\ ...,\ a_n] \\
 & | & closure.Alloc\ cx\ \{a_0,\ a_1,\ ...,\ a_n\} \\
 & | & closure.Enter\ a\ [a_0,\ a_1,\ ...,\ a_n] \\
 & | & struct.Alloc\ [a_0,\ a_1,\ ...,\ a_n] \\
 & | & struct.Select\ a\ i \\
 & | & int32.Alloc\ n \\
 & | & int32.Add\ a_1\ a_2 \\
 & | & int32.... \\
 & | & flt32.Alloc\ m \\
 & | & flt32.Add\ a_1\ a_2 \\
 & | & flt32....
\end{array}
$$

body of the continuation and the compilation process recurses onto the continuation. If a continuation does not exist, the expression is a **tail expression**, denoting the end of the sequence (JONES; BAILEY; COOPER, 2018).

Expressions, specified in Figure 5.4[1], represent a computation to be performed at runtime and they are used to describe the allocation and consumption of data. One noteworthy detail is that expressions do not mention other expressions and, instead, mention **atoms**: the operands to the operations that expressions perform must be trivial i.e. they must either be a name bound by a previous step of the sequence or they must be null.

Curios' intermediate representation has a text representation that is used mainly for debugging purposes, but it can also be used to demonstrate what Curios programs in their intermediate representation look like. For example, a program that allocates a closure for adding two `Int32` values looks like the following:

```
closure add {one} [other] do
  int32.Add [one, other]
end

```

[1]The ellipsis represents the remaining numeric operators from Section 3.2 that were omitted due to space reasons.

```
 5 block start [] do
 6   one <- int32.Alloc 2;
 7   other <- int32.Alloc 3;
 8   add_closure <- closure.Alloc add {one};
 9   closure.Enter [add_closure] [other]
10 end
```

In this example, lines 1 through 3 denote the closure that performs the addition operation. In line 1, the closure is defined to have the name `add`, to contain one closed-over variable (`one`) and to require one argument to be supplied when the closure is entered (`other`). The closure contains a single expression, in line 2, that performs the addition.

Lines 5 through 10 denote a block that uses the closure defined in lines 1 through 3. In line 5, the block is defined to have the name `start`, and because it is a block with specifically the name `start`, it is the block that will be executed after the WebAssembly module is loaded and instantiated and is considered the entrypoint of the program. Line 6 allocates the 32-bit integer `2` and line 7 allocates the 32-bit integer `3`. Line 8 allocates the `add` closure with the integer allocated in line 6 as part of its environment. Line 9 is a tail expression that performs the addition by entering the closure allocated in line 8 which provides the argument that the closure requires. The block is concluded in line 10.

Before the compilation process is concluded, Curios' WebAssembly backend is applied to the intermediate representation to emit a WebAssembly object file which will contain a number of unresolved symbols. The compilation process is concluded by employing LLVM's WebAssembly linker, `wasm-ld`, which will resolve the unresolved symbols with its own symbols and with the symbols coming from Curios' runtime.

5.5 Backend

Curios' WebAssembly backend is the third of four components and is, by far, the biggest component of all of Curios' compiler pipeline components and it implements, as faithfully as possible, a large portion of the WebAssembly specification (ROSSBERG, 2022). It also implements the LLVM convention for WebAssembly static libraries, which means that binaries emitted by Curios' WebAssembly backend are compatible with LLVM's WebAssembly linker, `wasm-ld`. Its functionalities are spread across three main modules: the `Syntax` module, the `Serialize` module and the `Construct` module.

The `Syntax` module houses all of the types necessary for representing the WebAssembly specification, plus some types used to emit linking information. The `Serialize` module takes care of converting the types defined in the `Syntax` module into their binary format. The `Construct` module is where the meat of the backend component is, and it offers a DSL (domain specific language) that allows the construction of WebAssembly modules without having to worry about indices.

The only way to refer to WebAssembly's functions, tables, memories, globals, locals and branch labels is through indices, and manually managing the indices of a WebAssembly module is prone to errors. This is specially true for branch labels, since much like De Bruijn indices, the most recently bound branch label is 0, and increasing indices refer to branch labels outwards. Because of that, the DSL is implemented through a monad whose API (application programming interface) allows the programmer to use strings instead of indices wherever an index would be required, such as in the target of a branch instruction or when getting a local variable.

To understand how the DSL offered by the `Construct` module works, the examples in section 2.3.2 will be reimplemented using the DSL. Let us begin with the module that declares a function that adds two `i32` values.

```
addExample :: Module
addExample = runConstruct $ do
  declareFunc "add" [("a", i32), ("b", i32)] [i32]
  startCode
  pushLocalGet "a"
  pushLocalGet "b"
  pushI32Add
  endCode
```

Easy enough, right? The DSL mimics WebAssembly's stack discipline on purpose so that WebAssembly modules in their text representation are not so different from the syntax that the DSL offers. To conclude, let us reimplement the module that declares a function that sums all numbers from `0` to `n`.

```
fromExample :: Module
fromExample = runConstruct $ do
  declareFunc "from" [("n", i32)] [i32]
  startCode
  pushLocal "result" i32

  pushI32Const 0
  pushLocalSet "result"

  pushBlock "break"
  pushLoop "loop"

  pushLocalGet "n"
  pushI32Eqz
  pushBrIf "break"

  pushLocalGet "result"
  pushLocalGet "n"
  pushI32Add
  pushLocalSet "result"

  pushLocalGet "n"
  pushI32Const 1
  pushI32Sub
  pushLocalSet "n"

  pushBr "loop"

  popLoop
  popBlock

  pushLocalGet "result"
  endCode
```

5.6 Runtime

Applying Curios' WebAssembly backend to a program in its intermediate representation does not generate a WebAssembly executable just yet. Instead, it generates a WebAssembly object file that still needs to be linked against the runtime, since the object file will contain unresolved symbols.

The runtime is the final of four components of Curios' compilation pipeline. It is written in the C programming language and it provides a set of operations that allows the program to manage its memory and collect its garbage. Curios objects follow an uniform representation, are reference counted and contain two segments: one segment is a list of nested objects known as *children* which contain objects whose lifetime is tied to the parent object, and the other segment is an unmanaged chunk of bytes where non garbage collected data can be stored known as the *trunk*. Curios' objects are laid out in memory as follows:

- **4 bytes** for the reference count;
- **4 bytes** for the capacity of the children list;
- **4 bytes * Capacity** for the children list;
- **N bytes** for the trunk of the object, determined by what's being stored.

