University of São Paulo
Institute of Mathematics and Statistics
Bachelor of Computer Science

An exploration of dependent types for data-centric programming

Eduardo Sandalo Porto

Final Essay

mac 499 — Capstone Project

Supervisor: Prof.ª Dr.ª Ana Cristina Vieira de Melo

São Paulo
2024

Acknowledgments

I thank my friends, family, and advisor for helping me through the process of writing this text and dealing with my many last minute changes. I thank the programming language research community for receiving me with open arms and providing many opportunities I previously thought impossible. I also thank the Lean programming language and theorem prover community for developing and disseminating this great tool.

Resumo

Eduardo Sandalo Porto. **Uma exploração de tipos dependentes em programação centrada em dados**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2024.

Sistemas de tipos definem regras que atribuem tipos a expressões de uma linguagem, e são objeto de estudo da teoria dos tipos. A primeira teoria dos tipos foi concebida por Bertrand Russell no início do século XX para ser um fundamento lógico da matemática, embora não tenha conseguido cumprir sua promessa. Linguagens de programação com tipos dependentes restauram esse objetivo através da correspondência de Curry-Howard, resumida pelo lema "proposições como tipos": tipos correspondem a proposições lógicas e programas às suas provas. Como tal, podem ser usadas não apenas para escrever programas, mas também para verificá-los formalmente.

O objetivo deste projeto é estudar tipos dependentes desde suas fundações e aplicá-los a um cenário real, especificamente, a programação centrada em dados. O estudo de teoria dos tipos começa em sua concepção, posteriormente aprofundando em sua conexão com o campo de linguagens de programação através do cálculo lambda. Para isso, primeiro olhamos sua formulação não tipada, depois o cálculo lambda simplesmente tipado, suas extensões dentro do cubo lambda, e finalmente estudamos teorias com tipos dependentes.

Os objetivos principais da programação centrada em dados são representar, consultar e transformar dados. Um exemplo clássico é a linguagem *SQL* e ambientes relacionados, porém neste projeto nos concentramos em ferramentas de análise exploratória de dados como a biblioteca *pandas*. Reimplementamos um sistema de *data frame* na linguagem com tipos dependentes *Lean 4* baseando-o em uma implementação existente em *Idris 2*. O resultado foi uma linguagem de domínio específico com verificação formal para carregar, consultar e transformar dados tabulares.

Concluímos que escrever programas verificados pode ser complexo, incluindo nossa implementação de *data frames* e seu uso, mas pode ser benéfico para alguns casos. Processos exploratórios podem ser melhorados ao recusarem operações semanticamente inválidas, mas nossa solução se destaca principalmente na escrita de operações de dados para casos de uso críticos. Um próximo passo para o projeto é implementar um motor de transpilação capaz de transformar consultas de nossa linguagem em sistemas conhecidos como *pandas* e *R*.

Palavras-chave: tipos dependentes. teoria dos tipos. programação centrada em dados.

Abstract

Eduardo Sandalo Porto. **An exploration of dependent types for data-centric programming**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2024.

Type systems define rules that assign types to expressions of a language, and are the subject of study of type theory. The first theory of types was devised by Bertrand Russel in the early 20th century to be a logical foundation of mathematics, although it couldn't fulfill its promise. Dependently-typed programming languages restore this goal through the Curry-Howard correspondence, summarized by the "propositions as types" motto: types correspond to logical propositions and programs to their proofs. As such, they can be used not only to write programs, but also to formally verify them.

The goal of this project is to study dependent types from the ground up and apply them to a real-world scenario, namely, data-centric programming. The study of types starts at its conception, later deepening into how it intertwined with the field of programming languages through the lambda calculus. To do so, we first look at its untyped formulation, then the simply-typed lambda calculus, its extensions within the lambda cube, and we finally study dependently-typed theories.

The primary goals of data-centric programming are to represent, query, and transform data. A classic example is the *SQL* language and related environments, however for this project we focused on exploratory data analysis tools like the *pandas* library. We re-implemented a data frame system in the dependently-typed language *Lean 4* basing it on an existing implementation in *Idris 2*. The result was a verified domain-specific language to load, query, and transform tabular data.

We conclude that writing verified programs can be tricky, including our data frame implementation and its usage, but can be beneficial for a few use cases. Exploratory processes can be improved by disallowing semantically-invalid operations, but our solution shines most in writing data operations for critical use-cases. A next step for the project is to implement a transpiling engine able to transform queries from our language into well-known systems like *pandas* and *R*.

Keywords: dependent types. type theory. data-centric programming.

List of Figures

List of Tables

List of Programs

Contents

Introduction

Background

Research within the field of programming languages (PL) is generally divided into three main categories: theory, design, and implementation. The first deals with the mathematical backing of programming languages, including formal syntax and semantic descriptions. The second is more hands-on, dealing with the way humans interact and think about programming. The third is about getting a language to run on a real computer, either compiled to another language or interpreted by an existing system.

One of the common themes in all three subfields of PL is **types**. They are a way to encode lightweight specification for parts of programs, providing modest formal verification (Pierce, 2002). Their mathematical foundations can get quite involved, and their study is called **type theory**. They are also extremely important to the design of a language, affecting every line of code written in it. Finally, they can be tricky to implement in more complex languages, and may require advancements in language implementation.

This means that types involve all main areas of programming language research, and are a good introduction to it. This project is an attempt to present, explore, and apply types in programming languages. To do it, we choose a specific topic in type theory to study and afterwards apply it to another field.

For this project, we chose to deepen into the field of **dependent types** and apply it to **data-centric programming**. Particularly, we wish to study the foundations of type theory, working up our way to reach dependent types to see how they fit the overarching story, and then utilize these types in a real-word scenario related to programming focused on data.

Thus, we divide this monograph into two parts: Part I contains the study of types from their conception to the development of dependent type theories; in Part II we implement a data frame library in the dependently-typed language Lean 4.

Chapter 1 is about the origin of type theory; Chapter 2 is about the untyped lambda calculus, a system of computation over which we will build type systems; Chapter 3 is about the simply-typed lambda calculus; and finally on Chapter 4 we discuss dependent types. For the second part, Chapter 5 is about data-centric programming and data frames, Chapter 6 presents our data frame implementation, and Chapter 7 concludes our implementation and the monograph.

For the rest of the introduction, we will briefly review the concepts of dependent types and data-centric programming.

Dependent types

In statically-typed programming languages, dependent types are types that can depend on values. That is, the language of types is able to use the language of (normally run-time) values for its definitions. This has many implications One example in the Lean programming language is the definition of a vector (`Vect`), which is a linked list with a fixed size. See the following example:

```
inductive Vect (α : Type u) : Nat → Type u where
  | nil  : Vect α 0
  | cons : α → Vect α n → Vect α (n + 1)
```

Vectors can be created via two constructors: `nil`, which creates an empty vector (of size 0), and `cons`, which creates a vector of size $n + 1$ from another vector of size n. `Vect` is treated as a type-level function (i.e., it returns a type) that receives as parameters another type, and more interestingly, a natural number, which is a value. In usual programming languages, both functional and imperative, this kind of computation is not expressible.

This is useful not only for the sake of expressivity, but also for practical formal verification. The Curry-Howard correspondence, discussed in Section 3.6 and Section 4.1.3, states that there is an isomorphism between certain type systems and logic systems. Specifically, the type system behind the Lean programming language is correspondent to higher-order logic, which allows the language to state and prove mathematical theorems. Below is a very simple example of this correspondence, proving *modus ponens* in Lean.

```
def modus_ponens {P Q : Prop} : P → (P → Q) → Q :=
  fun (p : P) (f : P → Q) => f p
```

So, dependent types allow us to both write programs and prove properties about these programs. This is extremely useful for any application critical enough to warrant verification, and being able to write programs and proofs in the same language is a great advantage over similar verification systems. As McKinna (2006) argues, dependently typed programs are proof carrying codes by their nature, and they provide a flexible means to provide verification up to the point needed. That is, they can offer both lightweight assertions and full-blown proven specifications. We discuss more how dependent types are relevant for programming in Chapter 4.

Data-centric programming

Programs centered on data have as their main purpose representing, querying and transforming data and data structures. The most commonly studied data-centric systems are databases, with programming done on data-centric languages like SQL. In our work, we focus on data frames as defined by McKinney (2010), which are flexible tabular data structures centered on data analysis and manipulation. Thus, our objective is to build a data frame library in Lean leveraging the power of dependent types. We go into deeper detail on this in Chapter 6, although the main idea will be to require and generate proofs of certain properties before for specific operations. Although we could go quite far through this lens, our work mostly uses dependent types for a primary feature, which is *column pointers*. They are data structures that can serve both as proof that a data frame contains a column and as an instruction on how to extract it.

Part I

Type theory

Chapter 1

Russell's Paradox and the theory of types

In this chapter, we present a brief history and description of Bertrand Russell's original theory of types, showing where the need for type theories stems from and the original principles that guided them. As we will see, it was shown after the conception of this work that the original theory was quite troublesome to work as a foundation of mathematics, but it still paved the way for how type theories are used in programming today.

1.1 Developments of set theory

The late 19th and early 20th century comprised many turning points in mathematics. While it is one of the oldest subjects of study, much of its development throughout human history was made informally and without much rigor. Although many results that are still useful today were already proven by then, very little of the current work on the foundations of mathematics had yet been formalized.

The search for formalized theories of mathematics and logic to serve as foundations was guided by and led to many developments in both fields. As Heijenoort (1967) compiles, a significant portion of this production happened between the years of 1879 and 1931, with the publication of *Begriffsschrift* by Gottlob Frege, which described a formula language for describing thought; and the papers which initially defined Kurt Gödel's incompleteness theorems. Within that period, lots of developments culminated in the creation of Bertrand Russell's **theory of types** after he found contradictions within one of the most commonly accepted formalizations of set theory at the time, defined by Georg Cantor in significant works between 1878 and 1885 (Ferreirós, 2023). Within those works is included what is now known as Cantor's Theorem, which has as a consequence that any set A has its cardinality strictly lesser than the cardinality of its power set $\mathcal{P}(A)$, that is, any set is smaller than its power set.

Cantor's proof for this theorem paved the way for **Russell's Paradox**, as Russell discusses in *The Principles of Mathematics* (B. Russell, 1903, Chapter 10), a seminal work and the first book written in English with a comprehensive study of the logical foundations of mathematics. Using contemporary language and the concept of naive set theory, we can derive this paradox assuming the unrestricted axiom of comprehension, which allows the definition of sets from any expressible formula (i.e. with no restrictions). Doing so, we can define the set of all sets that do not contain themselves

$$w = \{ x \mid x \notin x \}.$$

One might ask — *does w contain itself?* If we assume that it does, then it cannot contain itself, since it is the set of all sets that do not contain themselves. If we assume that it does not, then it must contain itself. Thus, we get a paradox, and we can be certain that accepting the aforementioned axiom leads to an inconsistent theory.

Since then, multiple solutions to this problem have been proposed. One such solution was initially informally described by Russell himself (B. Russell, 1903, Appendix B) in the section *The Doctrine of Types*, where he proposes a theory of types. This theory was later refined in *Mathematical logic as based on the theory of types* (B. Russell, 1908) and *Principia Mathematica* (Whitehead and B. A. Russell, 1927).

1.2 Russell's theory of types

This section presents aspects of the theory of types as developed by B. Russell (1908) and is also based on commentaries by Heijenoort (1967) in his discussion of the aforementioned paper. This theory presents itself as an alternative foundation for mathematics, and as such, does not take many of the traditional mathematical constructs for granted. It is not our goal to do an in-depth analysis and description of this work within this section, as other developments of type theory by Alonzo Church and branching results are more tightly related to the scope of this text, those of which will be presented in later sections.

After writing *The Doctrine of Types*, Russell initially dismissed his own idea by 1905, coming up with the *zigzag theory*, the *theory of limitation of size*, and the *no-classes theory*, focusing on the latter, but eventually acknowledged that it could be inadequate to a lot of classical mathematics. Coming back to the development of types, the author then published *Mathematical logic as based on the theory of types* in 1908.

B. Russell (1908) starts by introducing seven paradoxes, including Russell's Paradox, and then argues that each of them stems from what he calls *self-reference* or *reflexiveness*. Recall the w set, defined as the set of all sets that do not contain themselves. The set is defined by referencing all sets, which leads it to reference itself. If we decide that no set can be a member of itself, then w becomes the set of all sets, and we have to determine that it is not a member of itself. Therefore, w is not a set. As Russell states, this is only possible if the set of all sets cannot exist in the way the paradox requires; if we suppose that it does exist, then new sets that lie outside the rules we previously established can be formed.

So, we can define a rule that Russell calls the *vicious-circle principle* stating that anything (whether a proposition or another indeterminate abstraction) that involves all of a collection cannot be of that same collection, avoiding this and other paradoxes. That leads us to not being able to easily make statements referring to all of a given collection. For example, “all propositions are either true or false” cannot be a proposition itself. To deal with this, Russell discusses the difference between **all** and **any** — as we’ve determined, a proposition can’t refer to all propositions, but something like “a proposition is either true or false” is allowed by our rules. This statement refers to an undetermined variable that could be *any* proposition, allowing us to make general remarks on propositions without breaking previous rules. Although similar, *any* and *all* cannot be treated as exactly the same, and we cannot substitute any usage of one by the other, since *all* deals with bound (or quantified) variables and *any* with free variables.

These concepts are key to the central idea of this theory, which is to divide the universe into levels called **types**. Any logical object of a given type can only refer to other bound variables of its own type, or free variables. So, using *all* we do universal quantification over a type and *any* expresses an unspecified object, unrestricted by a type. This imposes a hierarchy such that there is a type 0, a type 1 with properties on values of type 0, a type 2 with properties on properties of values of type 0, and so on. Russell’s formulation of type theory in the aforementioned paper came to be known as *ramified*, because the type of functions depends both on the types of their arguments and on the types of bound variables contained within them. This would later prove to be a big issue for the theory, leading Russell to propose the *axiom of reducibility* to solve it, whose scope is beyond this text. It was also solved in other formulations of type theory, specifically, *simple type theory* within the context of the Alonzo Church’s **simply-typed lambda calculus**, which we’ll see in a later section.

Despite following a different direction than later developments, Russell’s theory of types was greatly influential to the foundations of mathematics and inspired many further works.

1.3 Simple type theory

Ramsey (1926) and other logicians of the time noticed that the paradoxes Russell originally intended to solve did not all stem from the same origin — Ramsey classified some as "logical" paradoxes, while others were considered "semantic". To him, the main issue with Russell’s theory of types was the way these two different kinds of paradoxes were unified by the *vicious-circle principle*. He then showed that by abandoning this principle, paradoxes of the “logical” kind could be solved, without the need for the axiom of reducibility since the ramified levels of Russell’s theory could be collapsed. This big change in how the theory worked led to the **simple theory of types** Gödel (1944), which is the idea that things that we think about or their symbolic expressions are put into groups called types. These groups include individuals, their properties, the relationships between individuals, the properties of these relationships, and so on. Also, sentences only make sense if the values they refer to or relate to are of types that fit together.

This paves the way for Alonzo Church’s **simply-typed lambda calculus** (Church, 1940), devised as a formulation of the simple theory of types built on Church’s *lambda calculus*. In the next section, we’ll explore the **untyped** lambda calculus so we’re prepared to take on its type-theoretic version, both with modern notation and improvements as described by Pierce (2002). These two concepts are the basis for understanding type theory's role in contemporary programming languages and newer features, such as the dependent types we wish to study.

Chapter 2

Untyped lambda calculus

This section is based on Barendregt (1985), Barendregt and Barendsen (2000), and Pierce (2002). The untyped λ-calculus is a formal system, a theory of functions, and a computational model that distills functions into two simple concepts — abstraction (or function definition) and application. It is *untyped* because it assigns no types to values of its theory, allowing any function to be applied to any other function. Its original inventors were seeking to both develop a theory of functions and to extend that same theory enough so that it could be a foundation for logic and mathematics, although attempts at this second goal failed, having been proven as an inconsistent theory for this purpose. That means, when using the untyped λ-calculus as a logical foundation for mathematics, it is possible to derive *false* ($\bot$) from *true* ($\top$) — there are other definitions for which the λ-calculus is consistent.

Nevertheless, the untyped λ-calculus found its purpose in the field of computability, serving as a mathematical basis for the notion of computation equivalent to the Turing Machine. In fact, the formalization of "effectively computable" given by Turing computability is equivalent to λ-definability, the property of a function to be defined in the λ-calculus. This means that the λ-calculus can be seen as a programming language, albeit simple, having been used as both a theoretical and a semantic basis for dozens of successful programming languages since the 1960s. It can especially be used in the specification of programming languages, language design and implementation, and the study of type systems.

2.1 Syntax

The language of this system is based on three building blocks: variables (`<var>`), which are usually sequences of alphabetical characters; abstraction, which defines new functions; and application, which allows the use of these functions. The language's syntax can be concisely described in Backus-Naur Form (BNF) as follows:

```
λ-term  <term>  ::=  <var>            (variable)
                 |   <term> <term>    (application)
                 |   λ <var>. <term>  (abstraction)
```

Expressions generated by this grammar are called λ-terms. See the following examples:

Example 2.1.1 (λ-terms)**.**

(a) x

(b) $x\ y$

(c) $\lambda x.x$

(d) $(\lambda a.b)(\lambda c.d)e$

(e) $(\lambda var.\,(\lambda bar.\,zar))(\lambda z.x)$

(f) $\lambda e.\lambda x.\lambda p.\lambda r.\, e\ x\ p\ r$

Note that application is left-associative and abstraction is right-associative, and application has higher precedence than abstraction. The next examples show a few slightly confusing λ-terms with and without the parenthesis defining their precedence:

Example 2.1.2 (λ-term precedence)**.**

(a) $\lambda x.\lambda y.\lambda z.\, x\ y\ z \equiv (\lambda x.(\lambda y.((x\ y)\ z)))$

(b) $\lambda x.\, x\ y\ \lambda w.\lambda z.\, z\ w \equiv (\lambda x.((x\ y)\,(\lambda w.(\lambda z.(z\ w)))))$

(c) $\lambda x.\, x\ \lambda y.\, y\ z \equiv (\lambda x.(x\,(\lambda y.\,(y\ z))))$

(d) $(\lambda x.\lambda y.\, x)\ y\ z \equiv (((\lambda x.\,(\lambda y.x)\ y)\ z)$

Expressions of the λ-calculus have a recursive structure, and as such, can be formally defined inductively. This is useful when writing inductive proofs over λ-terms.

Definition 2.1.3 (inductive definition of λ-terms)**.** *Given a set of variables $V = \{v_0, v_1, \ldots\}$ and the set of all λ-terms Λ, we define λ-terms inductively:*

(i) $x \in V \implies x \in \Lambda$

(ii) $M, N \in \Lambda \implies (MN) \in \Lambda$

(iii) $M \in \Lambda, x \in V \implies (\lambda x.M) \in \Lambda$

A more succinct inductive definition is by abstract syntax:

Definition 2.1.4 (definition of λ-terms by abstract syntax)**.**

$$\Lambda = V \mid \Lambda\Lambda \mid \lambda V.\Lambda$$

2.2 Reduction rules

The λ-calculus is a **rewriting system**, which means it is a system of formulas (expressions) and rules that dictate how these formulas or their sub-formulas can be rewritten (or reduced) by others. *λ-terms* can be reduced with, in one of the calculus' simplest forms, two rules: β-reduction ($\rightarrow_\beta$) and α-conversion ($=_\alpha$). As we define them, note that there are certain subtleties regarding the substitutions they perform — they'll be described shortly afterwards, with plenty of examples to illustrate their behavior.

