ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

Lapis-rs: a Dedukti type checker based on term graphs

Relatore:
Chiar.mo Prof.
Sacerdoti Coen Claudio

Presentata da:
Pizzo Nicolò

Sessione II
Anno Accademico 2022-23

Lapis-rs

A Dedukti type checker based on term graphs

Nicolò Pizzo

Abstract

Logical frameworks are formal systems that provide a meta-language for specifying and manipulating other logical systems. Logical frameworks play a crucial role in the field of formal verification. $\lambda\Pi$-calculus modulo is a modern and elegant logical framework for expressing type systems. This framework introduces the notion of dependent types, hence enabling more expressive systems. Moreover, $\lambda\Pi$ calculus modulo introduces the notion of rewrite rules, that provide a more flexible and expressive system, particularly useful for encoding type systems. Dedukti is a logical framework that implements $\lambda\Pi$-calculus modulo theory.

Many of the existing implementations of proof assistants and proof checkers, such as Dedukti, use de Brujin indices for representing and manipulating λ-terms. De Brujin indices provide a handy structure for handling λ terms and avoid issues such as variable renaming in many operations.

An alternative approach to handling λ terms is given by term graphs, graph based data structures where nodes represent subterms and edges represent relations between subterms. Term graphs provide the ability to easily represent shared subterms with a single node: this mechanism helps in eliminating redundancy and in handling efficiently the memory. For this reason, term graphs provide very efficient algorithms for computing operations on sub terms, such as β reduction and α-equivalence.

This work aims at the implementation of a bidirectional typechecker based on term graphs, compatible with Dedukti. One important task of this thesis consists in implementing an algorithm for checking α-equivalence using sharing equality, an algorithm that uses term graphs and computes the check in linear time.

Contents

Chapter 1

Introduction

In the realm of formal verification and mathematical reasoning, proof assistants emerge as powerful tools that facilitate the creation and verification of mathematical proofs. These sophisticated software systems play a pivotal role in the domain of formal methods, where precision and rigor are paramount. At their core, proof assistants are interactive and automated systems designed to assist mathematicians, computer scientists, and engineers in the development and verification of formal proofs. They leverage computational power to ensure the correctness of logical arguments, minimizing the risk of human error and enhancing the reliability of mathematical results. These systems are equipped with formal languages that allow users to express mathematical statements and construct step-by-step proofs. Proof assistants are not mere automated provers: they actively involve users in the proof process, providing a collaborative environment where human intuition and expertise synergize with machine-assisted verification. As technology advances, proof assistants continue to evolve, expanding their applicability beyond pure mathematics into fields such as software verification, hardware design, and formalized reasoning about complex systems. In several of these systems, proofs are expressed by λ-terms, and typing corresponds to proof checking.
$\lambda\Pi$-calculus modulo is a logical framework that enables an elegant and simple encoding for type systems. At its core, $\lambda\Pi$ modulo provides a framework for specifying and reasoning about computations. It introduces dependent types, allowing types to depend on terms, enabling a more expressive and fine-grained type system. This expressive capability is crucial for formalizing intricate mathematical structures and representing complex computations with a high level of precision. Dedukti and kontroli-rs represent two interesting and efficient implementations of $\lambda\Pi$-calculus modulo.
An important task in these systems is the possibility to manage and visualize efficiently terms. Most of the systems implement the De Brujin index for handling and comparing terms and expressions. There exist more efficent algorithms for handling terms based on term graphs. Term graphs are a data structure that easily represents terms and expression; this data structure is interesting for several reasons, as we will see throughout

this thesis. In particular, term graphs are inducted by proof nets, so their formalization is more accurate and sophisticated than De Brujin index; moreover, the cost of many operations is singificantly reduced when compared to the De Brujin index: for example, a single step of β-reduction costs $O(1)$, and the check for α-equivalence can be done in time linear with respect to the size of the terms.
In this work, I want to explore the realm of $\lambda\Pi$-calculus modulo, implementing a bidirectional typechecker that uses term graphs for checking α-equivalence compatible with Dedukti. Moreover, I implemented the typechecker in Rust. A first reason for this choice is the fact that it is needed a language that doesn't implement garbage collection for doing becnhmarks on the algorithms we present. Moreover, this choice leads to the possibility of exploring thread-safe parallel computing and to the possibility of comparing this system with `kontroli`, an existing typechecker for Dedukti written in Rust that uses De Brujin index.