Curios' objects, their lifetimes and the lifetimes of nested objects are determined by calls to the runtime which will manage the allocation and deallocation of objects (REINKING et al., 2021). The runtime provides the following operations for managing the lifetime of an object:

- `new(capacity, trunk_size)`, which allocates a new object with capacity for `capacity` children and a trunk of size `trunk_size`;
- `enter(object)`, which increments the reference count of `object` if it is not 0 i.e. it is not static;
- `leave(object)`, which decrements the reference count of `object` if it is not 0 i.e. it is not static. If the reference count of `object` becomes 0 after being decremented, all of `object`'s children leave its scope and `object` gets deallocated;
- `set(object, index, child)`, which sets the child at `index` of `object` to `child`;

- `get(object, index)`, which returns the child at `index` of `object`;
- `trunk(object)`, which calculates a pointer to the trunk of `object`.

To exemplify how different types of data can be stored using this object representation, we will look at how three different data types are allocated: 32 bit integers, structures and closures.

- 32 bit integers hold no nested objects, and the actual value is stored in the trunk since the value does not need to be garbage collected. Thus, 32 bit integers stored in memory have a capacity of `0` children and a trunk size of `4` bytes;
- Structures' capacities are based on the number of fields that they store, and they do not store any data that is not garbage collected. Thus, structures stored in memory have a capacity of `n` children and a trunk size of `0` bytes, where `n` is the amount of fields that the structure has;
- Closures store their closed over variables as children, and the function pointer to be invoked in the trunk. Thus, closures stored in memory have a capacity of `n` children, and a trunk of `4` bytes, where `n` is the amount of closed over variables.

5.7 Execution

Web browsers employ security policies that restrict how external resources can be requested and downloaded. Because of that, the WebAssembly executable that results from Curios' compilation pipeline needs to be served through a HTTP server. The Curios compiler toolchain includes a barebones HTTP server for the purpose of conforming to the web browser's security policies, so that it agrees to download the resources necessary to execute the Curios WebAssembly executable.

The first resource that the web browser downloads from the HTTP server is a HTML document containing the customarily expected HTML document skeleton: a set of `<html></html>` tags to delimit where the HTML document begins and ends, a set of `<head></head>` tags to delimit the header of the HTML document and a set of `<body></body>` tags to delimit the contents of HTML document (which, at first, is empty). The header of the HTML document contains a set of `<script></script>` tags that contain an inline JavaScript program that invokes the Fetch API (MDN WEB DOCS CONTRIBUTORS, 2023) to make a request to the HTTP server asking it to provide the WebAssembly module, which gets asynchronously downloaded and instantiated.

Once the WebAssembly module is downloaded and instantiated, it is ready to be executed. The JavaScript program expects a `_start` symbol to be exported, and expects this symbol to be a function requiring zero arguments and returning one value. The JavaScript program retrieves the `_start` symbol, invokes it, implicitly converts the returned value to a string and appends that string to the contents of the HTML document. Lastly, the web browser takes care of rendering the updated contents of the HTML document, which will effectively print the result of invoking the `_start` symbol to the screen.

6 RELATED WORK

In this chapter, works related to Curios are introduced, along with the extent of their influence in the research and development of its compiler pipeline. In Section 6.1 the Cedille proof assistant and dependently typed programming language is introduced. In Section 6.2 the Formality programming language and its offspring are introduced. In Section 6.3 the ΠΣ type system is introduced.

6.1 Cedille and the Calculus of Dependent Lambda Eliminations

Proof assistants such as Coq and Lean and programming languages such as Idris adopt inductive types following the Calculus of Inductive Constructions approach that extends the Calculus of Constructions with new primitives for constructing and eliminating inductive types (and consequently, adds new typing and evaluation rules for these new primitives).

The Calculus of Dependent Lambda Eliminations (STUMP, 2017) proposes an alternative way to introduce inductive types to the Calculus of Constructions. Instead of adding new primitives, the Calculus of Dependent Lambda Eliminations introduces inductive types to the type system through lambda encodings (empowered by Miquel's implicit function type (BARRAS; BERNARDO, 2008), Kopylov's dependent intersection type (KOPYLOV, 2003) and an equality type representing the standard Leibniz equality). The Calculus of Dependent Lambda Eliminations is the theory behind the typechecker of the Cedille[1] proof assistant and dependently typed programming language (STUMP, 2023).

To briefly recapitulate, lambda encodings are a technique for encoding data using nothing but function abstraction and application. Church numerals are the most well-known form of lambda encoding, where the natural numbers are encoded through higher-order functions that represent the iterator of the encoded number. Two functions represent each of the encoding's constructors: *zero* is represented by $\lambda z.\ \lambda s.\ z$ while *succ* (successor) is represented by $\lambda n.\ \lambda z.\ \lambda s.\ s\ n$. For example, the number 1 is represented by *succ zero*, the number 2 is represented by *succ* (*succ zero*) and so on.

[1]Cedille is a word formed by adding three vowels and one consonant to the acronym *CDLE* standing for **C**alculus of **D**ependent **L**ambda **E**liminations. It is the name of the **ç** character found in the alphabet of languages such as portuguese and french.

Eliminating the encoding is equivalent to applying two terms to the encoding, one for the case where the encoding represents *zero* and another for the case where the encoding represents *succ*, at which point the encoding chooses one of the terms with which to continue based on which constructor the encoding represents. For example, a function that tests whether its argument is zero can be defined as $\lambda n.\ n\, true\ (\lambda x.\ false)$.

Curios' inductive types are very different from Cedille's inductive types in the sense that Curios offers basic data types with which other data types can be derived whereas Cedille provides lambda encodings with enough gunpower so that they are capable of representing data types. But even if Curios and Cedille have their differences, Curios shares Cedille's (and the Calculus of Dependent Lambda Eliminations) philosophy of offering inductive types through constructs that are more fundamental and simpler to understand when compared to the approach that the Calculus of Inductive Constructions takes where types must obey formation schemes and behave as extensions to the type system. But, unfortunately, even though the type rules for Cedille's type system are comparatively simpler, the difference in complexity is transferred from the type checker to the user: the syntax for types and terms using the raw, low-level encoding is often very verbose (STUMP, 2018).