Definition 2.2.1 (β-reduction)**.** *Given λ-terms M and N, where $M[x := N]$ is the term M with free occurrences of x substituted by N, we say a term β-reduces to another in one step:*

$$(\lambda x.M)N \to_\beta M[x := N]$$

To allow different reduction orders, β-reduction follows these rules, where Z is a λ-term:

(i) $M \to_\beta N \implies ZM \to_\beta ZN$

(ii) $M \to_\beta N \implies MZ \to_\beta NZ$

(iii) $M \to_\beta N \implies \lambda x.M \to_\beta \lambda x.N$

Definition 2.2.2 (α-conversion)**.** *Given a λ-term M and variables x and y where y does not occur in M, we say a term α-converts to another:*

$$\lambda x.M =_\alpha (\lambda y.M)[x := y]$$

These definitions are trickily written to avoid trouble with name clashes in expressions. We say a variable is free in a λ-term when it is not bound by an abstraction, that is, if it is x, then it is not a sub-term of any expression $\lambda x.\Lambda$; and we say an occurrence of a variable in a term happens when it *is* the term or occurs in one of its sub-terms. Let's see a few examples of λ-terms and their reductions:

Example 2.2.3 (identity function)**.** $\lambda x.x$
Example 2.2.4 (constant function)**.** $\lambda x.y$
Example 2.2.5.

$$(\lambda x.x)z \to_\beta z$$

$$(\lambda x.y)z \to_\beta y$$

In these three examples, we've defined a function that always returns its argument, a function that always ignores its argument and returns a fixed variable, and showed how they can be reduced. Notice how, differently from usual set-theoretic functions, application is not of the form $f(x)$ but fx.

Example 2.2.6 (multiple arguments)**.** $(\lambda f.\lambda x.fx)(\lambda x.x)x \to_\beta (\lambda x.x)x \to_\beta x$

Although all λ-abstractions contain exactly one argument, it is possible to chain them together to "simulate" functions of multiple arguments, like in the example above. Before the reductions, the example contained terms $(\lambda f.\lambda x.fx)$, $(\lambda x.x)$, and (x) — note that in these three terms, x refers to different things; in the first, it is a variable bound by $\lambda x.fx$, in the second it is bound by the identity function $\lambda x.x$, and in the third it is a free variable.

These rules are enough to define **computation**, which is simply a β-reduction to a λ-term. α-conversions generally do not define a computational step because both sides of the relation are considered syntactically identical (we can get from one to the other and back with only α-conversions) and they're mostly used to avoid name collision in reductions. See the following example:

Example 2.2.7 (necessity of α-conversion).

$$(\lambda f.\lambda x.fx)x \not\rightarrow_\beta \lambda x.xx$$

$$(\lambda f.\lambda x.fx)x =_\alpha (\lambda f.\lambda y.fy)x \rightarrow_\beta \lambda y.xy$$

As this example shows, the bound variable x has to be renamed before the free variable x can take its place; that leads us to think of α-conversion as a syntactical equivalence relation between λ-terms. Alternative syntaxes to the λ-calculus, such as De Bruijn notation, avoid the problem (and consequently α-conversion) by representing bound variables with indices.

2.3 Developments on reduction

To represent subsequent iterations of reduction, we define the reflexive transitive closure of $\rightarrow_\beta$, which is simply a β-reduction in 0 or more steps:

Definition 2.3.1 ($\twoheadrightarrow_\beta$). *We say a λ-term M β-reduces to N (in 0 or more steps):*

$$M \twoheadrightarrow_\beta N$$

It follows these rules:

(i) $M \rightarrow_\beta N \implies M \twoheadrightarrow_\beta N$

(ii) $M \twoheadrightarrow_\beta M$

(iii) $M \twoheadrightarrow_\beta N,\ N \twoheadrightarrow_\beta L \implies M \twoheadrightarrow_\beta L$

By repeatedly applying β-reductions, we can start to treat the λ-calculus as an actual programming language. As an initial example, we'll define the basic boolean values *True* and *False* within the calculus. Given expressions E_1 and E_2, which should be reduced when a given condition is true or false respectively, we can define this behavior as such:

Example 2.3.2 (booleans).

$$\mathit{True} \equiv \lambda x.\lambda y.x$$

$$\mathit{False} \equiv \lambda x.\lambda y.y$$

$$\mathit{True}\ E_1\ E_2 \rightarrow_\beta (\lambda y.E_1)\ E_2 \rightarrow_\beta E_1$$

$$\mathit{False}\ E_1\ E_2 \rightarrow_\beta (\lambda y.y)\ E_2 \rightarrow_\beta E_2$$

Using the notation on Definition 2.3.1, we can shorten the application steps and define a term for an *if-then-else* construct, which receives a boolean condition and reduces the corresponding expression.

Example 2.3.3 (if-then-else).

$$\mathit{True}\ E_1\ E_2 \twoheadrightarrow_\beta E_1$$

$$\mathit{False}\ E_1\ E_2 \twoheadrightarrow_\beta E_2$$

$$\mathit{If} \equiv \lambda b.\,\lambda E_1.\,\lambda E_2.\ b\ E_1\ E_2$$

$$\mathit{If}\ \mathit{True}\ E_1\ E_2 \twoheadrightarrow_\beta E_1$$
$$\mathit{If}\ \mathit{False}\ E_1\ E_2 \twoheadrightarrow_\beta E_2$$

Example 2.3.4 (different paths).

$$(\lambda x.x)(\lambda x.y)z \to_\beta (\lambda x.y)z \to_\beta y$$

$$(\lambda x.x)(\lambda x.y)z \to_\beta (\lambda x.x)y \to_\beta y$$

This example illustrates one of the key concepts in the λ-calculus and similar rewriting systems — there are different orders that reduction can be done in. The following theorem and next few examples show how and why this matters.

Theorem 2.3.5 (Church-Rosser). *If M, M_1, M_2, are λ-terms such that $M \twoheadrightarrow_\beta M_1$ and $M \twoheadrightarrow_\beta M_2$, then there exists a λ-term N such that both $M_1 \twoheadrightarrow_\beta N$ and $M_2 \twoheadrightarrow_\beta N$.*

In other words, if there is a reduction from M to N, there may be multiple paths that this reduction can take, and most importantly, any point of a path taken that hasn't reduced to N still has additional steps that will reduce to it. Therefore, inside the λ-calculus, different reduction paths cannot produce results that are not *equal* (as we will define later). The only reduction paths that will not result in N are those that remain continuously unfinished in their reduction (or computing):

Example 2.3.6 (loop). $(\lambda x.xx)(\lambda x.xx) \to_\beta (\lambda x.xx)(\lambda x.xx) \to_\beta (\lambda x.xx)(\lambda x.xx) \to_\beta \ldots$

Interestingly, the above example β-reduces to itself. In this case, there is only one possible β-reduction to be done, so any attempt to reduce it will *not* lead to a *non-reducible* λ-term. This is not always the case when considering β-reduction loops:

Example 2.3.7 (path matters).

$$\mathit{False}\ ((\lambda x.xx)\,(\lambda x.xx))\,(\lambda x.x) \to_\beta \mathit{False}\ ((\lambda x.xx)\,(\lambda x.xx))\,(\lambda x.x) \to_\beta \ldots$$

$$\mathit{False}\ ((\lambda x.xx)\,(\lambda x.xx))\,(\lambda x.x) \to_\beta (\lambda y.y)\,(\lambda x.x) \to_\beta \lambda x.x$$

In this example, where we apply a function that applies its argument to itself to itself, if we try to reduce $(\lambda x.xx)\,(\lambda x.xx)$ before applying it to False, we get the same λ-term — however, if we apply it to False without reducing it first, it will disappear, and we can reduce the expression to $\lambda x.x$. This leads us to believe that although different reduction paths *can* always lead to the same result, it does *not* mean that they always *will.*

Another peculiar feature of the λ-calculus is that reduction does not necessarily cause the resulting expression to be smaller than the previous expression. See the following example:

Example 2.3.8 (increasing expression).

$$(\lambda x.xxx)(\lambda x.xxx) \to_\beta (\lambda x.xxx)(\lambda x.xxx)(\lambda x.xxx) \to_\beta \ldots$$

Before going further, let us define equality on λ-terms as the congruence relation generated by $\rightarrow_\beta$:

Definition 2.3.9 ($=_\beta$)**.** *λ-terms M and N are β-convertible (or equal):*

$$M =_\beta N$$

if they follow:

(i) $M \twoheadrightarrow_\beta N \implies M =_\beta N$

(ii) $M =_\beta N \implies N =_\beta M$

(iii) $M =_\beta N, N =_\beta L \implies M =_\beta L$

Succinctly, two λ-terms are considered equal if one can be β-reduced to the other or vice-versa, or if they are both equal to the same third λ-term. Remember that terms that are α-convertible are also considered equal. See the following examples:

Example 2.3.10 (equality)**.**

$$(\lambda x.x)z =_\beta z$$

$$(\lambda x.y)z =_\beta (\lambda x.x)y$$

$$\lambda x.x =_\beta \lambda y.y$$

$$\mathit{False}\ ((\lambda x.xx)\,(\lambda x.xx))\ (\lambda x.x) =_\beta \lambda x.x$$

Note that two terms that are α-convertible are also β-convertible, but not necessarily the converse.

2.4 Normal forms

Going back to the idea of taking different paths of reduction, we see that some terms cannot be reduced any further. Therefore, we define the following:

Definition 2.4.1 (β-redex)**.** *A λ-term is called a β-**redex** if it can be β-reduced.*
Definition 2.4.2 (normal form)**.** *A λ-term is in **normal form** if it has no β-redexes.*

Some terms can be β-reduced iteratively until they can no longer be reduced, and that final term is their normal form. Not all terms have a normal form, as we saw with Example 2.3.6. From the Church-Rosser Theorem (2.3.5), we know that, if a term has a normal form, then it is unique and there is always a reduction path to reduce to it. Another important consequence of this theorem is that, if two terms have the same normal form, they are β-convertible.

2.5 Reduction strategies

As we saw in the past few examples, the order in which a λ-term is reduced matters to its result. A set of rules that defines the order of β-reduction is called a **reduction strategy**. The most common reduction strategies are the *leftmost* strategy (also called *lazy* or *normal*) and the *rightmost* strategy (also called *eager* or *applicative*).

Definition 2.5.1 (left). *Given a λ-term M with β-redex subterms A and B, we say that A is to the left of B if B is a subterm of A or they are the same depth in M but A is to the left in the expression.*

Example 2.5.2 (same-depth left). $E\,((\lambda x.x)\,x)\,((\lambda y.y)\,y) \rightarrow_{\beta-left} E\,x\,((\lambda y.y)\,y)$

Example 2.5.3 (different-depth left). $E\,((\lambda x.x)\,((\lambda y.y)\,y)) \rightarrow_{\beta-left} E\,((\lambda y.y)\,y)$

Definition 2.5.4 (leftmost strategy). *Given a λ-term not in normal form, the leftmost reduction strategy always picks its leftmost β-redex to be reduced.*

Definition 2.5.5 (right). *Given a λ-term M with β-redex subterms A and B, we say that B is to the right of A if B is a subterm of A or they are the same depth in M but B is to the right in the expression.*

Example 2.5.6 (same-depth right). $E\,((\lambda x.x)\,x)\,((\lambda y.y)\,y) \rightarrow_{\beta-right} E\,((\lambda x.x)\,x)\,y$

Example 2.5.7 (different-depth right). $E\,((\lambda x.x)\,((\lambda y.y)\,y)) \rightarrow_{\beta-right} E\,((\lambda x.x)\,y)$

Definition 2.5.8 (rightmost strategy). *Akin to leftmost strategy, but to the right.*

The leftmost strategy is also called *lazy* because it avoids performing computation until its result is required. For instance, when complex expressions are passed as parameters to a function, these expressions undergo β-reduction just when their results are required, or at the end of the computation if they're not needed and we keep looking for a normal form. Although Haskell is a well-known language that uses a variation of it, it is not commonly used in modern programming languages due to efficiency and optimization issues caused by the lack of sharing reduction results. Its most important feature to the theory of the λ-calculus is the following theorem, due to logician Haskell Curry:

Theorem 2.5.9. *If a λ-term M has a normal form N, then M always reduces to N in the leftmost strategy.*

This theorem seems to solve any possible problems regarding normal forms, as we can use it to find them, although that is not the case since it requires a term to **have** a normal form. Unfortunately, determining whether a λ-term has a normal form is one of the earliest problems to have been proven undecidable (Church, 1936).

The rightmost strategy is much simpler to implement and optimize, and as such is used in most mainstream programming languages. In fact, all of the top 20 programming languages in the Stack Overflow Developer Survey (Stack Overflow, 2024) implement the rightmost strategy, excluding markup languages for which evaluation strategies are not applicable, and SQL, which, as a declarative language, can choose its evaluation strategy accordingly. While the leftmost evaluation strategy can be seen as reducing trees of expressions starting from the root and going down towards the leaves, the rightmost strategy can be thought of reducing trees of expressions starting from their leaves and going up to the root, which avoids the need to keep subtrees in terms that have already reduced. This strategy can calculate the results of expressions that are not needed — for example, if parameter a is passed to another function, but this function never uses the value of a, it will still be evaluated; if a results in a loop, then the program will also loop indefinitely.

2.6 Equivalence with Turing Machines

What we've seen up until now already provides us enough intuition on how Turing computability and λ-definability are equivalent notions. While Turing Machines execute computational steps through their transition functions, λ-terms do so by performing β-reductions. While the former loops their instructions until they reach a stopping state, the latter performs β-reduction until it reaches a normal form. A Turing machine diverges when it can't reach a stopping state, and a λ-term does so when it can't reach a normal form. It is possible to prove they are equivalent by simulating the execution of a Turing Machine as the calculation of a normal form in the λ-calculus, and by simulating the reduction of a λ-term in a Turing Machine. One such proof was given originally by Turing (1936), but for the purposes of this text, it is enough to intuit their similarities.

As we've seen, the λ-calculus looks like a programming language, and should behave like one. To think of the λ-calculus as a programming language, we still need to discuss how to deal with data and control, that is, the fundamental data structures we manipulate in programs, and how can we control the execution of these programs according to their states.

2.7 Dealing with data: Church encoding

Since the λ-calculus we've seen up until now can only really have as "values" functions and free variables, we can't simply use concepts external to the language such as numbers and strings, commonly done in other programming languages. It is possible, however, to simulate such concepts within the language as λ-encodings. Next, we'll see how to encode the natural numbers and some of their operations within the calculus:

Definition 2.7.1 (repeated application)**.** *The λ-term f can be applied multiple times:*

$$f^0 x \equiv x$$

$$f^n x \equiv \underbrace{f\,(f\,(f\,(\ldots f}_{n \text{ times}}\,x)))$$

Definition 2.7.2 (Church encoding)**.** *The Church encoding of a natural number n is represented by $\lceil n \rceil$:*

$$\lceil n \rceil \equiv \lambda f.\lambda x.f^n x$$

Example 2.7.3 (Church-encoded numerals)**.**

(a) $\lceil 0 \rceil \equiv \lambda f.\lambda x.x$

(b) $\lceil 1 \rceil \equiv \lambda f.\lambda x.fx$

(c) $\lceil 3 \rceil \equiv \lambda f.\lambda x.f\,(f\,(fx))$

(d) $\lceil 5 \rceil \equiv \lambda f.\lambda x.f\,(f\,(f\,(f\,(fx))))$

In other words, the Church encoding of a natural number n is a function that receives another function f and an initial "zero" value x, which is then applied to f n times. Among many other possible operations, we now define addition, multiplication, zero check, and predecessor functions on Church-encoded numbers:

Definition 2.7.4 (addition). $add \equiv \lambda m.\, \lambda n.\, \lambda f.\, \lambda x.\, m\, f\, (n\, f\, x)$
Definition 2.7.5 (multiplication). $mul \equiv \lambda m.\, \lambda n.\, \lambda f.\, \lambda x.\, m\, (\lambda x.\, n\, f\, x)\, x$
Definition 2.7.6 (zero check). $isZero \equiv \lambda n.\, n\, (\lambda x.\, False)\, True$
Definition 2.7.7 (predecessor). $pred \equiv \lambda n.\lambda f.\lambda x.\, n\, (\lambda g.\lambda h.\, h\, (g f))\, (\lambda u.\, x)\, (\lambda u.u)$

Intuitively, the addition function creates a new number that is similar to *m* but has its initial value incremented *n* times. The multiplication function increments *n* times for every increment in *m*. The zero check function instantiates a number with an initial value of True that gets replaced by a False in any subsequent increment. The predecessor function is much harder to intuit, so we'll just accept it. It is possible to prove by induction on the structure of a Church-encoded numeral that these operations have the same results as their counterparts defined on the natural numbers, though that is beyond our current scope. We will look at control strategies in the λ-calculus as an example of a larger program before returning to the topic of encoded numerals.

2.8 Dealing with control: Recursion

The mechanism of controlled looping (or just control) in the λ-calculus is **recursion**. You might have noticed that there is no recursive primitive in the theory, and the fact is that we don't need it to be a primitive.

Theorem 2.8.1 (fixed-point combinator). *For every λ-term F, there is a* ***fixed-point combinator*** *Y such that*

$$YF =_\beta F(YF) =_\beta F(F(YF)) =_\beta \ldots$$

Definition 2.8.2 (Y combinator). *A fixed-point combinator for every λ-term is the* ***Y combinator****:*

$$Y \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

Fixed-point combinators can be thought of as a way to allow self-reference, so terms can refer to themselves. If $F \equiv \lambda f.E$, then the resulting YF is the term E with itself bound to the variable f, so when E substitutes for f, it is actually substituting it with itself. It might seem like a function generated by the Y combinator will keep reducing forever and it may happen under specific reduction strategies. However, under the leftmost (lazy) strategy, the recursive call will stop being reduced when the base case is reached.