Cedille promptly solves this problem by adding a high-level syntax for inductive types that desugars into the encoding, but the encoding seldom resembles the high-level syntax, and this disconnect between the high-level syntax and the encoding can be detrimental to the implementation of a programming language: debugging the different components of a compiler pipeline might require inspecting the not-so-readable encoding manually. Another drawback of the encoding, this time regarding the generation of machine instructions, is that whenever a memory layout needs to be chosen for an encoded object, the compiler needs to either perform an extra analysis step that decodes the encoding into a representation that describes the memory layout of the object as a structure or it must resort to compiling the encoded object as a closure. The drawback of resorting to compiling encoded objects as closures has to do with the fact that unoptimized closures have associated runtime performance overheads, such as heap allocations and indirect jumps.

6.2 Formality, Kind and Yatima

Formality (MAIA, 2021a) was a dependently typed functional programming language that had the objective of showcasing the benefits that proof-carrying code could bring to day-to-day programming. Formality was heavily inspired by a previous revision of the Calculus of Dependent Lambda Eliminations called System S (FU; STUMP, 2014) and employed the concept of *self type* as a tool to represent inductive types through lambda encodings. After the authors of the language went their separate ways, Formality was split into two projects, Kind (MAIA, 2021b) and Yatima (BURNHAM, 2021), which continued to use self types in their type systems. For the remainder of this section, the term Formality will be used to refer to any of the three programming languages (Formality, Kind or Yatima).

Like previously mentioned, Formality adopted the self type, which originated from the research that led to Cedille, as the tool used to introduced new inductive types to its type system. The main differences between Formality and Cedille have to do with recursion and termination. While Cedille restricts recursion to its well-founded subset with the intent of guaranteeing termination, Formality embraces general recursion and does not guarantee termination. It is also worth mentioning that Cedille treats recursion in its types through an isorecursive approach (i.e. the type is isomorphic but not equal to its recursive unfolding) while Formality treats recursion in its types through an equirecursive approach (i.e. the type and and its recursive unfolding are equal). Each approach has its strengths and drawbacks: equirecursive types are more straightforward but they can present implementation challenges while isorecursive types are easier to implement but have additional syntax and typing rules.

The first prototype of Curios was, in essence, very similar to Formality: while Curios and Formality had different syntaxes and offered different sets of primitive capabilities, Curios also employed the self type as the tool through which new inductive types were introduced to its type system. Curios was also inspired by Formality's implementation of its β-equality check algorithm, since it was capable of handling general recursion at both term and type levels without modification. Curios eventually abandoned the self type for two reasons:

1. The self type as proposed by Formality had not yet been given a formal specification, and while there were attempts by the Curios authors to derive a formal specification from the implementation of Curios' type checker, the attempts were

not successful;

2. The overarching concept of self type exhibited potential through the perspective of an expressive and flexible type operator, but after a full year and a half struggling to understand its intricacies, it was eventually considered by the Curios authors to be too general without justification.

6.3 ΠΣ: Dependent types without the sugar

In the same vein of alternatives to the Calculus of Inductive Constructions, ΠΣ (ALTENKIRCH et al., 2010) is a type system for a dependently typed programming language that encodes inductive types through a combination of dependent functions (i.e. Π-types), dependent products (i.e. Σ-types), enumerations and lifting.

Unlike Cedille and Formality that propose comparatively novel and exotic constructs for the task at hand, ΠΣ breaks the components of an inductive type into already well-studied constituents. By focusing on a core language instead of a full featured language, ΠΣ demonstrates how complex constructs can be represented through combinations of more elementary capabilities. Like Formality, ΠΣ is partial: the tools for carrying out mathematical proofs are provided but its focus is to support general purpose functional programming, which leads to general recursion being an integral part of its system.

After abandoning self types, Curios' dependent function types, dependent ordered pair types and finite enumerations were directly inspired by ΠΣ's dependent functions, dependent products and enumerations. The main difference between Curios and ΠΣ lies in how recursion is handled:

- In ΠΣ, there exist operators used specifically for controlling how recursion is unfolded called *lifting* and *boxing*: boxed terms are typed by lifted types[2], and inside boxed terms, only α-equality (as opposed to full β-equality) is employed. When compared to Curios, these operators add a slight syntactical overhead but ΠΣ's type system rules, operational semantics and type checker implementation are simplified;
- In Curios, no syntax is required to control recursion other than the names of top-level declarations, but the implementation of its β-equality algorithm requires an ad-hoc heuristic: the β-equality algorithm keeps track of equations of terms that it

[2]As explained in the ΠΣ paper, "lifted" here is used as a synonym of "suspended", akin to "lazy".

has already tried to compare so that when it encounters an equation it has already seen, the terms in the equation are considered to be β-equal up to equirecursion. When compared to ΠΣ, Curios' recursion syntax overhead is almost nonexistent but the metatheoretical study of its type system becomes more complicated while also requiring its β-equality algorithm to employ ad-hoc heuristics (such as the approach that was just described) so that in most cases it does not diverge[3].

The hawk-eyed reader might have noticed that Curios' method (and consequently, ΠΣ's method) of encoding inductive types in Section 3.4 is eerily similar to C-style tagged unions where a structure composed of two fields (a tag and a union) is capable of storing values of different types. Similarities can also be drawn to Elixir algebraic data types, where a tuple stores its variant as an atom in its first entry, and the entries in the rest of the tuple vary according to the atom in the first entry. These similarities, along with the simplicity of ΠΣ's concepts, were an encouragement for the adoption of ΠΣ's ideas by Curios due to their focus on representing ideas fundamentally based on idioms coming from programming languages instead of proof assistants.

[3]Like mentioned in Section 5.3, there are still cases where Curios' β-equality algorithm diverges, such as when one of the terms of the comparison does not have a weak head-normal form.

7 CONCLUSION AND FUTURE WORK

In the present research project we have described Curios, a dependently typed functional programming language whose primary compilation target is WebAssembly. In its type system, Curios employs a combination of dependent function types, dependent ordered pair types and finite enumerations in order to represent inductive data types instead of using a more fully fledged but also more complex theory such as the Calculus of Inductive Constructions. An integral part of Curios is general recursion: while general recursion is at the heart of many programming languages, general recursion leads to partiality, and because of that, Curios' type system is inconsistent when viewed as a logic. To illustrate Curios' syntax and semantics, examples describing its basic and dependently typed capabilities and how these capabilities can potentially interact for the sake of representing more complex constructs were introduced. Curios syntax, type rules and operational semantics were described next and, subsequently, an explanation was given on each of the two representations that a Curios program assumes during its progress through the four components of its compiler pipeline. To conclude, related work in the field of type theory was presented along with a discussion on how Curios was influenced by them.

The following proposed objectives were achieved: a dependently typed functional programming language encompassing both a **type checking algorithm** for a type system based on the Calculus of Constructions; a **code generation algorithm** that outputs executable WebAssembly modules was implemented. The code demonstrating these capabilities is available in a public Github repository (PRETTO, 2023) as part of Curios' compiler toolchain.

There were some objectives that were initially set but were left as topics for future work: a convenient **high level syntax** for the concepts that its type system can represent; demonstrating whether there exists dependently typed idioms that can be of service for the **development of web apps**; the development of its **metatheory** such as proving important properties including progress of well-typed terms and type preservation.

As for the future of the Curios compiler toolchain, its development is still in a pre-alpha state and a lot remains to be done before it can be considered to meet even basic quality standards. Besides the investments necessary in Curios' compiler toolchain, there is also interest in the development of a termination checker which would allow a subset of Curios to be used as a proof assistant. Some areas where the frontend of the Curios compiler toolchain could improve are:

- **Syntax.** Before a programming language can be considered production-grade, its syntax must support a set of basic features where the programmer is capable of expressing their ideas as conveniently as possible. Curios offers dependent function types, dependent ordered pair types, finite enumerations and expects the programmer to use these constructs to encode data types, but while these constructs are capable of encoding inductive types, the syntax is verbose and inconvenient;
- **Type inference.** The ability to deduce the type of a variable of a term without requiring explicit type annotations from the programmer is crucial for a programming language: it helps the programmer to write code more quickly and with fewer errors since it allows the programmer to omit part of the type information. Currently, Curios has a very primitive mechanism for inferring the types of terms and can basically only infer a type for a term if it is obvious;
- **Ad-hoc polymorphism.** Allowing a single function to have multiple implementations based on the type of the input arguments a la Haskell's type classes and Rust's traits is advantageous because it allows programmers to write generic functions or operators that can be applied to a variety of types without having to write separate implementations for each type, effectively reducing code duplication and making code more modular and reusable;
- **Effect system.** An effect system opens the door for idioms such as modeling stateful computations in pure languages and it becomes a necessity if a programming language intends on providing input and output facilities: whereas a proof assistant is useful just by virtue of its type checker, a programming language that does not provide the tools necessary to interact with the external world is not a very useful programming language. Two options of effect systems for programming languages are monads (WADLER, 1995) and algebraic effects (LEIJEN, 2016);
- **Linear types.** Curios already supports dependent types and that allows its type system to track the properties of its resources, while the addition of linear types (WADLER, 1990) would allow its type system to track the usage of its resources. For example, linear types can guarantee during compile time that only a single reference to some object exists, and that enables optimizations such as destructive updates since data races can only occur when at least two observers exist.

With regards to its compiler backend, Curios lacks any form of optimization. Functional programs are susceptible to many well-known optimizations and there are many optimizations that could be implemented at Curios' intermediate representation level but, unfortunately, Curios' compiler does not yet implement and perform any of them. A set of basic optimizations that need to be implemented and performed have been identified before the Curios compiler toolchain can output executables that can be considered to perform at the most basic level of expectations:

- **Multi-argument functions.** Curios functions each bind a single argument and are eliminated one argument at a time. Representing functions in such a manner is awfully inefficient with regards to execution in bare-metal environments such as machine code, or in a low level virtual machine such as WebAssembly. An optimization pass can eliminate the overhead of allocating closures by gathering all of the arguments in an application and applying them at the same time to the function being eliminated;
- **Type erasure.** Curios is a dependently typed language and that means the boundary between types and terms is blurry. That also means that a separate analysis step is necessary for differentiating terms from types, which currently Curios does not perform: all types are emitted to executables as instances of the null pointer, which creates a runtime overhead that, even if small, is still unnecessary, since their only influence in the result of the program is the overhead itself. An optimization pass should prevent null pointers representing types from being emitted to the executable;
- **Tail-call recursion.** Pure, functional languages such as Curios do not provide mutable data structures, and recursion is employed when iteration and repetition are required. Unfortunately, unoptimized recursion has the heavy drawback of allocating a stack frame each time it recommences, which can easily lead to a stack overflow, even for small and trivial programs. An optimization pass should identify opportunities for tail call optimizations (CLINGER, 1998) where the stack frame for each iteration is unwound before the next iteration begins, leading to the program using a constant (instead of an increasing) amount of stack memory when evaluating recursion;
- **Memory, through the lens of dynamic typing.** Even though Curios is a statically typed language, the flexibility that dependent types permit can persuade a programmer to look at its type system from a dynamic perspective. Nonetheless, programs written in Curios currently use, even for the standards of dynamically typed languages, a large and inefficient amount of memory unnecessarily: every

single object is allocated in the heap, and consume at the very least 12 bytes. There are optimizations such as pointer tagging (GUDEMAN, 1995) and NaN-boxing (FATEMAN, 1982) that can identify small objects[1] and represent them without a heap allocations;

- **Memory, through the lens of static typing.** In the same train of thought, Curios' intermediate representation could be expanded to include a modest, first-order type system that describes the memory layout of its objects more accurately. Apart from the size and layout gains, this could also be the entrypoint for optimizations such as allocating objects in the stack instead of in the heap, where the allocation overhead is much smaller.

[1]Simple scalars (such integers and floating point numbers) and labels (which are represented as integers) are examples of small objects that would benefit greatly from such optimization. These objects comprise the vast majority of objects that a Curios program manipulates as it executes.

REFERENCES

ADAMS, N. et al. Orbit: An optimizing compiler for Scheme. **ACM SIGPLAN Notices**, ACM New York, NY, USA, v. 21, n. 7, p. 219–233, 1986.