There are other fixed-point combinators apart from the Y combinator with different properties. Note that, regarding the Y combinator, although $YF =_\beta F(YF)$, it is not true that $YF \twoheadrightarrow_\beta F(YF)$. A fixed-point combinator that has this property is Θ, defined by Alan Turing:

Definition 2.8.3 (Turing's fixed-point combinator).

$$\Theta \equiv (\lambda x.\lambda y.(y(xxy)))(\lambda x.\lambda y.(y(xxy)))$$

Coming back to Church-encoded numerals, we can define the factorial function using recursion like the following:

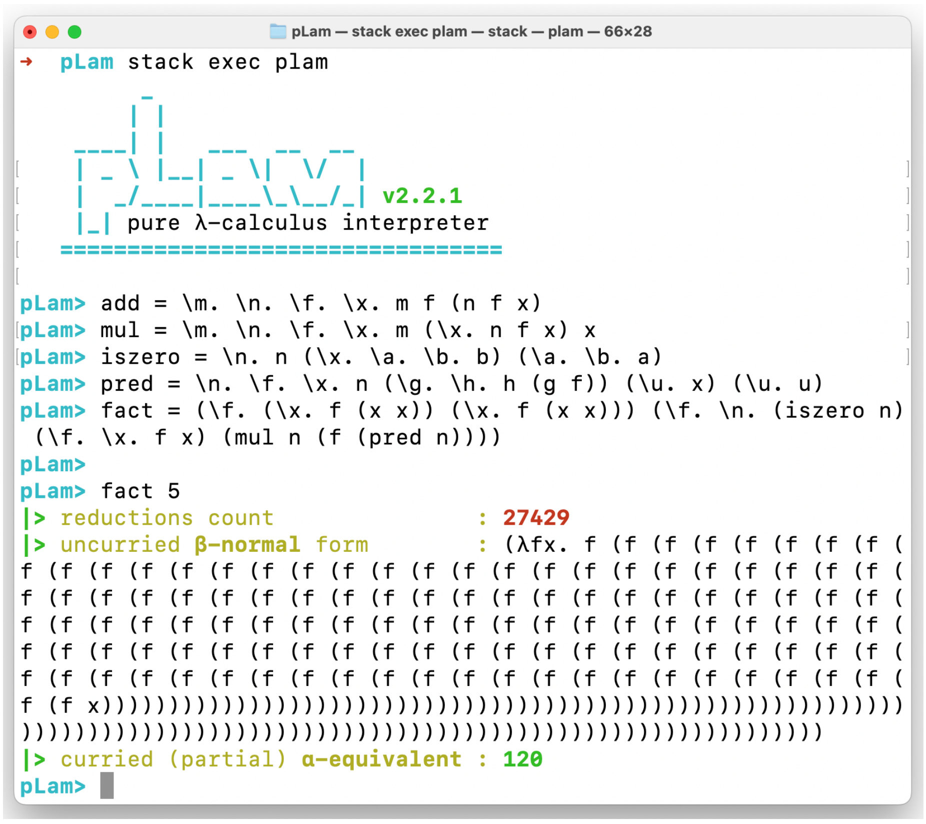
Example 2.8.4 (factorial)**.**

$$\mathit{factorial} \equiv Y\,(\lambda f.\lambda n.\,(\mathit{isZero}\,n)\,\lceil 1 \rceil\,(\mathit{mul}\,n\,(f\,(\mathit{pred}\,n))))$$

This definition is surprisingly similar to a more usual definition of the factorial as such:

$$\text{factorial}(n) = \text{if } n \leq 1 \text{ then } 1 \text{ else } n \cdot \text{factorial}(n-1)$$

The main difference is that the base of the recursion is not at 1 but at 0, but since the returned value 1 is the multiplicative identity, the result doesn't change. In summary, the λ-encoded factorial receives itself as parameter f through the Y combinator, then receives a number n and checks if it is zero — if it is, return the encoding of 1, and if it is not, multiply n by the factorial of $n-1$.



```
➜  pLam stack exec plam
          _
         | |
     ____| |   ___  __  __
    | _ \ |__| _ \|  \/  |
    |  _/____|____\_\__/_| v2.2.1
    |_| pure λ-calculus interpreter
   =================================

pLam> add = \m. \n. \f. \x. m f (n f x)
pLam> mul = \m. \n. \f. \x. m (\x. n f x) x
pLam> iszero = \n. n (\x. \a. \b. b) (\a. \b. a)
pLam> pred = \n. \f. \x. n (\g. \h. h (g f)) (\u. x) (\u. u)
pLam> fact = (\f. (\x. f (x x)) (\x. f (x x))) (\f. \n. (iszero n)
 (\f. \x. f x) (mul n (f (pred n))))
pLam>
pLam> fact 5
|> reductions count              : 27429
|> uncurried β-normal form       : (λfx. f (f (f (f (f (f (f (f (
f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (
f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (
f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (
f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (
f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (
f (f x))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
|> curried (partial) α-equivalent : 120
pLam> 
```

Figure 2.1: *Execution of the λ-encoded factorial of 5 in pLam.*

Figure 2.1 shows the result of the λ-encoded factorial function for the number 5 in the leftmost strategy using version 2.2.1 of the software pLam (Lovničκi, 2018), a tool designed to explore computations in the λ-calculus.

2.9 Developments on data: Scott encoding

The representation of data structures within the λ-calculus can be further improved using Scott encondings, developed by Dana Scott. Recall the concept of *algebraic data types* in functional programming languages, which includes *product types* and *sum types*. Product types work similarly to structures in imperative programming languages, in the sense that they "glue" together different types into a single one so they can be accessed from a single data structure. The simplest product type is `Pair`, which can be represented in Haskell as Program 2.1 — this code creates a new data type `Pair` that is polymorphic over `a` and `b`, and then creates a constructor also called `Pair` that can be used to create values of this structure, containing both an `a` and a `b`.

Program 2.1 The `Pair` product type in Haskell.

```haskell
data Pair a b = Pair a b
```

Slightly more interesting are sum types, which represent a choice between a fixed amount of types, that is, a tagged value of the sum type between T_1, T_2, ..., T_n is of and can only be of one of these types. The simplest example is the `Either` data type in Program 2.2, which represents a choice between an `a` constructed by `Left` or a `b` constructed by `Right`.

Program 2.2 The `Either` sum type in Haskell.

```haskell
data Either a b = Left a | Right b
```

The untyped λ-calculus, as its name suggests, does not have types, so we cannot really implement algebraic data types in it, though we can emulate them. Products are implemented as functions with parameters as each element's value, which can be accessed using projection functions. See the implementation of Pair in the λ-calculus:

Definition 2.9.1 (pair). *The pair of values a and b are created with the Pair function:*

$$Pair \equiv \lambda a.\lambda b.\lambda p.\, p\ a\ b$$

The value to the left of a Pair can be accessed using the fst function and the right with snd:

(i) $fst \equiv \lambda p.\, p\ True$

(ii) $snd \equiv \lambda p.\, p\ False$

Example 2.9.2 (pair of 1 and 2).

$$Pair\ \lceil 1 \rceil\ \lceil 2 \rceil$$

Example 2.9.3 (first of pair of 1 and 2).

$$fst\ (Pair\ \lceil 1 \rceil\ \lceil 2 \rceil) =_\beta \lceil 1 \rceil$$

Scott-encoded products can be generalized to simply receive their n values $x_1, \ldots, x_n$ as their first parameters, followed by a projection function p that selects the i^{th} desired value; in the case of pairs, the projection can simply pick the first element like in fst or the second element like in snd.

Scott-encoded sums can be seen as generalizations of booleans, possibly to more than 2 values. Similar to what we saw in Haskell, Either can be implemented like the following:

Definition 2.9.4 (either). *The left variant of an Either can be constructed using the Left function, and the right variant with Right:*

(i) $\mathit{Left} \equiv \lambda v.\lambda l.\lambda r.\, l\, v$

(ii) $\mathit{Right} \equiv \lambda v.\lambda l.\lambda r.\, r\, v$

Example 2.9.5 (left with 1). $\mathit{Left}\, \lceil 1 \rceil$
Example 2.9.6 (right with 2). $\mathit{Right}\, \lceil 2 \rceil$
Example 2.9.7 (double if left triple if right).

$$\mathit{doubleLeftTripleRight} \equiv \lambda x.\, x\, (\lambda v.\, \mathit{mul}\, v\, \lceil 2 \rceil)\, (\lambda v.\, \mathit{mul}\, v\, \lceil 3 \rceil)$$

Example 2.9.8 (triple 4). $\mathit{doubleLeftTripleRight}\, (\mathit{Right}\, \lceil 4 \rceil) =_\beta \lceil 12 \rceil$

As we saw, sums in the λ-calculus are created by choosing the appropriate constructor for the desired variant, and the extraction of a value in a sum is done by supplying a function for each possible variant of that sum. Note that this is very similar to pattern matching in functional programming languages.

We can use these techniques to define Scott-encoded natural numbers similarly to how it would be done in Haskell, like in Program 2.3.

Program 2.3 A definition for natural numbers based on Peano arithmetic in Haskell.

```haskell
data Nat = Zero | Succ Nat
```

Definition 2.9.9 (Scott-encoded natural numbers). *The Scott encoding* $\lfloor n \rfloor$ *of a natural number n is either zero (Zero) or the successor (Succ) of another natural number:*

$$\mathit{Zero} \equiv \lambda z.\lambda s.\, z$$

$$\mathit{Succ} \equiv \lambda n.\, \lambda z.\lambda s.\, s$$

Example 2.9.10 (Scott-encoded numerals).

(a) $\lfloor 0 \rfloor \equiv \mathit{Zero}$

(b) $\lfloor 1 \rfloor \equiv \mathit{Succ}\, \mathit{Zero}$

(c) $\lfloor 3 \rfloor \equiv \mathit{Succ}\, (\mathit{Succ}\, (\mathit{Succ}\, \mathit{Zero}))$

(d) $\lfloor 5 \rfloor \equiv \mathit{Succ}\, (\mathit{Succ}\, (\mathit{Succ}\, (\mathit{Succ}\, (\mathit{Succ}\, \mathit{Zero}))))$

This definition also works by repeated applications, but it's defined on top of sums and products, allowing us to use the same techniques as we use with them. Namely, we can define some operations over the natural numbers recursively, similar to how it would be done for Peano arithmetic:

Definition 2.9.11 (addition). $add \equiv Y\, \lambda r.\lambda m.\lambda n.\, m\, n\, (\lambda p.\, Succ\, (r\; p\; n))$
Definition 2.9.12 (multiplication). $mul \equiv Y\, \lambda r.\lambda m.\lambda n.\, m\, Zero\, (\lambda p.\, add\, n\, (r\; p\; n))$
Definition 2.9.13 (zero check). $isZero \equiv \lambda n.\, n\, True\, False$
Definition 2.9.14 (predecessor). $pred \equiv \lambda n.n\, Zero\, (\lambda p.p)$

The addition function pattern matches on m; if it is zero, then it returns n; if it is the successor of p, then it returns the successor of the addition between p and n. Multiplication pattern matches on m; if it is zero, return zero; if it is the successor of p, return the addition between n and the multiplication of p and n. The zero check patterns match on n; if it is zero return True and if it is not return False. The predecessor function is intuitive this time: if n is zero, return zero, if it is the successor of p, return p.

Using the `Nat` type we defined earlier in Program 2.3, we can implement these same functions in Haskell as in Program 2.4. If we continue developing more complicated Scott-encoded values, we can eventually represent lists as sums with variants “nil” and “cons” where “cons” stores a value and another list, use them to represent textual strings as lists of numbers that represent characters, represent matrices as lists of lists, and many other higher level programming concepts.

Program 2.4 Operations on Scott-encoded natural numbers in Haskell.

```haskell
add m n = case m of
  Zero -> n
  Succ p -> Succ (add p n)

mul m n = case m of
  Zero -> Zero
  Succ p -> add n (mul p n)

isZero n = case n of
  Zero -> True
  Succ p -> False

pred n = case n of
  Zero -> Zero
  Succ p -> p
```

As we discussed, the Scott encoding is very useful to represent intricate data structures. However, it may not be enough in cases where we desire to encode structures with properties that are more desirable when implemented outside of the λ-calculus, such as better performance when calculated by primitives in a Central Processing Unit (CPU). In the next section, we will discuss possible extensions to the untyped λ-calculus.

2.10 Extensions

The λ-calculus can be extended in its syntax, its reduction rules, and in the domain of its values; in this section, we’ll see examples of all three.

2.10.1 *Let* syntax

Functional programming languages tend to have an additional construct for declaring variables called the *let* syntax. It can be added to the syntax of the untyped λ-calculus without changing any of the system's semantic properties as such:

Definition 2.10.1 (let syntax)**.** *Given λ-terms M and E and a variable x, the let syntax is defined as:*

$$(let\, x = M\ in\, E) \equiv (\lambda x.E)M$$

or, similarly:

$$(let\, x = M\ in\, E) \equiv E[x\ := M]$$

The second definition of the let syntax, although β-convertible to the first in the untyped λ-calculus, is very important to **typed** λ-calculi and can specify different behavior.

2.10.2 η-conversion

The calculus we've studied up until now is also called the $\lambda\beta$-calculus, as its theoretical basis relies on β-reduction. There are other calculi such as the $\lambda\beta\eta$-calculus, which contains an additional rule called η-conversion:

Definition 2.10.2 (η-conversion)**.** *Given a λ-term F and a variable x that does not occur free in F, then the following λ-terms are η-convertible:*

$$\lambda x.Fx =_\eta F$$

This rule is called an *extensionality* rule because it allows the proof that, given λ-terms F and G and a variable x that is neither free in F nor in G, then $Fx = Gx \implies F = G$. It is also possible to prove that this rule is consistent with the axioms of the $\lambda\beta$-calculus, so its addition won't break any of the interesting results for the theory.

2.10.3 δ-rules

The system can also be extended with δ-rules representing functions that are external to the λ-calculus; for example, integer addition based on computation performed by a CPU, or boolean values, or strings, etc. Before we define these, we will add an extension to the syntax and the domain of the λ-calculus to allow constants in the language.

Definition 2.10.3 (λ-terms with constants)**.** *Given a set of constants C, the set of λ-terms with constants in C named $\Lambda(C)$ is described by abstract syntax:*

$$\Lambda(C) = C \mid V \mid \Lambda(C)\,\Lambda(C) \mid \lambda V.\Lambda(C)$$

This extension to the calculus allows us to have constants such as $+$, or $*$, or *true* and *false* and *if*, which will be quite useful to represent external functions. See the definition of the δ-rules so we can examine a couple of examples afterwards:

Definition 2.10.4 (δ-rules)**.** *Let $X \subset \Lambda$ be a set of λ-terms in normal form with no free variables. Usually, these terms are constants in the syntax of the calculus, so $X \subseteq C$. Let*

$f : X^k \to \Lambda$ be an external function. This function can be represented in the $\lambda\delta$-calculus as δ-rules with the following additions:

(i) A special constant is selected and given a name δ, or δ_f.

(ii) New reduction rules are added to the calculus, where M_1, ..., M_k are λ-terms in X:

$$\delta_f M_1 \dots M_k \to f(M_1, \dots, M_k)$$

Note that the above is not a single rule, but a rule schema — for every collection of M_1, ..., M_k in X a new δ-rule is added; also, $f(M_1, \dots, M_k)$ is not an expression in the calculus, but its result is. The notions of reduction $\to_{\beta\delta}$ and $\twoheadrightarrow_{\beta\delta}$ are defined similarly as before. The constraints on X are necessary to keep the Church-Rosser theorem from completely breaking — a theorem due to Gerd Mitschke is that the notion of reduction in $\twoheadrightarrow_{\beta\delta}$ also satisfies the Church-Rosser property when defined over a function on a set of λ-terms in normal form with no free variables. Note, however, that the choice of functions to define using δ-rules affects certain properties of the calculus, specifically, it might lead it no longer satisfying Church-Rosser when not following the constraints of Mitschke's theorem. Regarding reduction strategies, another theorem is that if $M \twoheadrightarrow_{\beta\delta} N$ and N is in normal form, then M always reduces to N in the leftmost strategy.

To conclude this section, we will define a few operations over the boolean values "true" and "false" using δ-rules, and subsequently, over the set of integers $\mathbb{Z}$.

Example 2.10.5 (booleans as δ-rules). *Select the following constants in C:*

true, ***false***, ***not***, ***and***, ***ite*** *(if-then-else)*

We introduce the following δ-rules:

not true $\to$ ***false***

not false $\to$ ***true***

and true true $\to$ ***true***

and true false $\to$ ***false***

and false true $\to$ ***false***

and false false $\to$ ***false***

ite true $\to$ *True (recall True $\equiv \lambda x.\lambda y.x$)*

ite false $\to$ *False (recall False $\equiv \lambda x.\lambda y.y$)*

Example 2.10.6 (integers as δ-rules). *For each integer $n \in \mathbb{Z}$, select a constant in C and name it $[n]$. Select constants* ***add***, ***sub***, ***mul***, ***divide***, ***error***, *and* ***equal***, *and assume there are external functions $+$, $-$, $\times$, $\div$, and $=$. We introduce the following δ-rules for $m, n \in \mathbb{Z}$:*

add $[m]\,[n] \to [m+n]$

sub $[m]\,[n] \to [m-n]$

mul $[m]\,[n] \to [m \times n]$

$\boldsymbol{div}\,[m]\,[n] \to [m \div n]$, *if* $n \neq 0$

$\boldsymbol{div}\,[m]\,[0] \to \boldsymbol{error}$

$\boldsymbol{equal}\,[m]\,[m] \to \boldsymbol{true}$

$\boldsymbol{equal}\,[m]\,[n] \to \boldsymbol{false}$, *if* $m \neq n$

We can also add rules akin to $\boldsymbol{add}\,[m]\,\boldsymbol{error} \to \boldsymbol{error}$.

Extending the λ-calculus like we did in this section is fundamental to the calculi we will discuss throughout the next sections; specifically, the addition of constants, external functions and domains is crucial to the simply-typed λ-calculus.

2.11 Typed lambda calculi

The main idea of including types to the λ-calculus is adding some sort of language within the calculus to describe the possible values an expression can take. That is, if a given expression E if *of* a given type τ, then τ describes all the values that E can take. Within the λ-calculus, a **type system** is a logical system that checks or assigns a type to each λ-term, and **type theory** is the field of mathematics concerned with studying type systems.

Adding types to the λ-calculus is useful for many applications, of which we will discuss two: programming and logic.

2.11.1 Types in programming

Programming languages as tools for writing programs should be human-centered, meaning they should guide people towards writing the programs they wish. When writing functions in a given language, it is essential for a programmer to communicate its expected inputs and outputs in some form. One way to do so is through a *specification*, which MORGAN (1990) describes as a contract specifying what a computer should do. Partial specification can be achieved through types, which communicate the expected shape of data, or values, of a language. The simply-typed λ-calculus adds basic types, treated as constants; and function types, which recursively contain an input and an output type. There are many other variations of the λ-calculus based on other type theories, including the dependently-typed calculi crucial to this monograph. We present here a short and informal way of thinking about types in programming languages.

An example of specification given by types can be done over the operations on Church-encoded and Scott-encoded natural numbers. In the untyped λ-calculus, they are predictable when applied to correctly encoded data, but may have completely different behavior when applied to any other values. Abstractions in this calculus have no restrictions over their parameters, so any expression can be applied to any expression. Although the following λ-term can be reduced, it is not clear what is meant by the addition between the *mul* and *div* functions.

$$\text{add mul div}$$

If type constants and function types existed in this language, we could specify that *add* is of a function type with a numerical input and a numerical output. Assume there is some *Nat* type inhabited by all Scott-encoded naturals, and there is a function type $A \to B$ where A and B are types. Then, the type of *add* would be

$$\text{Nat} \to (\text{Nat} \to \text{Nat})$$

That is, *add* is a function with a *Nat* parameter that returns a function with another *Nat* parameter that finally returns a third *Nat*. Customarily, $\to$ is treated as right-associative, so we could omit the parenthesis. Also, the λ-term represented by a type is usually written next to it, separated by a colon ($:$).

$$\text{add} : \text{Nat} \to \text{Nat} \to \text{Nat}$$

Informally, this shows us that typed λ-calculi have two languages embedded within them: the language of values (terms, expressions) and the language of types. The former is the code executed by a computer, and the latter is some specification over its behavior. The example above shows a very minimal specification within a very simple *type system*, since it specifies that addition occurs over natural numbers, but not what it means for two numbers to be added. More complicated type systems are more expressive, in the sense that they are able to communicate more information as specification, both to programmers and to the program itself.