AGDA CONTRIBUTORS. **Compilers – Agda 2.6.2.2.20221128 documentation**. 2022. <https://agda.readthedocs.io/en/v2.6.2.2.20221128/tools/compilers.html>.

ALTENKIRCH, T. et al. ΠΣ: Dependent types without the sugar. In: SPRINGER. **International Symposium on Functional and Logic Programming**. [S.l.], 2010. p. 40–55.

BARRAS, B.; BERNARDO, B. The implicit calculus of constructions as a programming language with dependent types. In: SPRINGER. **Foundations of Software Science and Computational Structures: 11th International Conference, FOSSACS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 11**. [S.l.], 2008. p. 365–379.

BARTHE, G.; GRÉGOIRE, B.; BÉGUELIN, S. Z. Formal certification of code-based cryptographic proofs. In: **Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages**. [S.l.: s.n.], 2009. p. 90–101.

BIERMAN, G.; ABADI, M.; TORGERSEN, M. Understanding TypeScript. In: SPRINGER. **ECOOP 2014–Object-Oriented Programming: 28th European Conference, Uppsala, Sweden, July 28–August 1, 2014. Proceedings 28**. [S.l.], 2014. p. 257–281.

BRADY, E. Idris, a general-purpose dependently typed programming language: Design and implementation. **Journal of functional programming**, Cambridge University Press, v. 23, n. 5, p. 552–593, 2013.

BRADY, E. State machines all the way down: an architecture for dependently typed applications. **Unpublished Draft**, 2016.

BRUIJN, N. G. D. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. In: ELSEVIER. **Indagationes Mathematicae (Proceedings)**. [S.l.], 1972. v. 75, n. 5, p. 381–392.

BURNHAM, J. C. **Yatima on Github**. [S.l.]: GitHub, 2021. <https://github.com/yatima-inc/yatima-lang-alpha/tree/4a2073554184a6527d10f1ec6d7669e5f7cc5ea9>.

CLINGER, W. D. Proper tail recursion and space efficiency. In: **Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation**. [S.l.: s.n.], 1998. p. 174–185.

COQUAND, T. Pattern matching with dependent types. In: CITESEER. **Informal proceedings of Logical Frameworks**. [S.l.], 1992. v. 92, p. 66–79.

CZAPLICKI, E. Elm: Concurrent FRP for functional GUIs. **Senior thesis, Harvard University**, v. 30, 2012.

DOWEK, G. The undecidability of typability in the lambda-pi-calculus. In: SPRINGER. **International Conference on Typed Lambda Calculi and Applications**. [S.l.], 1993. p. 139–145.

DUBOIS, C. Proving ml type soundness within coq. In: SPRINGER. **Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000 Portland, OR, USA, August 14–18, 2000 Proceedings 13**. [S.l.], 2000. p. 126–144.

FATEMAN, R. J. High-level language implications of the proposed IEEE floating-point standard. **ACM Transactions on Programming Languages and Systems (TOPLAS)**, ACM New York, NY, USA, v. 4, n. 2, p. 239–257, 1982.

FU, P.; STUMP, A. Self types for dependently typed lambda encodings. In: SPRINGER. **Rewriting and Typed Lambda Calculi: Joint International Conference, RTA-TLCA 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings 25**. [S.l.], 2014. p. 224–239.

GOGUEN, H.; MCBRIDE, C.; MCKINNA, J. Eliminating dependent pattern matching. **Algebra, Meaning, and Computation: Essays dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday**, Springer, p. 521–540, 2006.

GONTHIER, G. et al. Formal proof–the four-color theorem. **Notices of the AMS**, v. 55, n. 11, p. 1382–1393, 2008.

GUDEMAN, D. **Representing type information in dynamically typed languages**. [S.l.]: Citeseer, 1995.

HOARE, T. **Null References: The Billion Dollar Mistake**. 2009. <https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/>.

IDRIS CONTRIBUTORS. **Code generation targets – Idris 1.33 documentation**. 2020. <https://docs.idris-lang.org/en/v1.3.4/reference/codegen.html>.

IDRIS2 CONTRIBUTORS. **Idris2 Github repository**. 2022. <https://github.com/idris-lang/Idris2/tree/59aadd650f9f46f3d7ebd3ede50182a0bea3e280>.

JONES, M. P.; BAILEY, J.; COOPER, T. R. MIL, a monadic intermediate language for implementing functional languages. In: **Proceedings of the 30th Symposium on Implementation and Application of Functional Languages**. [S.l.: s.n.], 2018. p. 71–82.

KARPOV, M. **Megaparsec on Github**. 2023. <https://github.com/mrkkrp/megaparsec>.

KFOURY, A. J.; TIURYN, J.; URZYCZYN, P. The undecidability of the semi-unification problem. In: **Proceedings of the twenty-second annual ACM symposium on Theory of computing**. [S.l.: s.n.], 1990. p. 468–476.

KOPYLOV, A. Dependent intersection: A new way of defining records in type theory. In: IEEE. **18th Annual IEEE Symposium of Logic in Computer Science, 2003. Proceedings.** [S.l.], 2003. p. 86–95.

LEIJEN, D. **Algebraic effects for functional programming**. [S.l.], 2016.

LESANI, M.; BELL, C. J.; CHLIPALA, A. Chapar: certified causally consistent distributed key-value stores. **ACM SIGPLAN Notices**, ACM New York, NY, USA, v. 51, n. 1, p. 357–370, 2016.

MAIA, V. **Formality on Github**. [S.l.]: GitHub, 2021. <https://github.com/VictorTaelin/Formality/tree/040b40308720fab8281e41e1c30c8ee4da4d9a3e>.

MAIA, V. **Kind on Github**. [S.l.]: GitHub, 2021. <https://github.com/HigherOrderCO/Kind1/tree/77fee3a21c9d722a225b3023c2aea6d2251a687c>.

MARTIN-LÖF, P.; SAMBIN, G. **Intuitionistic type theory**. [S.l.]: Bibliopolis Naples, 1984.

MCBRIDE, C.; MCKINNA, J. Functional pearl: I am not a number – I am a free variable. In: **Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell**. [S.l.: s.n.], 2004. p. 1–9.

MDN WEB DOCS CONTRIBUTORS. **Fetch API – Web APIs | MDN**. 2023. <https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API>.