This is part of what makes typing so useful for programming languages — they offer lightweight, programmable specifications to code. More advanced type systems have recently garnered attention in software engineering because their expressivity allows specification to be as thorough as a programmer needs it to be. For instance, the specification of a program that controls a nuclear reactor requires much more care than a simple routine script, and advanced type systems are able to deal with both situations.

2.11.2 Types in logic

As mentioned in the start of Chapter 2, the untyped λ-calculus was unfit as a logical foundation for mathematics. The main reason for this is its Turing-completeness, specifically, the fact that there exists non-terminating reductions for certain terms. These terms, when directly interpreted as a logical system, can lead to Russell's Paradox due to the same reason discussed in Section 1.1, which is the possibility of self-reference. Assuming a subset of the calculus where every λ-term has a normal form is not enough, since this subset is in most cases non-computable, as per the Scott-Curry Theorem (Barendregt, 1985).

A way to deal with this problem is to assign a type to every λ-term proven by some type system to be normalizing, and reject any term which does not have a type. This is a trade-off — although non-termination will become impossible, there are normalizing terms in the calculus that may be reject in a given type system. Considering this and the discussion on Section 2.11.1, a second measure of expressivity of a type system is given by the amount of terminating λ-terms that it includes. As we'll see in later sections, the simply-typed λ-calculus is as expressive and is logically equivalent to intuitionistic propositional logic, and dependently-typed calculi are equivalent to stronger systems.

Chapter 3

Simply-typed lambda calculus

In this section, we will discuss modern formulations of the simply-typed λ-calculus (Church, 1940) as based on Barendregt, Dekkers, *et al.* (2013) and Pierce (2002).

The types of this system, called simple types, are syntactic objects. They are built from type constants (also called type atoms) using the function type, represented by the operator $\to$. Each valid λ-term in the language is assigned a type, and the main idea behind the system is that if a λ-term M is of type $A \to B$ and N is of type A, then the application $M\,N$ is allowed and its β-reduction will be of type B. This system is denoted by $\lambda_\to$.

3.1 Syntax

The language of simple types can be defined via a BNF, like we did with the λ-calculus:

```
Simple type  <type>  ::=  <atom>              (type atom)
                      |   <type> → <type>     (function type)
```

Accordingly, the set can be defined inductively and by abstract syntax:

Definition 3.1.1 (set of simple types). *Let $\mathbb{A}$ be a non-empty set of constants, where each element is called a type atom. The set of simple types over $\mathbb{A}$, called $\mathbb{T}^{\mathbb{A}}$ (or just $\mathbb{T}$) is defined inductively:*

(i) $\alpha \in \mathbb{A} \implies \alpha \in \mathbb{T}^{\mathbb{A}}$

(ii) $A, B \in \mathbb{T}^{\mathbb{A}} \implies (A \to B) \in \mathbb{T}^{\mathbb{A}}$

Definition 3.1.2 (set of simple types by abstract syntax).

$$\mathbb{T}^{\mathbb{A}} = \mathbb{A} \mid \mathbb{T}^{\mathbb{A}} \to \mathbb{T}^{\mathbb{A}}$$

As mentioned earlier, function types are right-associative. See the following type examples:

Example 3.1.3. *Let $\mathbb{A} = \{Int, String\}$. The following types are in $\mathbb{T}^{\mathbb{A}}$:*

(i) Int

(ii) $Int \to String$

(iii) $String \to (String \to Int) \to Int$

(iv) $Int \to Int \to Int$

Each well-typed λ-term can be assigned a type:

Definition 3.1.4 (type assignment statement). *If $M \in \Lambda$ and $T \in \mathbb{T}$, the statement $M : T$ is read as "M in T" and means the λ-term M is of type T. M is called the subject of the statement and T is called the predicate.*

3.2 Typing rules

This very simple syntax is enough to represent simple types, but we haven't yet discussed how to construct new types in $\mathbb{T}$ for specific λ-terms. We do so by deriving λ-terms step-by-step, in a bottom-up approach using 3 rules over a *basis*:

Definition 3.2.1 (declarations and basis).

*(i) A **declaration** is a (type assignment) statement with a variable as subject.*

*(ii) A **basis**, also called a **typing context**, is a set of declarations with distinct variables. It is usually denoted as Γ.*

Example 3.2.2 (basis).

(i) $\{\}$

(ii) $\{x : Int\}$

(iii) $\{y : String, z : Int \to String\}$

Given a basis, we can derive type assignment statements:

Definition 3.2.3 ($\lambda_\to$ rules). *A statement $M : T$ is derivable from a basis Γ, written*

$$\Gamma \vdash_{\lambda_\to} M : T$$

(or just $\Gamma \vdash M : T$) if it can be constructed from the following rules:

(i) $(x : T) \in \Gamma \implies \Gamma \vdash x : A$

(ii) $\Gamma \vdash M : (A \to B),\ \ \Gamma \vdash N : A \implies \Gamma \vdash (M\,N) : B$

(iii) $\Gamma, x : A \vdash M : B \implies \Gamma \vdash (\lambda x.M) : (A \to B)$

These rules are frequently presented in *Gentzen-style* trees as the following definition. In this style, the top of the line contains n premises, and when all of them hold, their conclusion under the line holds as well. Each of these rules is named after the syntactical variant of the λ-term it constructs.

Definition 3.2.4 ($\lambda_\to$ rules in Gentzen-style).

$$\frac{x:T \in \Gamma}{\Gamma \vdash x:T} \quad \text{(var rule)}$$

The variable rule gives each variable a type from the typing context, and if the variable is not in the context, it cannot be given a type.

$$\frac{\Gamma \vdash M:A \to B \qquad \Gamma \vdash N:A}{\Gamma \vdash MN:B} \quad \text{(app rule)}$$

Mentioned earlier as the main idea behind the system of the simply typed λ-calculus, the application rule states that given a function of type $A \to B$ and a value of type A, applying it will result in a value of type B.

$$\frac{\Gamma, x:A \vdash M:B}{\Gamma \vdash \lambda x.M:A \to B} \quad \text{(abs rule)}$$

The abstraction rule is used to assign types to functions. Note that $\Gamma, x:A$ means that $x:A$ must be in the typing context, and when its consequence does not have $x:A$ explicitly, it is removed from the basis. The term M may have free occurrences of x, so the typing context has to be extended - if it already has a type for x, it should be replaced. This is similar to the idea of variable shadowing in modern programming languages.

Below are a few example of *proof trees* showing the derivation of types for a few λ-terms.

Example 3.2.5 (type derivation)**.** *In the examples below, $\alpha, \beta \in \mathbb{T}$. Note that $\vdash F$ is the same as $\emptyset \vdash F$.*

(i) Type derivation of $\lambda x.x$ (with $\Gamma = \{x:\alpha\}$)

$$\dfrac{\dfrac{x:\alpha \in \{x:\alpha\}}{\{x:\alpha\} \vdash x:\alpha}\,\textit{(var)}}{\vdash \lambda x.x:\alpha \to \alpha}\,\textit{(abs)}$$

(ii) Type derivation of $\lambda x.\lambda y.x$ (with $\Gamma = \{x:\alpha, y:\beta\}$)

$$\dfrac{\dfrac{\dfrac{x:\alpha \in \{x:\alpha, y:\beta\}}{\{x:\alpha, y:\beta\} \vdash x:\alpha}\,\textit{(var)}}{\{x:\alpha\} \vdash \lambda y.x:\beta \to \alpha}\,\textit{(abs)}}{\vdash \lambda x.\lambda y.x \;:\; \alpha \to \beta \to \alpha}\,\textit{(abs)}$$

(iii) Type derivation of fx (with $\Gamma = \{f:\alpha \to \beta, x:\alpha\}$)

$$\dfrac{\dfrac{f:\alpha \to \beta \in \{f:\alpha \to \beta, x:\alpha\}}{\{f:\alpha \to \beta, x:\alpha\} \vdash f:\alpha \to \beta}\,\textit{(var)} \qquad \dfrac{x:\alpha \in \{f:\alpha \to \beta, x:\alpha\}}{\{f:\alpha \to \beta, x:\alpha\} \vdash x:\alpha}\,\textit{(var)}}{\{f:\alpha \to \beta, x:\alpha\} \vdash fx:\beta}\,\textit{(app)}$$

(iv) Type derivation of $\lambda f.\lambda x.f(fx)$ (with $\Gamma = \{f:\alpha \to \alpha, x:\alpha\}$)

$$\dfrac{\dfrac{\dfrac{\dfrac{f:\alpha \to \alpha \in \{f:\alpha \to \alpha, x:\alpha\}}{\{f:\alpha \to \alpha, x:\alpha\} \vdash f:\alpha \to \alpha} \qquad \dfrac{\dfrac{f:\alpha \to \alpha \in \{f:\alpha \to \alpha, x:\alpha\}}{\{f:\alpha \to \alpha, x:\alpha\} \vdash f:\alpha \to \alpha} \qquad \dfrac{x:\alpha \in \{f:\alpha \to \alpha, x:\alpha\}}{\{f:\alpha \to \alpha, x:\alpha\} \vdash x:\alpha}}{\{f:\alpha \to \alpha, x:\alpha\} \vdash fx:\alpha}}{\{f:\alpha \to \alpha, x:\alpha\} \vdash f(fx):\alpha}}{\{f:\alpha \to \alpha\} \vdash \lambda x.f(fx):\alpha \to \alpha}}{\vdash \lambda f.\lambda x.f(fx):(\alpha \to \alpha) \to \alpha \to \alpha}$$

As mentioned earlier, these rules are enough to guarantee the termination of repeated β-reduction to any typeable λ-term. This property is called **strong normalization**. Of course, this is not obvious and needs to be proven, although that is outside the scope of this text. One such proof can be found in Ziliani (2012).

A consequence of strong-normalization is that not all λ-terms are representable in $\lambda_{\to}$. Non-terminating terms, like the increasing expression in Example 2.3.8, cannot be assigned types and thus, cannot exist. One very important feature of this system is that self-application is invalid, that is, a λ-term xx cannot be typed. Its type would have to be a function of some form $A \to B$, but it should also be able to receive itself as parameter, so A would have to be $A = A \to B$, expanding to an infinite type. Since the Y combinator (Definition 2.8.2) uses self-application, it cannot exist in this system.

In fact, general recursion cannot be encoded in this formulation of $\lambda_{\to}$, as it would lead to losing strong normalization, being reduced to the halting problem. Recursion can be restored either by abandoning strong normalization or by restricting it to known terminating strategies. The former can be done extending typing with what Barendregt, Dekkers, *et al.* (2013) calls *recursive types*; and the latter by using *structural recursion*, a kind of recursion that iterates through a structure decreasing in size, eventually reaching a base case and provably terminating.

3.3 Typing styles

What we've seen up until now is called the **Curry style** of typing, or typing *à la Curry*. It is characterized by types only appearing in type assignment statements, that is, the language of types and terms are completely separate. This can make certain algorithms relating to types harder or impossible to implement. It is denoted by $\lambda_{\to}^{\text{Curry}}$.

Another style of typing, called **Church style** (or *typing à la Church*) and denoted $\lambda_{\to}^{\text{Church}}$, has the argument of every abstraction typed.[1] This style changes the language of the calculus to *pseudo λ-terms*, which only differ to the original definition by having explicit types in abstractions. See the following BNF definition:

```
Pseudo λ-term   <term>   ::=   <var>                      (variable)
                          |    <term> <term>              (application)
                          |    λ <var> : <type>. <term>   (abstraction)
```

Similarly, it can be defined by abstract syntax:

Definition 3.3.1 (λ-terms with typing à la Church by abstract syntax)**.**

$$\Lambda = V \mid \Lambda\Lambda \mid \lambda V : \mathbb{T}.\,\Lambda$$

The abstraction rule is different in this system. Its only change is the explicit type

[1] Barendregt, Dekkers, *et al.* (2013) calls this *typing à la de Bruijn*. Other sources, like Barendregt (1993), call it Church style. To avoid confusion with terminology commonly used in the field, we will keep the second name. The first source defines Church style systems as having every variable (free and bound) with a type attached, and de Bruijn style only attaches types to bound variables. The authors also state *"these two systems are basically isomorphic"*.

added to the argument:

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A.\, M : A \to B} \quad \text{(abs rule)}$$

This small change has an important consequence. In $\lambda_{\to}^{\text{Curry}}$, a derivation for a type τ holds for every type $\tau \in \mathbb{T}$, but in $\lambda_{\to}^{\text{Church}}$ it is only for a specific τ. For example, if $\alpha, \beta \in \mathbb{T}$ and we wish to type the identity function, then $\vdash_{\text{Curry}} \lambda x.x : \alpha \to \alpha$ is derivable, but so is $\vdash_{\text{Curry}} \lambda x.x : \beta \to \beta$. Yet, in $\lambda_{\to}^{\text{Church}}$, we can assign to $\lambda x : \alpha.\, x$ the type $\alpha \to \alpha$, but not $\beta \to \beta$.

Developments of the simply-typed λ-calculus such as the polymorphic simply typed λ-calculus and dependently-typed λ-calculi also differ by the Curry and Church styles. As the Curry style is less restrictive, some algorithmic problems within it are undecidable, while decidable in the Church variant. Next, we will discuss a few common problems in typing.

3.4 Algorithmic problems in typing

A few natural problems appear when thinking about type systems, and many of them are extremely useful in programming languages. Three problems commonly discussed in type systems are **type checking**, **type inference**, and **type inhabitation**. Within type inference, we can separate the problems of **typability** and **type reconstruction**, and within type inhabitation we can also discuss **enumeration**.

Type checking is the problem of checking whether a λ-term is well-typed, that is, given a basis Γ, λ-term M and type A, checking if $\Gamma \vdash M : A$ holds. In most statically-typed programming languages, a "type error" usually means that type checking failed, that is, $\Gamma \vdash M : A$ does not hold. Stylistically (and not formally) it can be summarized as

$$\Gamma \vdash M : A \ ? \quad \text{(type checking)}$$

Type inference is when we wish to find a type (or the most general type) for a given λ-term M from a given basis Γ. Similarly, typability is the problem of determining whether there is some type that can be assigned to M, and reconstruction is finding every possible type and basis for M. Note that nomenclature for this case can vary in the literature.

$$\begin{aligned} \Gamma \vdash M : ? \quad & \text{(type inference)} \\ \exists A, \Gamma\ [\Gamma \vdash M : A]\ ? \quad & \text{(typability)} \\ ? \vdash M : ? \quad & \text{(type reconstruction)} \end{aligned}$$

Finally, type inhabitation is finding whether a type A is inhabited by some term M in a given basis Γ, that is, if there exists some M of type A derived by Γ. Type enumeration is similarly about determining all possible λ-terms of a type A derived by Γ.

$$\begin{aligned} \exists M\ [\Gamma \vdash M : A]\ ? \quad & \text{(type inhabitation)} \\ \Gamma \vdash ? : A \quad & \text{(type enumeration)} \end{aligned}$$

Interestingly, all three problems and their variations are decidable in the simply typed λ-calculus both in Curry and Church style. This is not true for all type systems — for instance, type checking and typability are undecidable for the *polymorphic* simply typed λ-calculus à la Curry, known as System F (Wells, 1999).

3.5 Data types

As we did earlier in Section 2.7, we can also represent the types of data. In this section, we will discuss a few ways to encode the types of the same data as before. All definitions will be over the system $\lambda^0_{\to}$, which is given by the set of type constants $\mathbb{A} = \{\alpha\}$. This system only contains one type constant.

3.5.1 Boolean values

Earlier, we gave definitions for λ-terms *True* and *False*, which were $\text{True} \equiv \lambda x.\lambda y.x$ and $\text{False} \equiv \lambda x.\lambda y.y$. Notice how both terms are functions to a function of a value. Both can trivially be shown to be of type $\alpha \to \alpha \to \alpha$.

Definition 3.5.1 (the *Bool* type)**.** *The type of boolean values is*

$$Bool \equiv \alpha \to \alpha \to \alpha$$

3.5.2 Natural numbers

Recall the Church encoding of a natural number n as $\lceil n \rceil \equiv \lambda f.\lambda x.f^n x$. Terms of this format obviously receive a function f, and a value x which can be applied to f, and this result must also be able to be applied again. As such, the type of Church-encoded numbers is $(\alpha \to \alpha) \to \alpha \to \alpha$.

Definition 3.5.2 (the *Nat* type)**.** *The type of natural numbers is*

$$Nat \equiv (\alpha \to \alpha) \to \alpha \to \alpha$$

The addition function $\text{add} \equiv \lambda m.\lambda n.\lambda f.\lambda x.\, m\, f\, (n\, f\, x)$ receives as parameters two values of type $(\alpha \to \alpha) \to \alpha \to \alpha$, a function of type $\alpha \to \alpha$ and a value of type α, finally returning α. So, its type is:

$$\text{add} : ((\alpha \to \alpha) \to \alpha \to \alpha) \to ((\alpha \to \alpha) \to \alpha \to \alpha) \to (\alpha \to \alpha) \to \alpha \to \alpha$$

But, $\text{Nat} \equiv (\alpha \to \alpha) \to \alpha \to \alpha$, so *add*'s type is equivalent to

$$\text{add} : \text{Nat} \to \text{Nat} \to (\alpha \to \alpha) \to \alpha \to \alpha$$

Since the $\to$ operator is right-associative, the return type $(\alpha \to \alpha) \to \alpha \to \alpha$ can also be rewritten as *Nat*.

$$\text{add} : \text{Nat} \to \text{Nat} \to \text{Nat}$$

Similarly, the typing statements for the other operations previously defined can be shown to be:

$$\begin{aligned} \text{mul} &: \text{Nat} \to \text{Nat} \to \text{Nat} \\ \text{isZero} &: \text{Nat} \to \text{Bool} \\ \text{pred} &: \text{Nat} \to \text{Nat} \end{aligned}$$

3.5.3 Product

Pairs of values (Definition 2.9.1) can be seen as elements of a Cartesian product between the domain of their first and second values. The *Pair* function receives a boolean value which, when *True*, selected the first value and when *False*, the second.

In a given type system over the λ-calculus with reduction rules represented by R, we can define the triple of terms $\langle Pair, fst, snd \rangle$ as an R-pairing, and define the type $A \times B$ as the result of *Pair* and arguments of *fst* and *snd*.