MUIJNCK-HUGHES, J. de; BRADY, E.; VANDERBAUWHEDE, W. Value-dependent session design in a dependently typed language. **arXiv preprint arXiv:1904.01288**, 2019.

PAULIN-MOHRING, C. **Introduction to the calculus of inductive constructions**. [S.l.]: College Publications, 2015.

PIERCE, B. C. Higher-order polymorphism. In: **Types and programming languages**. [S.l.]: MIT press, 2002. chp. 30.

PIERCE, B. C.; TURNER, D. N. Local type inference. **ACM Transactions on Programming Languages and Systems (TOPLAS)**, ACM New York, NY, USA, v. 22, n. 1, p. 1–44, 2000.

PRETTO, V. **Curios on Github**. 2023. <https://github.com/valmirjunior0088/curios/tree/7c107652489967a0ad0577b4b684979d79ef0ab4>.

REINKING, A. et al. Perceus: Garbage free reference counting with reuse. In: **Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation**. [S.l.: s.n.], 2021. p. 96–111.

ROSSBERG, A. (Ed.). **WebAssembly Core Specification**. 2022. Available from Internet: <https://www.w3.org/TR/wasm-core-1/>.

ROUX, C. **What exactly is "large elimination"?** 2023. <https://cstheory.stackexchange.com/q/40342>.

ROUX, C. **Where can I find more information about "dependent elimination"?** 2023. <https://cstheory.stackexchange.com/q/52722>.

SØRENSEN, M. H.; URZYCZYN, P. **Lectures on the Curry-Howard isomorphism**. [S.l.]: Elsevier, 2006.

STUMP, A. The calculus of dependent lambda eliminations. **Journal of Functional Programming**, Cambridge University Press, v. 27, 2017.

STUMP, A. From realizability to induction via dependent intersection. **Annals of Pure and Applied Logic**, Elsevier, v. 169, n. 7, p. 637–655, 2018.

STUMP, A. **Cedille on Github**. 2023. <https://cedille.github.io/>.

WADLER, P. Linear types can change the world! In: CITESEER. **Programming concepts and methods**. [S.l.], 1990. v. 3, n. 4, p. 5.

WADLER, P. Monads for functional programming. In: SPRINGER. **Advanced Functional Programming: First International Spring School on Advanced Functional Programming Techniques Båstad, Sweden, May 24–30, 1995 Tutorial Text 1**. [S.l.], 1995. p. 24–52.

APPENDIX A — INSTALLATION AND USAGE OF CURIOS

In this chapter, a short tutorial on what is necessary to experiment with Curios' compiler toolchain is given. The following dependencies are required to be installed:

- `git`, for cloning the repository;
- `stack`, for managing Curios' compiler toolchain source code and its dependencies;
- `clang`, for compiling Curios' runtime libraries;
- `lld`, for linking Curios' runtime with its compiler output.

The first step is to download Curios' compiler toolchain source code. It is available in a public Github repository, and the `git` tool will take care of this task. To download Curios' compiler toolchain source code, run the following terminal command:

```
git clone https://github.com/valmirjunior0088/curios
```

The next step is to build Curios' compiler toolchain. The `stack` tool is responsible for this task, and it will automatically download all Haskell libraries that the Curios compiler toolchain depends upon. After `stack` is finished downloading the necessary dependencies, it will generate three binaries: one for Curios' interpreter, one for Curios' compiler, and one that spawns a HTTP server. Navigate to the folder where the `git` tool downloaded Curios' compiler toolchain source code, and run the following terminal command:

```
stack build
```

After these two steps are done, the Curios compiler toolchain is available for usage. To facilitate interaction with the compiler toolchain, a `Makefile` is available inside the `runtime` folder that runs the correct commands for interpreting, compiling and serving the resulting WebAssembly executable over a local HTTP server. The `Makefile` will use the `program.crs` file available in the same folder as the Curios program to interpret or compile. The following commands are available:

- `make interpret` will interpret the `program.crs` file and print the result of executing the `start` definition;

- `make serve` will compile the `program.crs` file and spawn a HTTP server on `http://localhost:8080` which will serve the resulting WebAssembly executable and print the result of executing the `start` definition.

APPENDIX B — VARIABLES IN THE ABSTRACT SYNTAX TREE

Like previously mentioned, the noteworthy characteristic of the `Term` data structure is how it chooses to deal with variables: the `Scope` data structure indicates subtrees of `Term` where an additional variable is bound. For example, `Term` is assumed to have no bound variables (such as in the branch of a match expression), `Scope Term` is assumed to have one bound variable (such as in a function) and `Scope (Scope Term)` is assumed to have two bound variables (such as in a split expression)[1].

There are two different ways of representing variables within `Term`s: they are either globals or locals. Global variables, also known as top-level variables, refer to declarations in the global context, while local variables refer to variables bound locally by function types, functions, pair types and split expressions. Local variables are slightly more involved: they are distinguished between free or bound and are used together with `Scope` to represent subtrees of `Term`s where an additional local bound variable is available.

For the time being, let us ignore the concrete implementation of `Scope`s and variables and let us focus on how to manipulate `Term`s through a set of primitive procedures. There are two main procedures used to manipulate `Term`s, `Scope`s and their variables:

1. `abstract`, which creates a `Scope a` by recursing over `a` and binding some local free variable as a local bound variable i.e. it substitutes all occurrences of some local free variable by the local bound variable representing the `Scope`;
2. `instantiate`, which returns the `a` that is underneath a `Scope a` by recursing over the `a` and substituting all occurrences of the local bound variable that the `Scope` represents with some `Term`.

When parsing a `Term`, identifiers that do not match with a specific grammatical rule (such as the `Int32`, which results in the type of 32-bit integers) will be parsed as local free variables, which means that `Term`s will initially contain only local free variables. The `abstract` procedure is used to correctly parse scope formers such as functions where it reconstructs a `Term` as a `Scope Term` by substituting all occurrences of some local free variable with an appropriate local bound variable.

[1]These and related data structures along with all of the associated operations are available in the `src/Core/Syntax.hs` source code file of the Curios Github repository (PRETTO, 2023).

Apart from that, `abstract` is also responsible for constructing synthetic[2] terms: when the parser comes across syntax containing syntactic sugar, the parser constructs `Term`s containing manually injected local free variables and employs the `abstract` procedure to construct `Scope Term`s, desugaring the syntactic sugar that it has encountered. Currently, for this purpose, `abstract` is being used to desugar nested split expressions while also avoiding unlawful capture of global variables.

The `instantiate` procedure is the polar opposite of the `abstract` procedure: the `abstract` procedure constructs `Scope`s while the `instantiate` procedure eliminates `Scope`s. It does so by substituting all occurrences of some local bound variable represented by the `Scope` with some `Term`. The `instantiate` procedure has two responsibilities: to eliminate redexes and to inspect `Term`s contained within `Scope`s.

The first case where `instantiate` is employed is to eliminate redexes that may arise, such as applications of functions to arguments. In the case of a redex, there exists a readily available `Term` that can be used to instantiate the `Scope`, but this may not always be the case. For example, we can easily check whether the domain of a function type inhabits `Type`. But the range of a function type is a `Scope Type`, representing the fact that the range of a function type can depend on the term the domain. How do we get underneath the `Scope Type` to inspect the `Type` and check whether it actually inhabits `Type`? This is the second case where `instantiate` is employed: the context is extended with the declaration of a (fresh) local free variable that inhabits the function type's domain, and that local free variable can be used to instantiate the `Scope` in the function type's range, at which point it can be checked through the usual means.

It is noteworthy that three additional helper procedures exist with regards to the manipulation of `Scope`s and their variables:

1. `unbound`, whose responsibility is to construct a `Scope` within which the variable that the `Scope` represents does not occur i.e. the variable that the `Scope` represents is not free within itself;
2. `open`, whose responsibility is to instantiate a `Scope` with the name of a local free variable;
3. `commit`, whose responsibility is to substitute all local free variables of a `Term` by global variables of the same name.

[2]"Synthetic" in this case is used in the sense of a term that was constructed artificially by the parser and does not originate in the source code.

While the `unbound` procedure is very simple in its nature, the `open` procedure requires a little more thought but it functions almost exactly like `instantiate`: `open` instantiates a `Scope a` with a local free variable while `instantiate` instantiates a `Scope a` with a `Term`. The difference between the two is subtle and in theory, `open` could be implemented in terms of `instantiate`, but in practice, `Term`s and their variables contain source position information that can only be preserved by the variable-for-variable substitution that the `open` procedure performs. Last but not least, the `commit` procedure is used at the end of the process of parsing a `Term` to convert the variables that were not bound locally into global variables.

Let us resume our discussion with regards to the implementation of `Scope`s and variables in `Term`. There is a number of different techniques that can be employed which allow nodes of the `Term` tree down the road to mention variables bound by earlier nodes. The simplest and easiest technique to implement is to use names, but names offer ample opportunity for issues such as capturing, where substitution can cause a variable bound by a node to start referring to a node which it was not originally meant to refer. Using names also presents a challenge with regards to α-equality: `x => x` and `y => y` are the same function even though the name of the variable being bound differs since α-equality is not concerned with names but rather with structure. A very popular alternative is to use De Bruijn indices (BRUIJN, 1972), where variables are represented not by names but by natural numbers: the most recently bound variable (or, the innermost variable) is 0, and increasing indices refer to variables outwards.

One big drawback to using De Bruijn indices to represent variables is that expressions end up requiring procedures for strengthening and weakening their subtrees when performing substitution: all local free variables in the expression being inserted would need to be strengthened (i.e. incremented) for each scope that it crosses (for example, when substituting a variable for an expression inside the body of a function), and all local free variables need to be weakened (i.e. decremented) when a scope gets eliminated (for example, when eliminating an application of a function to an argument). On top of that, the strengthening and weakening procedures must be invoked against specific subtrees of expressions, and invoking these procedures against the wrong subtree can lead to a a number of issues including wrongful capturing of variables and variables becoming no longer bound in the context.

There is one more drawback to using De Bruijn indices that is not as egregious, but it is still discouraging nonetheless: expressions become hostile to visual inspection.

If de Brujin indices are used exclusively, expressions mention numbers all the way down instead of human readable names, and human readable names can be extremely valuable when inspecting an expression during a code-run-debug cycle of the implementation of the compiler.

In Curios, variables are represented through a mix of names and De Bruijn indices, where global variables and local free variables are represented using a name, while local bound variables are represented by a De Bruijn index (MCBRIDE; MCKINNA, 2004). This approach offers a number of significant advantages over using De Bruijn indices exclusively: substitution is capture-avoiding, strenghtening and weakening procedures become unnecessary, α-equality is a simple equality comparison (i.e. `a == b`, where `a` and `b` are `Term`s) and local variables can be manipulated using a name instead of a number which also leads to `Term`s having the capacity of being rearranged (i.e. through substitution) without the need for dedicated machinery (that is often complex and error prone) responsible for tracking De Bruijn indices in both the context and the `Term` being checked and/or manipulated. Even though this approach requires the generation of fresh local free variables, this requirement is substantially less complex than the alternative.

This approach of using unique, fresh names to represent local free variables has simplified the implementation of the parser and the type checker by a great measure, and has eliminated a source of subtle, hard to chase bugs. It has also simplified the implementation of the procedure that translates `Term`s into their intermediate representations because names can be chosen and reused as the translation procedure recurses over the `Term`.

APPENDIX C — RESUMO ESTENDIDO

Em anos recentes, linguagens de programação funcionais tem aproveitado de um pico em popularidade, com muitos desenvolvedores de software descobrindo os benefícios que o paradigma pode oferecer. Idiomas funcionais como casamento de padrões e funções de primeira classe começaram a não só ser adotados por linguagens de programação clássicas e bem estabelecidas como também influenciam o design de todas as linguagens de programação que estão por vir.

Programação funcional é conhecida por suas raízes profundas em cálculo lambda e teoria de tipos. Linguagens como Haskell e OCaml podem ter as suas funcionalidades que voltadas ao usuário reduzidas para uma linguagem central concisa baseada no cálculo lambda polimórfico de ordem maior, também conhecido como Sistema Fω (PIERCE, 2002). Para brevemente recapitular, o Sistema Fω (PIERCE, 2002) é um sistema de tipos que permite termos dependerem de terms (funções), termos dependerem de tipos (polimorfismo), tipos dependerem de tipos (construtores de tipo), mas não é permitido que tipos dependam de termos (tipos dependentes).