If $\tau, \sigma \in \mathbb{T}$, assume *Pair* is of type $\tau \to \sigma \to \text{Bool} \to ?$, with *?* unspecified. If the *fst* projection is chosen to extract a value from a pair, then the resulting type of *Pair* has to be τ, but if *snd* is chosen, it has to be σ. So, for this function to be well-typed, it must return one of two types, and determining which one it is depends on the value of the argument of type *Bool*. The simply typed λ-calculus can neither have types depend on terms nor on other types, so this cannot be a valid type in the system.

If, however, both values of a pair are of the same type, then a *Pair* can be well-typed.

Definition 3.5.3 (pair of same types)**.**

$$\begin{aligned} \alpha \times \alpha &\equiv Bool \to \alpha \\ Pair &: \alpha \to \alpha \to (\alpha \times \alpha) \\ fst &: (\alpha \times \alpha) \to \alpha \\ snd &: (\alpha \times \alpha) \to \alpha \end{aligned}$$

If η-conversion (Definition 2.10.2) is added to the system, then there are other less straight-forward definitions for *Pair*, *fst*, *snd* that form an R-pairing for products $A \times B$ for any two types A and B specifically in system $\lambda^0_\to$, though they are beyond the scope of this text.[2]

Products can also be added to the language as an extension of its semantics and type system. This can be seen in Pierce (2002).

Below, we extend the set of λ-terms by abstract syntax to add operators for constructing and deconstructing **product types**, and the relevant typing rules over them. This is based on Sørensen and Urzyczyn (2006).

Definition 3.5.4 (λ-terms with product types)**.**

$$\Lambda = \ldots \mid \langle \Lambda, \Lambda \rangle \mid fst(\Lambda) \mid snd(\Lambda)$$

[2] For an in-depth discussion, see Barendregt (1974).

Given statements $M : A$ and $N : B$, then $\langle M, N \rangle$ is the pair between M and N and is of type $A \times B$. If M is of a product type, then $\mathrm{fst}(M)$ extracts its first value, and $\mathrm{snd}(M)$ its second. Two reductions must be added for this to be true:

$$\begin{aligned} \mathrm{fst}\,(\langle M, N \rangle) &\to M \\ \mathrm{snd}\,(\langle M, N \rangle) &\to N \end{aligned}$$

Below are the formalized typing rules.

Definition 3.5.5 (product typing rules)**.**

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B}{\Gamma \vdash \langle M, N \rangle : A \times B}\ \textit{(pair rule)}$$

$$\frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \mathit{fst}\,(M) : A}\ \textit{(fst rule)} \qquad \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \mathit{snd}\,(M) : B}\ \textit{(snd rule)}$$

3.5.4 Sum

A similar discussion of the *Either* constructors in the untyped λ-calculus (Definition 2.9.4) can be done, though we will skip it and present the extension of sum types to the simply typed λ-calculus.

Sum types must have a way to create variants, namely *left* and *right* constructors, and a way to perform case analysis, that is, perform some operation when they are of the left or right variant. To do so, we introduce syntax for *lft* and *rgt*, and for case analysis *case.*

Definition 3.5.6 (λ-terms with sum types)**.**

$$\Lambda = \dots \mid \mathit{lft}^{\mathbb{T}+\mathbb{T}}\,(\Lambda) \mid \mathit{rgt}^{\mathbb{T}+\mathbb{T}}\,(\Lambda) \mid \mathit{case}\,(\Lambda; V.\Lambda; V.\Lambda)$$

Note that the left and right constructors must also specify which is the sum type they build. The *case* syntax creates "pseudo" abstractions, where the variable before the dot will be the extracted values from the sum and substituted into each branch. Here are the added reduction rules:

$$\begin{aligned} \mathit{case}\,(\mathit{lft}^{A+B}\,(N); x.K; y.L) &\to K[x := N] \\ \mathit{case}\,(\mathit{rgt}^{A+B}\,(N); x.K; y.L) &\to L[y := N] \end{aligned}$$

Briefly explained, if the first argument to *case* is of the left variant, substitute x in K by N and return it. If it is of the right variant, substitute y in L by N and return it. Next, are the sum typing rules:

Definition 3.5.7 (sum typing rules)**.**

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \mathit{lft}^{A+B}\,(M) : A + B}\ \textit{(lft rule)} \qquad \frac{\Gamma \vdash M : B}{\Gamma \vdash \mathit{rgt}^{A+B}\,(M) : A + B}\ \textit{(rgt rule)}$$

$$\frac{\Gamma \vdash L : A + B \qquad \Gamma, x : A \vdash M : \rho \qquad \Gamma, y : B \vdash N : \rho}{\Gamma \vdash \mathit{case}(L; x.M; y.N) : \rho}\ \textit{(case rule)}$$

3.6 The Curry-Howard correspondence

The Curry-Howard correspondence, also known as Curry-de Bruijn-Howard isomorphism and a few other names, is an observation made by many mathematicians and logicians on the connection of type theory and formal logic. One of its many formulations is given by Howard (1969).

This isomorphism is nowadays associated with the motto *"propositions as types, programs as proofs"* (Wadler, 2015). The central idea is that there is a correspondence between typed programs and logic: types correspond one-to-one to some logical proposition, and if there is some λ-term (program) of that type, then it acts as a proof to the corresponding proposition. For instance, if a programmer writes a program of a type correspondent to $P \wedge Q \supset P \vee Q$, then this statement is proven to hold.

Describing it with more formal syntax, we say that for many logical systems L, there is a type theory λ_L and a map that translates formulas A of logic L into types $[A]$ of λ_L with some typing context Γ_A that "explains" A such that

$$\vdash_L A \iff \Gamma_A \vdash_{\lambda_L} M : [A], \text{ for some } M$$

The above reads as "proposition A is provable in system L if and only if basis Γ_A derives a type $[A]$ for some λ-term M in system λ_L". In other words, the proposition that a type τ maps to holds if and only if τ is inhabited.

If the map $[\cdot]$ is also extended to translate a proof D in system L into its corresponding λ-term, then we can also state

$$\vdash_L A, \text{ with proof } D \iff \Gamma_A \vdash_{\lambda_L} [D] : [A]$$

The simply-typed λ-calculus is equivalent to *implicational intuitionistic propositional logic*, that is, intuitionistic logic with only the implication ($\supset$) operator. In fact, the inference rules of this logic are very similar to the typing rules of $\lambda_{\rightarrow}$. See its syntax and rules:

Definition 3.6.1 (syntax of the implicational propositional logic in BNF).

Formula	`<form>`	::=	`<var>`
		|	`<form>` $\supset$ `<form>`
Variable	`<var>`	::=	p
		|	`<var>'`

Definition 3.6.2 (rules of the implicational propositional logic). *Let Γ be a set of formulas and A a formula. We say A is derivable from Γ, written $\Gamma \vdash_{PROP} A$, if $\Gamma \vdash A$ can be produced from these rules:*

(a) $A \in \Gamma \implies \Gamma \vdash A$

(b) $\Gamma \vdash A \supset B,\ \Gamma \vdash A \implies \Gamma \vdash B$

(c) $\Gamma, A \vdash B \implies \Gamma \vdash A \supset B$

It is easy to see the similarities between these rules and the typing rules of $\lambda_\to$ (Definition 3.2.4). We can define the mapping $[A]$ to be p if $A = p$ and $[P] \to [Q]$ if $A = P \supset Q$; we also define $\Gamma_A = \emptyset$. The following theorem states the Curry-Howard correspondence between the implicational intuitionistic propositional logic and the simply typed λ-calculus. A proof can be found in Barendregt, Dekkers, *et al.* (2013).

Theorem 3.6.3 (Curry-Howard correspondence for $\lambda_\to$)**.** *Let A be a proposition and Δ a set of propositions.*

$$\Delta \vdash_{PROP} A \iff [\Delta] \vdash_{\lambda_\to} M : [A], \text{ for some } M.$$

Recall the *abs* and *app* rules from Definition 3.2.4. The abstraction rules works as an *introduction rule* for $\to$ and application as an *elimination rule*. Similarly, $\lambda_\to$ extended by product types (3.5.5) and sum types (3.5.7) also present this introduction/elimination duality: a product is introduced by the *pair* rule and eliminated by the *fst* and *snd* rules; a sum is introduced by the *lft* and *rgt* rules and eliminated by the *case* rule. These introductions and eliminations behave the same way as the rules of the full intuitionistic propositional logic. This means that the simply typed λ-calculus with product and sum types corresponds to this logic, and whatever holds for one holds for the other. Even more important, syntactical and semantic features of each system have direct correspondence, as shown in Table 3.1.

Intuitionistic logic	**Type theory**
Proposition P	Type P
Proposition $P \supset Q$	Type $P \to Q$
Proposition $P \wedge Q$	Type $P \times Q$
Proposition $P \vee Q$	Type $P + Q$
Proof of proposition P	λ-term of type P
Proposition P is provable	Type P is inhabited

Table 3.1: *Correspondence between intuitionistic logic and type theory (Pierce, 2002).*

As we'll briefly see in Chapter 4, type systems with more features and rules are equivalent to stronger logic systems.

3.7 Shortcomings

One of the main goals of a strongly normalizing type system is to limit the language of λ-terms to some strongly normalizing subset. These systems try to maintain as many terminating (or well-behaving) terms as possible, though computable systems will always prohibit some terminating terms due to Rice's Theorem (Rice, 1953). So, a "strong" type system tries to maximize the subset of terminating terms.

The simply typed λ-calculus cannot type many terminating terms, and specially, many useful terminating terms. We've already seen how it cannot type products and sums in Section 3.5; now, we'll see another important example.

Recall the identity function $\lambda x.x$. It must be assigned some type $\alpha \to \alpha$, and α cannot change from one application to the other. Consider the following λ-term:

$$(\lambda f.\langle f \lceil 1 \rceil, f\,\text{True}\rangle)(\lambda x.x)$$

In the untyped λ-calculus with reduction rules for pairs, this term reduces to $\langle \lceil 1 \rceil, \text{True}\rangle$. However, the original term cannot be typed in $\lambda_{\rightarrow}$. Assume we assign to $\lambda x.x$ the type $Nat \rightarrow Nat$, then, the first element of the pair is well-typed, but not the second. Analogously, the same can be done for the second element.

The reason this term cannot be typed is that the identity function must assume different types depending on what argument it receives. This can be solved through polymorphism, as done in System F.

Chapter 4

Dependent types and terms

Having thoroughly discussed some general concepts about type systems and the simply typed λ-calculus, we will now briefly mention a few other stronger type systems and discuss dependent types in the context of the calculus of constructions (Coquand and Huet, 1988). This chapter is based on Barendregt (1991) and Nederpelt and Geuvers (2014).

4.1 The lambda cube

The λ-cube is a system defined by Barendregt (1991) that describes eight different type systems through the combination of three features. Its formulation can be seen in Figure 4.1. Starting with the simply typed λ-calculus $\lambda_{\to}$ on the bottom left, each arrow in a dimension points to another system that adds a specific feature:

- **x-axis ($\rightarrow$)**: types can depend on terms
- **y-axis ($\uparrow$)**: terms can depend on types
- **z-axis ($\nearrow$)**: types can depend on types

$$\begin{tikzcd}
 & \lambda\omega \arrow[rr] & & \lambda C \\
\lambda 2 \arrow[ur] \arrow[rr] & & \lambda P2 \arrow[ur] & \\
 & \lambda\underline{\omega} \arrow[uu, dashed] \arrow[rr, dashed] & & \lambda P\underline{\omega} \arrow[uu] \\
\lambda_{\to} \arrow[uu] \arrow[ur, dashed] \arrow[rr] & & \lambda P \arrow[uu] \arrow[ur] &
\end{tikzcd}$$

Figure 4.1: *The λ-cube (Barendregt, 1991) describing features added to type systems.*

Each of the three features create or extend some new form of abstraction related to its improvement. Below we briefly see the additions of each feature's "representative" system. For a complete formulation of each system, see Nederpelt and Geuvers, 2014.

4.1.1 System $\lambda 2$

$\lambda 2$ (known as *System F*, or *polymorphic typed λ-calculus*, or *second-order typed λ-calculus*) adds a mechanism of quantification over types. This means a type can be passed to another function as a parameter. It allows what is called *function polymorphism* in traditional programming languages. As such, it is also a way of solving the issue on the type of the identity function discussed in Section 3.7. In Church style, the new identity function is of the following form:

$$\lambda\alpha : *.\, \lambda x : \alpha.\, x$$

The asterisk ($*$) can be treated as “the type of all types”. Since this system is only second-order, it is not susceptible to Russell’s Paradox (Section 1.1). Applying types to the identity function, we can have a well-typed version of the earlier example:

$$(\lambda f.\langle f\ \mathit{Nat}\ \lceil 1 \rceil, f\ \mathit{Bool}\ \mathit{True}\rangle)(\lambda\alpha : *.\, \lambda x : \alpha . x)$$

The system as of now has no way of expressing the type of this new identity function. Any instance of α in its type will be treated as a free variable. We must then add quantification to the language of types through Π-*types*. Finally, the typing rules added to System F are the following:

Definition 4.1.1 (System F typing rules).

$$\frac{\Gamma, \alpha : * \vdash M : A}{\Gamma \vdash (\lambda\alpha : *.\, M) : (\Pi\alpha : *.\, A)}\ (\mathit{abs}_2\ \mathit{rule}) \qquad \frac{\Gamma \vdash M : (\Pi\alpha : *.\, A) \qquad \Gamma \vdash B : *}{\Gamma \vdash M\, B : A[\alpha := B]}\ (\mathit{app}_2\ \mathit{rule})$$

As mentioned earlier, type checking and inference are non-computable in this system. These two algorithms are fundamental to programming languages that wish to add type polymorphism, so System F cannot be added directly to them. A variation of this system, called the *Hindley-Milner type system*, restricts $\lambda 2$ by differentiating between “poly-types” and “mono-types”, granting decidability for both type checking and inference. Programming languages in the ML family usually implement this system.

4.1.2 System $\lambda\underline{\omega}$

This system, also known as System F$\underline{\omega}$, allows types to depend on other types. This is done through the addition of *type constructors*, which are functions (abstractions) of types to types. It allows *type polymorphism* in traditional programming languages. Examples of this include lists: a list of type *List α* can be instantiated to *List Nat*, *List Bool*, etc.

The type of types $*$ can now be mapped to others in the form $* \to *$. To build over it, we must specify something akin to the “type of the type of types”, which we will call *sort* and represent with □. Here are a few examples:

$$
\begin{aligned}
List &: * \to * \\
Product &: * \to * \to * \\
* &: \Box \\
* \to * &: \Box
\end{aligned}
$$

We won't discuss this system any further, but below are included its typing rules for completeness.

Definition 4.1.2 (System F$\underline{\omega}$ typing rules)**.**

(i) *(sort)* $\emptyset \vdash * : \Box$

(ii) *(var)* $\dfrac{\Gamma \vdash A : s}{\Gamma, x{:}A \vdash x{:}A}$ *if* $x \notin \Gamma$

(iii) *(weak)* $\dfrac{\Gamma \vdash A : B \qquad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B}$ *if* $x \notin \Gamma$

(iv) *(form)* $\dfrac{\Gamma \vdash A : s \qquad \Gamma \vdash B : s}{\Gamma \vdash A \to B : s}$

(v) *(app)* $\dfrac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\,N : B}$

(vi) *(abs)* $\dfrac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash A \to B : s}{\Gamma \vdash \lambda x{:}A.\, M : A \to B}$

(vii) *(conv)* $\dfrac{\Gamma \vdash A : B \qquad \Gamma \vdash B' : s}{\Gamma \vdash A : B'}$ *if* $B =_\beta B'$

4.1.3 System λP

This system allows types to depend on terms. In most of the literature (and in the title of this monograph) the notion of **dependent types** refer exactly to this concept. These types have a general form

$$\lambda x{:}A.\, M$$

where M and A are types but x is a variable. This abstraction and its result, which is a type, must then depend on the term x.

λP is special over the two previously discussed extensions to $\lambda_\to$ due to its isomorphism to first-order implicational intuitionistic logic by the Curry-Howard correspondence. When extended with *dependent* product and sum types, it is fully equivalent to first-order intuitionistic logic.

The most important new typing rule is the following, which allows Π-types as terms:

$$\frac{\Gamma \vdash A : * \qquad \Gamma, x{:}A \vdash B : s}{\Gamma \vdash (\Pi x{:}A.\, B) : s}\ \textbf{(form)}$$

This system completely substitutes the → operator with Π-types. Most other rules are similar to $\lambda\underline{\omega}$ except for *abs* and *app*, as shown in Definition 4.1.3. Table 4.1 shows the corresponding terms and notions between λP and implicational first-order intuitionistic logic.

Intuitionistic logic	**Type theory**
Proposition P	Type $P : *$
Set S	Type $S : *$
D is a proof of P	$D : P$
$A \supset B$	$\Pi x : A.\, B$
$\forall_{x \in S}(P(x))$	$\Pi x : S.\, Px$
$\supset$-*elim* and *intro*	*app* and *abs* rules
$\forall$-*elim* and *intro*	*app* and *abs* rules

Table 4.1: *Curry-Howard correspondence for λP.*

Definition 4.1.3 (λP typing rules).

(i) *(sort)* $\varnothing \vdash * : \square$

(ii) *(var)* $\dfrac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}$ *if* $x \notin \Gamma$

(iii) *(weak)* $\dfrac{\Gamma \vdash A : B \qquad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B}$ *if* $x \notin \Gamma$

(iv) ***(form)*** $\dfrac{\Gamma \vdash A : * \qquad \Gamma, x : A \vdash B : s}{\Gamma \vdash (\Pi x : A.\, B) : s}$

(v) ***(app)*** $\dfrac{\Gamma \vdash M : \Pi x : A.\, B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\, N : B[x := N]}$

(vi) ***(abs)*** $\dfrac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash \Pi x : A.\, B : s}{\Gamma \vdash (\lambda x : A.\, M) : (\Pi x : A.\, B)}$

(vii) *(conv)* $\dfrac{\Gamma \vdash A : B \qquad \Gamma \vdash B' : s}{\Gamma \vdash A : B'}$ *if* $B =_\beta B'$

4.2 Calculus of constructions

The calculus of constructions, also known as CoC or λC, is the resulting system when mixing together the central ideas of the three previously discussed systems. It allows terms to depend on terms ($\lambda_\to$), terms to depend on types ($\lambda 2$), types to depend on types ($\lambda\underline{\omega}$) and types to depend on terms (λP). It only has one different typing rule to λP, which is the following:

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \Pi x : A.\, B : s_2} \ (\text{form}_{\lambda C})$$

For an in-depth discussion on this change, see Nederpelt and Geuvers (2014). Having incorporated all dimensions of the λ-cube, this theory is the strongest of all eight, and is equivalent to the implicational fragment of higher-order intuitionistic logic. As such, it (or variations of it) is frequently used in proof-oriented programming languages, that is, programming languages able to prove mathematical propositions.

A slight modification of CoC, called the Calculus of Inductive Constructions (CIC), is the theoretical basis for the Coq Proof Assistant, created around the 1990s, and the Lean 4 programming language, gaining traction over the last few years. Both languages are commonly used for formalizing proofs of mathematical theorems, and can also be used to write programs and verify them via the Curry-Howard correspondence. These languages have been used in many successful industry applications, of which we will discuss two examples.