Sob o isomorfismo de Curry-Howard (SØRENSEN; URZYCZYN, 2006), o problema de verificar a corretude de uma prova pode ser reduzida ao problema de checar se um dado programa habita um certo tipo: uma proposição é considerada equivalente a um tipo, e a prova desta proposição é equivalente a um programa. Tipos dependentes empregam o isomorfismo de Curry-Howard especialmente bem porque, por permitir que termos apareçam ao nível de tipo, tipos dependentes podem atuar como fórmulas da lógica de predicados intuicionista (MARTIN-LÖF; SAMBIN, 1984) e, por consequência, eles se tornaram uma funcionalidade mais comumente associada com assistentes de prova.

Tipos dependentes não vêm sem suas desvantagens: tipos dependentes completos (ou seja, permitir que os termos ocorram sem restrição de tipos) leva à indecidibilidade da inferência de tipos. Resumindo, a inferência de tipos para tipos totalmente dependentes pode ser reduzida a um problema chamado *semi-unificação* onde resolver o conjunto de restrições de tipo coletadas das expressões do programa pode levar à não terminação (KFOURY; TIURYN; URZYCZYN, 1990; DOWEK, 1993). O Cálculo das Construções Indutivas (PAULIN-MOHRING, 2015) (que é o formalismo por trás de Coq[1] e Lean[2], dois assistentes de prova) evita esse problema definindo um conjunto de esquemas para a formação de tipos que restringem suas ocorrências recursivas de forma que a terminação

[1]<https://coq.inria.fr/>

[2]<https://leanprover.github.io/>

seja obtida.

Em linguagens de programação funcional, essa restrição é considerada muito pesada porque exclui uma série de expressões populares envolvendo recursão geral, o que levou tipos dependentes a serem um recurso abandonado aos assistentes de prova. Embora a dependência de tipos em termos não seja um tópico novo quando o assunto são linguagens de programação[3], na maioria dos casos eles eram acompanhados por uma série de limitações.

Apesar do desafio de incorporar tipos totalmente dependentes na programação de uso geral, Idris (BRADY, 2013) demonstra a viabilidade de expressões idiomáticas de tipo dependente como ferramentas para escrever software de forma melhor. Por exemplo, tipos de sessão (MUIJNCK-HUGHES; BRADY; VANDERBAUWHEDE, 2019) permitem que protocolos de comunicação sejam especificados inteiramente no nível do tipo, potencialmente permitindo que aplicativos da web assegurem que sua comunicação com um servidor é verificada estaticamente pelo sistema de tipos. Outro exemplo é o módulo `Control.ST` (BRADY, 2016) que oferece uma mônada para efeitos dependentes, que tem o potencial de fornecer tipos mais precisos para expressões idiomáticas contendo efeitos colaterais, como contêineres de estado centralizados.

O cenário atual em relação à execução de programas escritos em linguagens de tipagem dependente tem uma tendência para a transpilação: Idris tem como alvo C e Javascript (IDRIS CONTRIBUTORS, 2020); Agda visa Haskell e JavaScript (AGDA CONTRIBUTORS, 2022); Idris2, a próxima iteração do Idris, tem como alvo Chez Scheme (IDRIS2 CONTRIBUTORS, 2022). Uma desvantagem significativa da transpilação é que uma linguagem de programação transpilada acaba dependendo indiretamente de toda a cadeia de ferramentas do compilador de uma segunda linguagem antes que seus programas possam ser executados. Com isso em mente, levantamos uma questão: seria desejável compilar uma linguagem com tipos dependentes para um formato executável (ou pelo menos uma representação intermediária de baixo nível)?

WebAssembly (ROSSBERG, 2022), um formato de instrução binária para aplicativos da web que é **rápido**, **seguro** e **portátil**, conquistou o cenário da programação da web. O JavaScript costumava ser a única opção como linguagem de programação e como alvo de transpilação para outras linguagens de programação, mas o WebAssembly agora oferece a oportunidade para linguagens compiladas como C++ e Rust, que são vistas como linguagens de programação de baixo nível, para serem executadas no navegador.

[3]Como evidenciado por Pascal que, em 1970, permitiu que o tipo de um array fosse indexado por seu tamanho.

Tão importante quanto trazer as linguagens existentes para o cenário de programação da Web, o WebAssembly tem o potencial de nutrir novas linguagens de programação voltadas especificamente para seu conjunto de instruções. Com isso em mente, refinamos nossa pergunta original: o WebAssembly pode servir como meio para executar programas de tipagem dependente no navegador da Web?

No presente trabalho, exploramos como alguém pode abordar a tarefa de compilar uma linguagem de programação com tipos dependentes introduzindo a linguagem de programação funcional com tipos dependentes Curios. Os principais pontos de venda da Curios são:

- **De propósito geral.** Curios propõe um método para reconciliar recursão geral com tipos dependentes, duas noções que podem ser vistas como contraditórias. Seu sistema de tipos oferece uma alternativa ao Cálculo de Construções Indutivas que tem o potencial de se prestar bem a linguagens de programação;
- **Compila para WebAssembly.** Os benefícios do WebAssembly como destino de compilação são perceptíveis. O Curios procura validar a aptidão do WebAssembly como um destino de compilação para uma linguagem de programação de tipagem dependente.

Os seguintes objetivos propostos foram alcançados: uma linguagem de programação funcional de tipagem dependente englobando tanto um **algoritmo de verificação de tipos** quanto um sistema de tipos baseado no Cálculo de Construções; um **algoritmo de geração de código** que gera módulos WebAssembly executáveis foi implementado. O código que demonstra esses recursos está disponível em um repositório Github público (PRETTO, 2023) como parte da cadeia de ferramentas do compilador Curios.

Quanto ao futuro da cadeia de ferramentas do compilador Curios, seu desenvolvimento ainda está em um estado pré-alfa e muito ainda precisa ser feito antes que possa ser considerado para atender até mesmo aos padrões básicos de qualidade. Não apenas existem muitas otimizações que podem ser aplicadas ao compilador e ao código gerado, mas também há interesse no desenvolvimento de um verificador de terminação que permitirá que um subconjunto de Curios seja usado como um assistente de prova.