One notable example is CompCert (Leroy, 2009), a formally verified optimizing C compiler developed using Coq. Created by Xavier Leroy and his team at INRIA, it translates source code written in a large subset of the C programming language into assembly code for several processor architectures, including x86, ARM, and PowerPC. What makes CompCert particularly significant is that it comes with a machine-checked proof of semantic preservation for the entire compilation chain, meaning that the compiler is mathematically proven to generate machine code that behaves exactly as prescribed by the semantics of the source C program. This eliminates the possibility of compiler-introduced bugs, which is unprecedented in traditional compiler development — even widely-used compilers like GCC or LLVM occasionally introduce subtle bugs during optimization phases.

Over the past year, the AWS development team has been using Lean to verify core parts of its authorization policy language called Cedar, as reported by Hietala and Torlak (2024). Cedar allows developers to write authorization policies that define and enforce permissions for their applications, separating access control logic from application code. For instance, a Cedar policy can specify that users can only perform actions on documents they own, or that deleted documents in a “Trash” folder cannot be edited by any user. Cedar’s development follows a verification-guided approach, where each core component is first modeled and verified in Lean before being implemented in Rust. The Lean model serves as a highly readable prototype that can be formally proven to satisfy key properties.

Part II

Data-centric programming

Chapter 5

Exploration in data-centric programming

In the first part of this monograph, we thoroughly discussed the theoretical basis for untyped and typed λ-calculi and showed how they are relevant to formal verification. In this second part, our goal is to explore how a dependently-typed language can be used in data-centric programming. In this chapter, we will focus on a topic in data-centric programming (data frames) and define it.

5.1 Data-centric programming

What we call *data-centric programming* in this monograph is simply the set of programming tasks directly related to dealing with data. That is, a data-centric programming system is not centered around the side effects it produces, but the data it receives, represents, transforms, and outputs.

A classic example of this is a database management system (DBMS). The main goal of these systems is to define, create, maintain, and control access to a database (Connolly and Begg, 2015). These systems are usually used through a data-centric programming language such as SQL. This language, based on relational algebra, has three main purposes: defining schemas of data; inserting, deleting, and updating data in a DBMS; and querying existing data.

Another data-centric programming language is R, used mostly in statistical computing and data science. One of its central building blocks is the **data frame**, a mixed-type tabular data structure. These structures represent data as an indexed set of rows that each contain a tuple of elements, similarly to relational algebra but in a more flexible way, not requiring previously-defined schemas. Data frames also include operations to extend, reduce, and update rows, columns and individual values. Finally, it is possible to query specific parts of a data frame.

Although different, relations in relational algebra and tables in data frames have three main corresponding purposes: **representing**, **transforming**, and **querying** data. We believe that these are the foundational pillars for a data-centric system, and will divide the discussion of our implementation in these three parts. We will focus our analysis of existing systems and our implementation on data frames, which are a lower-hanging fruit than full DBMSs in terms of complexity and existing literature.

5.2 Data frames

Our focus on the study of data frames will be over the *DataFrame* class in the Python library **pandas**, described by McKinney (2010).

Data frames were originally created to represent instances from statistical data sets, which usually arrive in a tabular format. They contain a list of observations (rows) and a name (column) for its fields. Data in this format can easily be encoded as lists of tuples (or lists of lists), although leaving it at that leads to awkward transformations and queries. Because of this, several abstractions over this structure are added.

A data frame in `pandas` looks like the following:

```
>>> data = DataFrame.fromcsv('data', index_col=None)
  date          item    value    volume
0 2009-12-28    GOOG    622.9    1.698e+06
1 2009-12-29    GOOG    619.4    1.425e+06
2 2009-12-30    GOOG    622.7    1.466e+06
3 2009-12-31    GOOG    620      1.22e+06
4 2009-12-28    AAPL    211.6    2.3e+07
5 2009-12-29    AAPL    209.1    1.587e+07
6 2009-12-30    AAPL    211.6    1.47e+07
7 2009-12-31    AAPL    210.7    1.257e+07
```

Each column is labeled by a name (e.g. `date`, `item`, ...) and each row is indexed by a number. The frame can also be reshaped (transformed) like the following example, which groups values by `date` and `item`:

```
>>> df = data.pivot('date', 'item', 'value')
>>> df
              AAPL     GOOG
2009-12-28    211.6    622.9
2009-12-29    209.1    619.4
2009-12-30    211.6    622.7
2009-12-31    210.7    620
```

This transformation creates a new view of the data where rows are indexed by dates and columns represent different items (although our implementation in the next chapter will not be able to change the indices of rows). The original values are preserved, but organized differently. This exemplifies one of the key features of data frames: the ability to reshape data.

Data frames in pandas support various operations that can be categorized into our three pillars:

Data Representation

Data frames represent tabular data through:

(i) **Column labels**: Names that identify each field in the data;

(ii) **Row indices**: Unique identifiers for each observation;

(iii) **Mixed-type data**: Unlike arrays, columns can have different types;

(iv) **Missing values**: Special handling of null or missing data points.

Data Transformation

Common transformations include:

(i) **Reshaping**: Operations like `pivot`, `melt`, and `transpose`;

(ii) **Combining**: Merging or concatenating multiple data frames;

(iii) **Grouping**: Aggregating data based on common values;

(iv) **Filtering**: Removing or selecting specific rows or columns;

(v) **Extending**: Creating new columns based on existing ones.

Data Querying

Data frames can be queried through:

(i) **Label-based access**: Selecting data by column names or row labels;

(ii) **Position-based access**: Selecting data by numerical indices;

(iii) **Boolean indexing**: Filtering data based on logical conditions;

(iv) **Set operations**: Finding unique values or checking membership.

For example, we can query the previous data frame to find all stock prices above 500:

```
>>> data[data['value'] > 500]
  date          item    value    volume
0 2009-12-28    GOOG    622.9    1.698e+06
1 2009-12-29    GOOG    619.4    1.425e+06
2 2009-12-30    GOOG    622.7    1.466e+06
3 2009-12-31    GOOG    620      1.22e+06
```

These operations make data frames a powerful tool for data manipulation and analysis. However, their implementation in existing systems like `pandas` lacks formal verification and type safety. For instance, there's no compile-time guarantee that column names exist or that operations between columns are type-compatible. This motivates our exploration of implementing data frames in a dependently-typed language, where we can provide these guarantees while maintaining the flexibility and expressiveness of existing implementations.

Chapter 6

A dependently-typed data framework

From Chapters 1 through 4 we discussed the theoretical framework behind dependent types, a key feature of the Lean programming language. In Chapter 5, we discussed data-centric programming and specifically data frames in the context of the `pandas` library. Now, we join everything we've discussed so far to implement the core of a data frame library in Lean, leveraging the benefits of dependent types.

Our code is a re-implementation of the `idris-data-frame` library written for Idris 2 by Tejiščák (2020). This library is inspired by and similar to `dplyr` (Wickham *et al.*, 2023), a grammar of data manipulation.

6.1 Data representation

As mentioned earlier, data frames can be seen as lists of lists. Underneath a few layers of abstraction, this is essentially what our code will implement, although we will use some features of dependent types to provide additional checking to certain invariants. At its core, a data frame (DF) is a list of columns, and each column is a vector (list with fixed size) of values of a fixed type.

6.1.1 Named types

First, we must specify the name and type of each column of a data frame. We do this through the `Named` structure, defined in Program 6.1.

The structure `Named` is essentially a pair between a string, which gives a name to a column, and a type α in universe $u - 1$, which itself is of a type in universe u. This makes sense once we consider the discussion on Russell's Paradox from Chapter 1. The same way set theory had to develop to avoid the "set of all sets", Lean's underlying type system also must avoid the type of all types. It does this similarly to Russell's ramified theory of types, creating different levels (called *universes*) of types indexed by a natural number. `Type 0` is the lowest level, and contains the types of simple values; `Type (n + 1)`

Program 6.1 Named type.

```
structure Named (α : Type u) where
  name : String
  type : α

infix:30 " :- " => Named.mk
-- example: "Name" :- String
```

contains all types of universe n. Although universes can frequently be omitted in Lean using an underscore (_), we will use them throughout our code to be more explicit and avoid implicit errors. Our `Named` structure is *universe-polymorphic*, so the universe of its type parameter α can be any natural number.

Lean has an in-depth meta-programming system, allowing (among other things) easily defined extra syntax. A value of type `Named (Type _)` can be created using the implicitly created function `Named.mk` (e.g. `Named.mk "Number", Int`), its angle brackets syntax sugar (e.g. `⟨"Number", Int⟩`), and our extra notation ":-" (e.g. `"Number" :- Int`).

6.1.2 Signatures

Next, we define the **signature** of each data frame. A signature is a list containing each column's name and type, seen in Program 6.2.

Program 6.2 Data frame signature.

```
universe u
abbrev Sig := List (Named (Type u))
```

Notice how `Named` is parametrized by `Type u`. Most data frames store values whose type are of universe 0, such as strings, integers, date times, etc, so it wouldn't really be necessary to have universe polymorphism. However, we will keep this feature as part of our exploration of dependent types.

6.1.3 Column pointers

Next, we have the central piece of verified data frame access in Program 6.3, which Christiansen (2023) calls **column pointers**. These are structures that serve both as *proofs* that a signature contains a given column (a specific named type), and as *data* on how to find it. This is a concrete example on how Lean code can contain proofs for arbitrary propositions and how it can be stored as data.

The `inductive` keyword is how sum types (Section 3.5.4) are defined in Lean. the `InSig` type is essentially a linked list with two possible ways to be created: a constructor `here`, which creates the "end" (last element) of the list, and `there`, which "points" to another list. The difference between this type and actual lists in Lean is that the second constructor

Program 6.3 Column pointers.

```
inductive InSig
  : (name : String)
  → (α : Type u)
  → (sig : Sig)
  → Type (u + 1)
 where
  | here  : InSig name α ((name :- α) :: sig)
  | there : InSig name α sig → InSig name α ((name' :- α') :: sig)
```

doesn't store a concrete value passed as parameter, but stores information as part of its types. The definition of `InSig` is quite tricky, so let us review each of its parts.

First, the type `InSig` receives as parameters a string `name` and a type α that represent which `Named` instance it points to in a signature; then it receives a signature `sig` that will contain `name :- α`; and finally returns a new type, whose universe must be greater than α's to avoid Russell's Paradox.

The `here` constructor receives as implicit parameters `name`, α, and `sig`, which are only named the same as its correspondents in the type parameters to facilitate comprehension. This same constructor could be rewritten to show all implicit parameters as:

```
 | here {name : String} {α : Type u} {sig : List (Named (Type u))}
   : InSig name α ((name :- α) :: sig)
```

It then constructs an `InSig` instance by providing the string `name`, the type α, and a *new* signature that *must* contain the value `name :- α` as its first element. This means that an `InSig` can *only* be created if its third type parameter (a signature) contains as its first element the exact `Named` instance created by the first and second type parameter.[1] This is the most complicated part of this trick, and takes a while to internalize. Chapter 8 of Functional Programming in Lean (Christiansen, 2023) goes more in depth.

The base case of the `InSig` list is generated by the `here` constructor, so the only way to create instances of this type other than `here` is to point to an existing `InSig`. This also means that any `InSig name α sig` **must** contain `name :- α` somewhere inside `sig`. The second constructor, `there`, shows how to find it. It receives as implicit parameters the same values as before, but also the additional parameters `name'` and `α'`. Their values don't really matter, as they're only used to tell the language that the result of the constructor will increase the size of the signature it points to. It could also be rewritten as:

```
 | there : InSig name α sig → InSig name α (_ :: sig)
```

Considering this, the behavior of this constructor is the following: it receives another existing `InSig` value and a corresponding signature proven to contain the desired `Named` value, and points to it, associated to another larger super set signature.

[1] In Lean, lists are constructed by the *cons* operator, written `x :: xs`. This code creates a new linked list that has `x` as its first value and the other list `xs` as the rest.

Values of the type `InSig` look like the following:

- `InSig.here`
- `InSig.there .here`
- `InSig.there (.there (.there .here))`
- `InSig.there (.there (.there (.there (.there .here))))`

These values are isomorphic to natural numbers, and the only information they actually contain is the index of a specific named type in a given signature. The proofs they represent only exist in the typing information, which can simply be erased in compile time. This is the duality between proofs and data: in Lean, a type can purely exist as verification of an invariant, a value can exist just as data, or they can both coexist in the same structure.

One interesting aspect of this encoding is that a signature will be contained in full in the types of the first constructor of a `there` chain, then the next value will have in its types the same signature but with its head removed, then the next will remove another one, and so on, until a value constructed by `here` will have in its types a signature with its head as the desired `Named` instance. So, while deconstructing an `InSig`, a programmer is also deconstructing a `Sig`. See the following examples of `InSig` instances alongside their types:

```
example
  : InSig "name" String ["name" :- String]
  := InSig.here

example
  : InSig "age" Int [ "name" :- String
                    , "height" :- Float
                    , "age" :- Int]
  := .there (.there .here)
```

Although defining column pointers directly works, it is certainly undesirable to write down the full `.there` chain every time they are needed. Lean has a meta-programming system called *tactics*, which allows us to write code that writes other code automatically. Recall how the existence of an `InSig` instance is already a proof of a signature containing a given named type. Then, if we somehow create any instance of this type, it's already enough to get a desired column. The `constructor` tactic tries to construct an inductive type from the first available constructor. The `repeat` tactic tries to execute a specific tactic, and if it fails, tries it again with its next possible state, until it either stops by succeeding or by reaching a pre-determined limit. If we join these two tactics, we can tell Lean to try to construct an `InSig` by just repeating the constructors until we get a valid instance. See the following example:

```
example : InSig "age" Int ["name" :- String, "age" :- Int]
  := by repeat constructor
```

6.1.4 Columns

Next, we define lists of columns in Program 6.4, which is the main inner structure of a data frame.

Program 6.4 List of columns, each containing a list of rows.

```
inductive Columns : Nat → Sig → Type (u + 1) where
  | nil  : Columns n []
  | cons
      : {α : Type u}
      → Vect α n
      → Columns n sig
      → Columns n ((name :- α) :: sig)
```

This structure receives as type parameters a natural number, which represents the (fixed) amount of observations (rows) in each column; and a signature, which specifies the names and types of each column. Similarly to how we defined `InSig`, this type is at its core a linked list, but each node stores a row.

The `nil` constructor is the base case — it doesn't store a row, and its signature is an empty list (since it doesn't contain any columns). It, however, still stores the number n of rows in each column.

The `cons` constructor receives an implicit type α, which will be the type stored by the node's rows; a vector of size `n` containing a value of α for each row; and the next node of the linked list, which will be another value of the `Columns` type. The resulting type, `Columns n ((name :- α) :: sig)`, builds `(name :- α)` into the columns' signature, so the existence of a `Named` can be verified when deconstructing `Columns`. See the following example of an instance of `Columns`:

```
def exampleColumns : Columns 2 ["name" :- String, "age" :- Int]
  := Columns.cons (Vect.cons "Alice" (Vect.cons "Bob" Vect.nil))
    (Columns.cons (Vect.cons 25      (Vect.cons 24 Vect.nil))
     Columns.nil)
```

Circling back to column pointers, we can now show how to use an `InSig` to extract the observations of a single column given its pointer. See the extraction function in Program 6.5. Note that variables whose names start with an underscore are not used.

Program 6.5 Column extracting function.

```
def Columns.extract
  : Columns rowCount sig
  → InSig name α sig
  → Vect α rowCount
  | .cons xs _cols, .here => xs
  | .cons _xs cols, .there pf => extract cols pf
```

This function works by pattern matching over a `Columns` (`.cons _ _`) and an `InSig` (`.here | .there`). You might notice that the pattern match does not contain a case for the `Columns.nil` constructor, and that is a consequence of how we construct column pointers. Since both `InSig` constructors have non-empty signatures (that is, `Sig` is constructed with the cons operator (`::`), if a `Columns` and an `InSig` use the same signature, then the `Columnns` instance cannot be `.nil`. For a similar reason, matching a `.here` constructor guarantees that `xs` is of the desired type. The feature that allows both instances is called dependent pattern matching, and is one of the nicest features brought by dependent types.

6.1.5 Data frames

Finally, we can define the data frame (DF) structure in Program 6.6.

Program 6.6 Data frame structure.

```
structure DF (sig : Sig) where
  {rowCount : Nat}
  columns : Columns rowCount sig
```

A `DF` of a fixed signature `sig` contains a list of columns (of type `Columns`); and a natural number for the amount of rows it contains, which is implicitly created from `columns`. For example:

```
example : DF ["name" :- String, "age" :- Int]
  := ⟨Columns.cons (Vect.cons "Alice" (Vect.cons "Bob" Vect.nil))
     (Columns.cons (Vect.cons 25      (Vect.cons 24 Vect.nil))
      Columns.nil)⟩
```

The above syntax for creating data frames works, although it is not very convenient. Leveraging Unicode characters, we add an operator to create rows (¦) and another to create columns (‖):

```
infixr:55 " ¦ " => Vect.cons
notation:55 lhs:55 " ¦ " => Vect.cons lhs Vect.nil
example := 1 ¦ 2 ¦ 3 ¦
notation:50 " ‖ " lhs:51 rhs:50  => Columns.cons lhs rhs
notation:50 " ‖ " lhs:51  => Columns.cons lhs Columns.nil
example : Columns 2 ["i1" :- Int, "i2" :- Nat]
         := ‖ 1 ¦ 2 ¦ ‖ 3 ¦ 4 ¦
```

Finally, we can show a full example for a data frame in Program 6.7.

6.2 Transformations and queries

6.2.1 Expressions

In our library, transformations and queries are part of an embedded domain-specific language (DSL). The expressions of this DSL compose with each other to form complex

Program 6.7 Example data frame in our library.

```
def exampleDF : DF
  ([ "name"    :- String
   , "age"     :- Int
   , "country" :- Option String
   ]) := ⟨
   ‖   "Artur" ¦      "Bob" ¦  "Claire" ¦  "Blorg" ¦ "Zorg" ¦
   ‖        32 ¦         54 ¦        41 ¦      101 ¦     99 ¦
   ‖ some "BR" ¦ some "US" ¦ some "FR" ¦    none  ¦   none ¦⟩
```

operations. Program 6.8 shows the `Expr` type, which defines the main (but not all) operations that can be done over a data frame. Its code is quite involved, so we will go through each part step-by-step.

Program 6.8 Domain-specific language (DSL) expression type

```
1   inductive Quantity where | one | many
2
3   inductive Expr : Quantity → Sig → Type → Type (max v u + 1) where
4     | L : α → Expr q sig α
5     | V : (name : String) → InSig name α sig → Expr .many sig α
6     | Count : [Add α] → [Mul α] → [FromInt α] → Expr q sig α
7     | Map : (α → β) → Expr q sig α → Expr q sig β
8     | BinOp
9       : (α → β → γ) → Expr q sig α → Expr q sig β → Expr q sig γ
10    | Aggregate : (List α → β) → Expr .many sig α → Expr .one sig β
```

First, let's see the type parameters required by `Expr`. It requires a value of type `Quantity`, which as defined in line 1, can only be `one` or `many`. This allows expressions to specify if they will return or operate over a single value or multiple at a time, for instance, an entire column. Some operations only make sense over a single value, and some only over multiple; this is specified in each constructor. `Expr` also receives a signature and a type; the first is the signature of data frames the expression will deal with, and the second is the type returned when evaluating the DSL.

Now, let us go over each constructor:

- Constructor **L**: introduces a value into an expression. For example, `Expr.L (some "x")` introduces the optional string "x".
- Constructor **V**: references a column in a signature. Note that it requires both the column name and a column pointer, the latter serving as a **proof** that the column does indeed exist in a given signature. Its return type is marked by `.many`, so evaluating an expression of `V` will return multiple values.
- Constructor **Count**: counts the number of rows in a data frame.
- Constructor **Map**: maps the value(s) referenced by an expression from type α to

type β.

- Constructor **BinOp**: executes a binary operation between the value of two expressions.
- Constructor **Aggregate**: aggregates many values into one given a function of a list of values to a single output. Note its argument is marked by `.many` and result by `.one`, so it must receive multiple values and will return only one.

To improve the expressivity of the language we add functions `val` and `col`, which are simply aliases of `Expr.L` and `Expr.V`.

```
def val : α → Expr q sig α := Expr.L
def col : (name : String) → InSig name α sig → Expr .many sig α :=
    Expr.V
```

This DSL will make extensive use of *type classes*, which will compose over the values of each expression to build more complex results. Another way to accomplish this task is what Chapter 8.2 of Christiansen (2023) calls the *universe design pattern*, which uses Tarski-style universes to encode types. This is more simple than it seems, and we will get back to it in Section 6.3.

We add a few generally straight-forward instances for expressions, which are enough for basic operations `add`, `mul`, `sub`, `div`, `neg`:

```
instance [Add α] : Add (Expr q sig α) where
  add := Expr.BinOp Add.add
instance [Mul α] : Mul (Expr q sig α) where
  mul := Expr.BinOp Mul.mul
instance [Sub α] : Sub (Expr q sig α) where
  sub := Expr.BinOp Sub.sub
instance [Div α] : Div (Expr q sig α) where
  div := Expr.BinOp Div.div
instance [FromInt α] : FromInt (Expr q sig α) where
  fromInt n := pure (FromInt.fromInt n)
instance : Functor (Expr q sig) where
  map := Expr.Map
instance [Neg α] : Neg (Expr q sig α) where
  neg := Functor.map Neg.neg
```

A more complex binary operation can be formed, like the `over` function:

```
def over [Ord α] : Expr q sig α → Expr q sig α → Expr q sig Bool :=
  Expr.BinOp (fun x y => Ord.compare x y == .gt)
```

6.2.2 Expression evaluation

Finally, we can implement evaluation for expressions of the DSL in Program 6.9. This program uses a few functions we haven't defined in this chapter; most have equivalent names in functional programming literature, and the full code will be presented in Appendix A.

Program 6.9 DSL expression evaluation.

```
abbrev EvalTy : Quantity → Nat → Type → Type
  | .one,  _, α => α
  | .many, n, α => Vect α n

def Expr.eval (df : DF sig) : {q : Quantity} → Expr q sig α →
    EvalTy q (df.rowCount) α
  | .one,  L x => x
  | .many, L x => Vect.replicate x df.rowCount
  | .many, V name loc => df.get name loc
  | .one,  Count => FromInt.fromInt df.rowCount
  | .many, Count => Vect.replicate (FromInt.fromInt df.rowCount)
    df.rowCount
  | .one,  Map f xs => f (xs.eval df)
  | .many, Map f xs => Functor.map f (xs.eval df)
  | .one,  BinOp f xs ys => f (xs.eval df) (ys.eval df)
  | .many, BinOp f xs ys => Vect.zipWith f (xs.eval df) (ys.eval df)
  | .one,  Aggregate f e => f (ToList.toList (e.eval df))
```

The return type of `Eval.eval` is the type `EvalTy`, which contains a single value if its quantity is `.one` or *n* if it is `.many`. Like we did with the definition of `Expr`, let's go over each constructor one by one:

- **L**: Either returns a single value `x` or replicates it `n` times;
- **V**: Returns every observation from a column;
- **Count**: Either returns a single value with the row count or replicates it `n` times;
- **Map**: Either maps a value resulting from evaluation an expression or multiple values;
- **BinOp**: Runs a binary operation between the evaluation of two other expressions, or if they have multiple results, between all values of two expressions;
- **Aggregate**: Aggregates multiple values resulting from an evaluation into a single one.

To conclude this section, we present an example of the DSL in action. It uses a few functions not defined throughout the text, which are available in Appendix A.

```
def alienAges := exampleDF
  |>.where (Option.isNone <$> col "country" (by repeat constructor))
  |>.select (
    · ("name"    :- col "name" (by repeat constructor))
    · ("age"     :- col "age"  (by repeat constructor))
    · ("century" :- over (col "age"  (by repeat constructor))
                         (FromInt.fromInt 100)))
  |>.get "century" (by repeat constructor)
```

6.3 Further developments

There are a few extensions we could implement to the code presented so far to make it more usable and interesting as a standalone framework for data frames. We now discuss two possible features to improve the usability of our library.

Column type inference

The types representable in this system are all the possible instantiations of `Named`, and the operations over these types are all those that can be expressed with the `Expr` constructors, such as `BinOp`. One way to restrict these expressions is through the *universe design pattern* (Christiansen, 2023), also called *Tarski-style universes*. They are a way to encode types in data structures, associated to some function that transforms them back to the type level. They can be useful to write code that pattern matches over types, as that cannot be done purely at the type level in Lean. An application of this is generating types automatically for a given untyped data frame, effectively doing type inference for columns. This idea is inspired by type providers in F, presented in Syme *et al.* (2013). These type providers generate types at compile time for structures given a schema or an example instantiation, removing the need to explicitly specify the shape of a data structure in code.

Transpilation

A benefit of writing verified queries and transformations for data is having the assurance that they will work for a given specification. However, most pipelines dealing with data are already implemented in different systems such as *pandas* and *R*, and it would not be desirable to re-implement them in Lean. To do this, teams would have to be trained in a new and uncommon programming language, taking up time and resources. A way to deal with this problem is to have a well-defined DSL in Lean that wouldn't need knowledge of the language's inner workings to write code in, and the expressions of this DSL could be transpiled to more commonly utilized systems. This way, users could benefit from the confidence and verification brought by our system in Lean, while not sacrificing the environments they already have in production.

Chapter 7

Conclusion

In this monograph, we explored the foundations of type theory and its applications to data-centric programming. The work was divided into two main parts: a theoretical study of types, from their origins to dependent types; and a practical implementation of a data frame library in the dependently-typed programming language Lean 4.

The theoretical exploration began with Russell's Paradox and the development of type theory as a solution to inconsistencies in set theory. We then thoroughly examined the untyped lambda calculus, discussing its syntax, reduction rules, and ways to encode data within it. This led to the simply-typed lambda calculus, where we explored typing rules, different typing styles, and the Curry-Howard correspondence. Finally, we discussed dependent types through the lens of the lambda cube and the calculus of constructions, showing how they enable the expression of more complex properties and proofs within the type system itself.

In the practical portion, we implemented a data frame library that leverages dependent types to provide stronger guarantees about data manipulation operations. Our re-implementation of the library by Tejiščák (2020) focused on three main aspects of data-centric programming: representation, transformation, and querying. The core of our implementation uses column pointers - an application of dependent types that serves both as proof of column existence and as data for column access. We developed a domain-specific language for expressing transformations and queries, which maintains type safety while allowing flexible operations on data frames.

Although parts of the code can be tricky, specially to those not versed in dependently-typed programming languages, there are many advantages of this approach. The practical benefits of this approach include compile-time verification of column existence and type compatibility for operations, which helps prevent common runtime errors found in traditional data frame implementations. Our work demonstrates how dependent types can be effectively used to add formal verification to data manipulation while maintaining the flexibility and expressiveness expected from data frame libraries.

Future work could extend this implementation in several ways. Two promising directions are:

(i) Adding column type inference through Tarski-style universes, which would reduce the need for explicit type annotations while maintaining type safety.

(ii) Implementing transpilation of our DSL to other data frame systems, allowing the benefits of verification while keeping compatibility with existing data processing pipelines.

This work contributes to both the theoretical understanding of type systems and their practical application in data-centric programming. It shows how advanced type system features like dependent types can be used to build safer and more reliable data processing tools, while maintaining the expressiveness needed for real-world applications.

Appendix A

Source code

```lean
-- All code was written for Lean version 4.13.0
-- This library is a reimplementation of https://github.com/ziman/idris-data-frame

namespace DF

structure Named (α : Type u) where
  name : String
  type : α

infix:30 " :- " => Named.mk
-- example: "Name" :- String

def Named.mapItemType (f : α → β) : Named α → Named β
  | (nam :- α) => nam :- f α

universe u
abbrev Sig := List (Named (Type u))

inductive SigF : (Type u → Type v) → Sig → Type ((max u v) + 1) where
  | nil  : SigF f []
  | cons : (e : Named (p a)) → SigF p sig → SigF p ((e.name :- a) :: sig)

notation:50 " · " lhs:51 rhs:50  => SigF.cons lhs rhs
notation:50 " · " lhs:51  => SigF.cons lhs SigF.nil

inductive InSig : (name : String) → (α : Type u) → (sig : Sig) → Type (u + 1) where
  | here  : InSig name α ((name :- α) :: sig)
  | there : InSig name α sig → InSig name α (_ :: sig)

example
```

```
  : InSig "name" String ["name" :- String]
  := InSig.here

example
  : InSig "age" Int [ "name"   :- String
                    , "height" :- Float
                    , "age"    :- Int]
  := .there (.there .here)

abbrev MapTy : (Type u → Type v) → List (Named (Type u)) → List
    (Named (Type v)) :=
  List.map ∘ Named.mapItemType

def Sig.sigMapId : (sig : Sig) → (MapTy (fun x => x) sig) = sig
  | [] => rfl
  | ((name :- value) :: sig) =>
    congrArg ((name :- value) :: ·) (Sig.sigMapId sig)

def mapId : {xs : List α} → xs.map (fun x => x) = xs
  | [] => rfl
  | x :: _ => congrArg (x :: ·) mapId

def Sig.insert (name : String) (α : Type u) : Sig → Sig
  | [] => [name :- α]
  | (name' :- α') :: sig =>
  if name == name'
    then (name  :- α)  :: sig
    else (name' :- α') :: insert name α sig

def Sig.overrideWith : Sig → Sig → Sig
  | lhs, [] => lhs
  | lhs, (name :- α) :: rhs => overrideWith (insert name α lhs) rhs

inductive Row : Sig → Type (u + 1) where
  | nil  : Row []
  | cons : α → Row sig → Row ((name :- α) :: sig)

inductive Vect (α : Type u) : Nat → Type u where
  | nil  : Vect α 0
  | cons : α → Vect α n → Vect α (n + 1)
deriving Repr, BEq

def exampleVect := Vect.cons 4 $ .cons 3 $ .cons 1 $ .cons 2 $ .cons
    0 $ .nil

def Vect.snoc : Vect α n → α → Vect α (n + 1)
  | .nil, v => .cons v .nil
  | .cons x xs, v => .cons x (snoc xs v)

```

```
-- Warning: O(²n)
def Vect.reverse : Vect α n → Vect α n
  | .nil => .nil
  | .cons x xs => .snoc (reverse xs) x

def Vect.append' {n m : Nat} : Vect α n → Vect α m → Vect α (m + n)
  | .nil, ys => ys
  | .cons x xs, ys => .cons x (xs.append' ys)

def Vect.append {n m : Nat} : Vect α n → Vect α m → Vect α (n + m)
    :=
  Nat.add_comm n m ▷ Vect.append'

instance : HAppend (Vect α n) (Vect α m) (Vect α (n + m)) where
  hAppend := Vect.append

def Vect.empty : Vect α 0 := .nil

def Vect.singleton (x : α) : Vect α 1 := Vect.cons x .nil

def Vect.splitAt' : (n : Nat) → Vect α (m + n) → Vect α n × Vect α m
  | 0, rest => (.nil, rest)
  | n' + 1, .cons x xs =>
    let (lft, rgt) := xs.splitAt' n'
    (.cons x lft, rgt)

def Vect.splitAt (n : Nat) : Vect α (n + m) → Vect α n × Vect α m :=
  Nat.add_comm n m ▷ Vect.splitAt' n

def Vect.toList : Vect α n → List α
  | .nil => []
  | .cons x xs => x :: toList xs

def Vect.fromList : (xs : List α) → Vect α xs.length
  | [] => .nil
  | x :: xs => .cons x (fromList xs)

def Vect.insertSorted [Ord α] (v : α) : Vect α n → Vect α (n + 1)
  | .nil => .cons v .nil
  | .cons x xs =>
    match compare x v with
    | .lt => .cons x (xs.insertSorted v)
    | .eq
    | .gt => .cons v (.cons x xs)

-- Warning: O(²n)
def Vect.sort [Ord α] : Vect α n → Vect α n
  | .nil => .nil
  | .cons x xs => xs.sort |>.insertSorted x
```

```

def Vect.zip : Vect α n → Vect β n → Vect (α × β) n
  | .nil, .nil => .nil
  | .cons x xs, .cons y ys => .cons (x, y) (xs.zip ys)

def Vect.map (f : α → β) : Vect α n → Vect β n
  | .nil => .nil
  | .cons x xs => .cons (f x) (xs.map f)

instance : Functor (fun α => Vect α n) where
  map := Vect.map

def Vect.permute [Ord α] (perm : Vect α n) (xs : Vect β n) : Vect β
    n :=
  let pairs := Vect.zip perm xs
  let _ordInstance : Ord (α × β) := { compare := fun p1 p2 =>
    compare p1.fst p2.fst }
  let sorted := pairs.sort

  sorted.map Prod.snd

def Vect.replicate (x : α) : (n : Nat) → Vect α n
  | 0      => .nil
  | n' + 1 => .cons x (replicate x n')

def Vect.zipWith (f : α → β → γ) : Vect α n → Vect β n → Vect γ n
  | .nil, .nil => .nil
  | .cons x xs, .cons y ys => .cons (f x y) (zipWith f xs ys)

theorem min_succ : (min n m) + 1 = min (n + 1) (m + 1) := by
  cases Nat.le_total n m with
  | inl h => rw [Nat.min_eq_left h, Nat.min_eq_left (Nat.succ_le_succ
    h)]
  | inr h => rw [Nat.min_eq_right h, Nat.min_eq_right
    (Nat.succ_le_succ h)]

def Vect.take' : (m : Nat) → Vect α n → Vect α (min m n)
  | 0, _ => by simp!; exact .nil
  | n' + 1, .nil => .nil
  | n' + 1, .cons x xs =>
    min_succ ▷ .cons x (take' n' xs)

class ToList (f : Type u → Type v) (α : Type u) where
  toList : f α → List α

instance : ToList (fun α => Vect α n) α where
  toList := Vect.toList

inductive Columns : Nat → Sig → Type (u + 1) where
```

```
  | nil  : Columns n []
  | cons : {α : Type u} → Vect α n → Columns n sig → Columns n
    ((name :- α) :: sig)

def exampleColumns : Columns 2 ["name" :- String, "age" :- Int]
  := Columns.cons (Vect.cons "Alice" (Vect.cons "Bob" Vect.nil))
    (Columns.cons (Vect.cons 25      (Vect.cons 24 Vect.nil))
     Columns.nil)

def Columns.toList : {sig : Sig} → Columns n sig → List (Σ α :
    Type, Vect α n)
  | _, .nil => []
  | (_ :- α) :: _, .cons col cols => ⟨α, col⟩ :: cols.toList

def Columns.append {sig : Sig} : Columns m sig → Columns n sig →
    Columns (m + n) sig
  | .nil, .nil => .nil
  | .cons xs cs, .cons xs' cs' => .cons (xs ++ xs') (Columns.append
    cs cs')

def Columns.bindCols : Columns n sig → Columns n sig' → Columns n
    (sig ++ sig')
  | .nil, ys => ys
  | .cons x xs, ys => .cons x (bindCols xs ys)

def Columns.reverse : Columns n sig → Columns n sig
  | .nil => .nil
  | .cons xs cs => .cons xs.reverse cs.reverse

def Columns.empty : {sig : Sig} → Columns 0 sig
  | [] => .nil
  | _ :: _ => .cons Vect.empty empty

def Columns.deepMap
  {sig : Sig}
  (p : Type _ → Type _)
  (f : {α : Type _} → Vect α n → Vect (p α) m)
  : Columns n sig
  → Columns m (MapTy p sig)
  | .nil => .nil
  | .cons xs cs => .cons (f xs) (deepMap p f cs)

def Columns.map
  {sig : Sig}
  (f : {α : Type _} → Vect α n → Vect α m)
  (cols : Columns n sig)
  : Columns m sig :=
  Sig.sigMapId sig ▷ Columns.deepMap (fun x => x) f cols

```

```
def Vect.trueCount : Vect Bool n → Nat
  | .nil => 0
  | .cons true xs => trueCount xs + 1
  | .cons false xs => trueCount xs

def Vect.where_ : (mask : Vect Bool n) → Vect α n → Vect α
    (Vect.trueCount mask)
  | .nil, .nil => .nil
  | .cons true mask, .cons x xs => .cons x (where_ mask xs)
  | .cons false mask, .cons _ xs => (where_ mask xs)

def Columns.where_
  {sig : Sig}
  (mask : Vect Bool n)
  : Columns n sig
  → Columns (Vect.trueCount mask) sig
  | .nil => .nil
  | .cons xs cs => .cons (Vect.where_ mask xs) (where_ mask cs)

def Columns.rowCons : Row sig → Columns n sig → Columns (n + 1) sig
  | .nil, .nil => .nil
  | .cons x xs, .cons col cols => .cons (.cons x col) (rowCons xs
    cols)

def Columns.rowUncons : Columns (n + 1) sig → Row sig × Columns n sig
  | .nil => (.nil, .nil)
  | .cons (.cons x xs) cols =>
    let (firstRow, rest) := rowUncons cols
    (.cons x firstRow, .cons xs rest)

def Columns.takeRows {sig : Sig} (k : Nat) : Columns (k + n) sig →
    Columns k sig × Columns n sig
  | .nil => (.nil, .nil)
  | .cons col cols =>
    let (gcol, rcol) := Vect.splitAt k col
    let (gcols, rcols) := takeRows k cols
    (.cons gcol gcols, .cons rcol rcols)

def Columns.toRows : {n : Nat} → Columns n sig → List (Row sig)
  | 0, _ => []
  | _n' + 1, cols =>
    let (row, rest) := cols.rowUncons
    row :: toRows rest

def Columns.singleton : Row sig → Columns 1 sig
  | .nil => .nil
  | .cons x xs => .cons (Vect.singleton x) (singleton xs)

def Columns.extract
```

```
  : Columns rowCount sig
  → InSig name α sig
  → Vect α rowCount
  | .cons xs _cols, .here => xs
  | .cons _xs cols, .there pf => extract cols pf

example : InSig "age" Int ["name" :- String, "age" :- Int]
  := by repeat constructor

def Columns.order [Ord α]
  (f : {β : Type _} → Vect β n → Vect β n)
  (perm : Vect α n)
  : Columns n sig → Columns n sig
  | .nil => .nil
  | .cons col cols => .cons (f (Vect.permute perm col))
    (Columns.order f perm cols)

-- Rewriting magic required to work on Lean 4.13.0
def Columns.insert (name : String) (xs : Vect α n) : {sig : Sig} →
    Columns n sig → Columns n (Sig.insert name α sig)
  | [], .nil => .cons xs .nil
  | (name' :- α') :: sig', .cons xs' xss' => by
    simp!
    if h : name = name' then
      rw [h]
      simp
      exact (.cons xs xss')
    else
      rw [if_neg h]
      exact .cons xs' (insert name xs xss')

def Columns.overrideWith : {sig' : Sig} → Columns n sig → Columns n
    sig' → Columns n (sig.overrideWith sig')
  | [], xss, .nil => xss
  | (name' :- α') :: sig', xss, (.cons xs' xss') => by
    simp!
    exact overrideWith (Columns.insert name' xs' xss) xss'

structure DF (sig : Sig) where
  {rowCount : Nat}
  columns : Columns rowCount sig

example : DF ["name" :- String, "age" :- Int]
  := ⟨Columns.cons (Vect.cons "Alice" (Vect.cons "Bob" Vect.nil))
     (Columns.cons (Vect.cons 25      (Vect.cons 24 Vect.nil))
      Columns.nil)⟩

def DF.get (df : DF sig) (name : String) (loc : InSig name α sig) :
    Vect α (df.rowCount) :=
  df.columns.extract loc

def DF.fromRow (row: Row sig) : DF sig :=
  ⟨Columns.singleton row⟩

infixr:55 " ¦ " => Vect.cons
notation:55 lhs:55 " ¦ " => Vect.cons lhs Vect.nil

notation:50 " ‖ " lhs:51 rhs:50  => Columns.cons lhs rhs
notation:50 " ‖ " lhs:51  => Columns.cons lhs Columns.nil

example := 1 ¦ 2 ¦ 3 ¦
example : Columns 2 ["i1" :- Int, "i2" :- Nat]
        := ‖ 1 ¦ 2 ¦ ‖ 3 ¦ 4 ¦

#eval ((1 ¦ 2 ¦ 3 ¦) : Vect Nat 3)
#reduce ‖ 1 ¦ 2 ¦ ‖ 3 ¦ 4 ¦

def exampleDF : DF
  ([ "name"    :- String
   , "age"     :- Int
   , "country" :- Option String
   ]) :=
  ⟨‖   "Artur" ¦      "Bob" ¦  "Claire" ¦  "Blorg" ¦ "Zorg" ¦
   ‖        32 ¦         54 ¦        41 ¦      101 ¦     99 ¦
   ‖ some "BR" ¦ some "US" ¦ some "FR" ¦     none  ¦   none ¦⟩

class FromInt (α : Type u) where
  fromInt : Int → α

instance : FromInt Int   where fromInt := id
instance : FromInt Float where fromInt := Float.ofInt

inductive Quantity where | one | many

inductive Expr : Quantity → Sig → Type → Type (max v u + 1) where
  | L : α → Expr q sig α
  | V : (name : String) → InSig name α sig → Expr .many sig α
  | Count : [Add α] → [Mul α] → [FromInt α] → Expr q sig α
  | Map : (α → β) → Expr q sig α → Expr q sig β
  | BinOp : (α → β → γ) → Expr q sig α → Expr q sig β → Expr q
    sig γ
  | Aggregate : (List α → β) → Expr .many sig α → Expr .one sig β

def val : α → Expr q sig α := Expr.L
def col : (name : String) → InSig name α sig → Expr .many sig α :=
    Expr.V

instance : Functor (Expr q sig) where
```

```
  map := Expr.Map

instance : Applicative (Expr q sig) where
  pure := Expr.L
  seq f x := Expr.BinOp id f (x ())

instance [Add α] : Add (Expr q sig α) where
  add := Expr.BinOp Add.add
instance [Mul α] : Mul (Expr q sig α) where
  mul := Expr.BinOp Mul.mul
instance [Sub α] : Sub (Expr q sig α) where
  sub := Expr.BinOp Sub.sub
instance [Div α] : Div (Expr q sig α) where
  div := Expr.BinOp Div.div
instance [FromInt α] : FromInt (Expr q sig α) where
  fromInt n := pure (FromInt.fromInt n)

instance [Neg α] : Neg (Expr q sig α) where
  neg := Functor.map Neg.neg

def Expr.eq [BEq α] : Expr q sig α → Expr q sig α → Expr q sig Bool
  := Expr.BinOp BEq.beq

notation:50 lhs:50 " E== " rhs:50  => Expr.eq lhs rhs

def Expr.and : Expr q sig Bool → Expr q sig Bool → Expr q sig Bool
  := Expr.BinOp Bool.and

notation:50 lhs:50 " E&& " rhs:50  => Expr.and lhs rhs

def over [Ord α] : Expr q sig α → Expr q sig α → Expr q sig Bool :=
  Expr.BinOp (fun x y => Ord.compare x y == .gt)

-- ...

abbrev EvalTy : Quantity → Nat → Type → Type
  | .one,  _, α => α
  | .many, n, α => Vect α n

def Expr.eval (df : DF sig) : {q : Quantity} → Expr q sig α →
    EvalTy q (df.rowCount) α
  | .one,  L x => x
  | .many, L x => Vect.replicate x df.rowCount
  | .many, V name loc => df.get name loc
  | .one,  Count => FromInt.fromInt df.rowCount
  | .many, Count => Vect.replicate (FromInt.fromInt df.rowCount)
    df.rowCount
  | .one,  Map f xs => f (xs.eval df)
  | .many, Map f xs => Functor.map f (xs.eval df)
  | .one,  BinOp f xs ys => f (xs.eval df) (ys.eval df)
  | .many, BinOp f xs ys => Vect.zipWith f (xs.eval df) (ys.eval df)
  | .one,  Aggregate f e => f (ToList.toList (e.eval df))

def DF.where (expr : Expr .many sig Bool) (df : DF sig) : DF sig :=
  DF.mk <| df.columns.where_ (expr.eval df)

def head (n : Nat) (df : DF sig) : DF sig :=
  DF.mk <| df.columns.map (Vect.take' n)

set_option linter.unusedVariables false
def DF.uncons : (df : DF sig) → Option (Row sig × DF sig)
  | @DF.mk _ 0         _ => .none
  | @DF.mk _ (_ + 1) cols =>
    let (row, rest) := cols.rowUncons
    some (row, ⟨rest⟩)
set_option linter.unusedVariables true

def DF.selectCols (df : DF sig) : SigF (Expr .many sig) newSig →
    Columns df.rowCount newSig
  | .nil => .nil
  | .cons e es' => .cons (e.type.eval df) (selectCols df es')

def DF.select (df : DF sig) (es : SigF (Expr .many sig) newSig) : DF
    newSig :=
  DF.mk (df.selectCols es)

def DF.modify (df : DF sig) (es : SigF (Expr .many sig) addSig) : DF
    (sig.overrideWith addSig) :=
  DF.mk (df.columns.overrideWith (df.selectCols es))

inductive OrderBy : Sig → Type _ where
  | asc : [Ord α] → Expr .many sig α → OrderBy sig
  | dsc : [Ord α] → Expr .many sig α → OrderBy sig

def DF.orderStep (df : DF sig) : OrderBy sig → DF sig
  | OrderBy.asc e => DF.mk (Columns.order id (e.eval df) df.columns)
  | OrderBy.dsc e => DF.mk (Columns.order Vect.reverse (e.eval df)
    df.columns)

def DF.orderBy (df : DF sig) (xs : List (OrderBy sig)) : DF sig :=
  List.foldl DF.orderStep df xs

def DF.toList (df : DF sig) : List (Σ α : Type, Vect α df.rowCount)
    :=
  df.columns.toList

def people : DF
  ([ "name"   :- String
   , "age"    :- Int
   , "gender" :- Option String
   , "pet"    :- Option String
   ]) := ⟨
   ‖       "Joe" ¦   "Anne" ¦     "Lisa" ¦      "Bob" ¦      "Zorg" ¦
   ‖          11 ¦       22 ¦         22 ¦         33 ¦          22 ¦
   ‖    some "M" ¦ some "F" ¦   some "F" ¦   some "M" ¦        none ¦
   ‖ some "fish" ¦     none ¦ some "dog" ¦ some "cat" ¦ some "mech" ¦⟩

-- ["Joe", "Anne", "Lisa", "Bob", "Zorg"]
#eval people.get "name" (by repeat constructor) |>.toList

-- true ¦ false ¦ false ¦ true ¦ false ¦
def modified :=
  people
    |>.modify (
      · ("male_with_pet" :-
              ((col "gender" (by repeat constructor)) E== val (some
    "M"))
          E&& (Option.isSome <$> col "pet" (by repeat constructor))))
    |>.get "male_with_pet" (by repeat constructor)

def alienAges := exampleDF
  |>.where (Option.isNone <$> col "country" (by repeat constructor))
  |>.select (
    · ("name"    :- col "name" (by repeat constructor))
    · ("age"     :- col "age"  (by repeat constructor))
    · ("century" :- over (col "age"  (by repeat constructor))
                         (FromInt.fromInt 100)))
  |>.orderBy [.asc (col "age" (by repeat constructor))]
  |>.get "century" (by repeat constructor)

#eval
  people
    |>.where (Option.isSome <$> col "pet" (by repeat constructor))
    |>.select (
      · ("name"   :- col "name" (by repeat constructor))
      · ("age"    :- col "age"  (by repeat constructor))
      · ("plus10" :- Add.add (col "age"  (by repeat constructor))
    (FromInt.fromInt 10))
      · ("-1*2"   :- Mul.mul
                      (Sub.sub (col "age"  (by repeat constructor))
    (FromInt.fromInt 1))
                      (FromInt.fromInt 2)))
    |>.orderBy
      [.asc (col "age" (by repeat constructor))]
    |>.get "-1*2" (by repeat constructor)
```

References

[Barendregt 1974] Henk Barendregt. “Pairing without conventional restraints”. *Zeitschrift fur mathematische Logik und Grundlagen der Mathematik* 20.19-22 (1974), pp. 289–306. doi: 10.1002/malq.19740201902 (cit. on p. 35).

[Barendregt 1985] Henk Barendregt. *The Lambda Calculus - Its Syntax and Semantics.* Vol. 103. Studies in Logic and the Foundations of Mathematics. North-Holland, 1985. isbn: 978-0-444-86748-3 (cit. on pp. 11, 27).

[Barendregt 1991] Henk Barendregt. “Introduction to generalized type systems”. *Journal of Functional Programming* 1.2 (1991), pp. 125–154. doi: 10.1017/S0956796800020025 (cit. on pp. vii, 41).

[Barendregt 1993] Henk Barendregt. “Lambda calculi with types”. In: *Handbook of Logic in Computer Science (Vol. 2): Background: Computational Structures.* USA: Oxford University Press, Inc., 1993, pp. 117–309. isbn: 0198537611 (cit. on p. 32).

[Barendregt and Barendsen 2000] Henk Barendregt and Erik Barendsen. *Introduction to Lambda Calculus.* Mar. 2000. url: https://www.cse.chalmers.se/research/group/logic/TypesSS05/Extra/geuvers.pdf (visited on 09/29/2024) (cit. on p. 11).

[Barendregt, Dekkers, *et al.* 2013] Henk Barendregt, Wil Dekkers, and Richard Statman. *Lambda Calculus with Types.* Perspectives in Logic. Cambridge University Press, 2013 (cit. on pp. 29, 32, 38).

[Christiansen 2023] David Thrane Christiansen. *Functional Programming in Lean.* Microsoft, 2023. url: https://leanprover.github.io/functional_programming_in_lean/ (cit. on pp. 54, 55, 60, 62).

[Church 1936] Alonzo Church. “An unsolvable problem of elementary number theory”. *American Journal of Mathematics* 58 (1936), p. 345. url: https://api.semanticscholar.org/CorpusID:14181275 (cit. on p. 17).

[Church 1940] Alonzo Church. “A formulation of the simple theory of types”. *The Journal of Symbolic Logic* 5.2 (1940), pp. 56–68. issn: 00224812. url: http://www.jstor.org/stable/2266170 (visited on 09/04/2024) (cit. on pp. 9, 29).

[Connolly and Begg 2015] T. Connolly and C. Begg. *Database Systems: A Practical Approach to Design, Implementation, and Management, Global Edition*. Pearson Education, 2015. isbn: 9781292061849 (cit. on p. 49).

[Coquand and Huet 1988] Thierry Coquand and Gérard Huet. "The calculus of constructions". *Information and Computation* 76.2 (1988), pp. 95–120. issn: 0890-5401. doi: https://doi.org/10.1016/0890-5401(88)90005-3. url: https://www.sciencedirect.com/science/article/pii/0890540188900053 (cit. on p. 41).

[Ferreirós 2023] José Ferreirós. "The Early Development of Set Theory". In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta and Uri Nodelman. Summer 2023. Metaphysics Research Lab, Stanford University, 2023 (cit. on p. 7).

[Gödel 1944] Kurt Gödel. "Russell's mathematical logic". In: *The Philosophy of Bertrand Russell*. Ed. by Kurt Gödel. Northwestern University Press, 1944, pp. 123–154 (cit. on p. 9).

[Heijenoort 1967] Jean van Heijenoort, ed. *From Frege to Gödel: A Source Book in Mathematical Logic, 1879-1931*. Cambridge, MA, USA: Harvard University Press, 1967 (cit. on pp. 7, 8).

[Hietala and Torlak 2024] Kesha Hietala and Emina Torlak. *Lean Into Verified Software Development*. Apr. 2024. url: https://aws.amazon.com/blogs/opensource/lean-into-verified-software-development/ (visited on 12/04/2024) (cit. on p. 45).

[Howard 1969] William A. Howard. "The formulae-as-types notion of construction". In: 1969. url: https://api.semanticscholar.org/CorpusID:118720122 (cit. on p. 37).

[Leroy 2009] Xavier Leroy. "Formal verification of a realistic compiler". *Commun. ACM* 52.7 (July 2009), pp. 107–115. issn: 0001-0782. doi: 10.1145/1538788.1538814. url: https://doi.org/10.1145/1538788.1538814 (cit. on p. 45).

[Lovnički 2018] Sandro Lovnički. *pLam*. 2018. url: https://github.com/slovnicki/pLam (visited on 09/25/2024) (cit. on p. 20).

[McKinna 2006] James McKinna. "Why dependent types matter". *SIGPLAN Not.* 41.1 (Jan. 2006), p. 1. issn: 0362-1340. doi: 10.1145/1111320.1111038 (cit. on p. 2).

[McKinney 2010] Wes McKinney. "Data Structures for Statistical Computing in Python". In: *Proceedings of the 9th Python in Science Conference*. Ed. by Stéfan van der Walt and Jarrod Millman. 2010, pp. 56–61. doi: 10.25080/Majora-92bf1922-00a (cit. on pp. 2, 50).

[Morgan 1990] Carroll Morgan. *Programming from specifications.* USA: Prentice-Hall, Inc., 1990. isbn: 0137262256 (cit. on p. 26).

[Nederpelt and Geuvers 2014] Rob Nederpelt and Herman Geuvers. *Type Theory and Formal Proof: An Introduction.* Cambridge University Press, 2014 (cit. on pp. 41, 44).

[Pierce 2002] Benjamin C. Pierce. *Types and Programming Languages.* 1st. The MIT Press, 2002. isbn: 0262162091 (cit. on pp. vii, 1, 9, 11, 29, 35, 38).

[Ramsey 1926] F. P. Ramsey. "The foundations of mathematics". *Proceedings of the London Mathematical Society* s2-25.1 (1926), pp. 338–384 (cit. on p. 9).

[Rice 1953] H. G. Rice. "Classes of recursively enumerable sets and their decision problems". *Transactions of the American Mathematical Society* 74.2 (1953), pp. 358–366. issn: 00029947, 10886850. url: http://www.jstor.org/stable/1990888 (visited on 01/16/2025) (cit. on p. 38).

[B. Russell 1903] Bertrand Russell. *Principles of Mathematics.* 1st ed. Cambridge University Press, 1903 (cit. on pp. 7, 8).

[B. Russell 1908] Bertrand Russell. "Mathematical logic as based on the theory of types". *American Journal of Mathematics* 30.3 (1908), pp. 222–262. doi: 10.2307/2272708 (cit. on p. 8).

[Sørensen and Urzyczyn 2006] M. H. Sørensen and P. Urzyczyn. *Lectures on the Curry-Howard Isomorphism.* Elsevier, 2006 (cit. on p. 35).

[Stack Overflow 2024] Stack Overflow. *Stack Overflow Developer Survey 2024.* 2024. url: https://survey.stackoverflow.co/2024/technology#2-programming-scripting-and-markup-languages (visited on 09/27/2024) (cit. on p. 17).

[Syme *et al.* 2013] Donald Syme, Keith Battocchi, Kenji Takeda, Donna Malayeri, and Tomas Petricek. "Themes in information-rich functional programming for internet-scale data sources". In: *Proceedings of the 2013 Workshop on Data Driven Functional Programming.* DDFP '13. Rome, Italy: Association for Computing Machinery, 2013, pp. 1–4 (cit. on p. 62).

[Tejiščák 2020] Matúš Tejiščák. *idris-data-frame.* 2020. url: https://github.com/ziman/idris-data-frame (cit. on pp. 53, 63).

[Turing 1936] Alan M. Turing. "On computable numbers, with an application to the Entscheidungsproblem". *Proceedings of the London Mathematical Society* 2.42 (1936), pp. 230–265 (cit. on p. 18).

[Wadler 2015] Philip Wadler. "Propositions as types". *Commun. ACM* 58.12 (Nov. 2015), pp. 75–84. issn: 0001-0782. doi: 10.1145/2699407. url: https://doi.org/10.1145/2699407 (cit. on p. 37).

[Wells 1999] J.B. Wells. “Typability and type checking in system f are equivalent and undecidable”. *Annals of Pure and Applied Logic* 98.1 (1999), pp. 111–156. issn: 0168-0072. doi: https://doi.org/10.1016/S0168-0072(98)00047-5. url: https://www.sciencedirect.com/science/article/pii/S0168007298000475 (cit. on p. 34).

[Whitehead and B. A. Russell 1927] Alfred North Whitehead and Bertrand Arthur Russell. *Principia Mathematica*. 2nd ed. Cambridge University Press, 1927 (cit. on p. 8).

[Wickham *et al.* 2023] Hadley Wickham, Romain François, Lionel Henry, Kirill Müller, and Davis Vaughan. *dplyr: A Grammar of Data Manipulation*. R package version 1.1.4, https://github.com/tidyverse/dplyr. 2023. url: https://dplyr.tidyverse.org (cit. on p. 53).

[Ziliani 2012] Beta Ziliani. *Strong Normalization for Simply Typed Lambda Calculus*. July 2012. url: https://people.mpi-sws.org/~dg/teaching/pt2012/sn.pdf (visited on 11/05/2024) (cit. on p. 32).