1.1 Structure

In Chapter 2 we will talk about the theoretical foundations of my work. We will explain in detail $\lambda\Pi$-calculus, type theory and Dedukti. Moreover, we will explain in detail the theoretical foundations of some properties and algorithms I implemented, such as syntax direction, bidirectional typingterm graphs and sharing equality.
In Chapter 3, we will talk about the extensions of the algorithms introduced in Section 2 for $\lambda\Pi$ modulo. Moreover, we will see some implementation details (Section 3.5) that represent both strong and weak points of the type checker.
Finally, in Chapter 4 we will present the benchmark and the obtained results; moreover, we will briefly introduce some ideas for future works about this typechecker.

Chapter 2

Theoretical Foundation

This chapter covers the theoretical, non-original studies I have used for my thesis. In Section 2.1 I will talk about $\lambda\Pi$-modulo calculus. Section 2.2 will cover the main features of Dedukti, and will show some simple examples. In Section 2.4 I will give a detailed explanation about bidirectional typechecking, why it is useful and the new typing rules used. In Section 2.5 I will cover the De Brujin index technique. In Section 2.6 we will introduce the structure of term graphs, and will se how it provides efficient ways for handling terms and expressions. Finally, in Section 2.7 we will show an algorithm based on term graphs for checking α-equivalence between terms that goes under the name of *sharing equality*

2.1 $\lambda\Pi$-modulo calculus

In this chapter I will briefly explain what is $\lambda\Pi$-modulo calculus. In Section 2.1.1 I will talk about $\lambda\Pi$ calculus and I will present the typing rules adopted. In Section 2.1.2 I will talk about $\lambda\Pi$-modulo calculus and its typing rule extension.

2.1.1 $\lambda\Pi$ calculus

$\lambda\Pi$ calculus is an extension of λ calculus with dependent types. It introduces a new type called *Type* inhabited by types. $\lambda\Pi$ calculus introduces the possibility to define dependent types: this family of types is stronger than simple typing, since types can depend on *terms.* For instance, we can define the type encoding "a vector of size `n`", that depends on the term `n`. Additionally, we introduce a special type, $Kind$, to type terms such as $Type$, $A \to Type$, and so on. Finally, we extend the notion of function $A \to B$ to a dependent product, $\Pi x : A.B$; if x does not appear in B, we have $\Pi x : A.B \equiv A \to B$. Formally, we define a $\lambda\Pi$ term as follows.

$$t ::= \langle x, \lambda x.t, \Pi x : t.t, tt \rangle$$

Before showing the typing rules for $\lambda\Pi$-calculus, we introduce some notations we will use:

- Empty context: the empty context will be denoted with the notation $[\cdot]$.
- Context: with Γ we denote the context. The context is composed by a list of pairs (x, t), where x is the name of term, and t is its type.
- β-equivalence: with $\equiv_\beta$ we denote the notion of β-equivalence, that is the reflexive, symmetric, transitive closure of the β-step $\longrightarrow_\beta$.

We are now ready to define the typing rules of the $\lambda\Pi$ calculus.

Well-formedness of empty context

$$\frac{}{[\cdot] \text{ well-formed}}$$

Declaration of a type or type family

$$\frac{\Gamma \vdash A : \mathit{Kind}}{\Gamma \vdash x : A}$$

Declaration of an object variable

$$\frac{\Gamma \vdash A : \mathit{Type}}{\Gamma \vdash x : A}$$

Type

$$\frac{\Gamma \text{ well-formed}}{\Gamma \vdash \mathit{Type} : \mathit{Kind}}$$

Variable

$$\frac{x : A \in \Gamma \qquad \Gamma \text{ well-formed}}{\Gamma \vdash x : A}$$

Product for types

$$\frac{\Gamma \vdash A : \mathit{Type} \qquad \Gamma, x : A \vdash B : \mathit{Type}}{\Gamma \vdash \Pi x : A.\ B : \mathit{Type}}$$

Product for kinds

$$\frac{\Gamma \vdash A : \mathit{Kind} \qquad \Gamma, x : A \vdash B : \mathit{Kind}}{\Gamma \vdash \Pi x : A.\ B : \mathit{Kind}}$$

Abstraction for type families

$$\frac{\Gamma \vdash A : \mathit{Type} \qquad \Gamma, x : A \vdash t : B \qquad \Gamma, x : A \vdash B : \mathit{Kind}}{\Gamma \vdash (\lambda x : A.\ t) : \Pi x : A.\ B}$$

Abstraction for object variables

$$\frac{\Gamma \vdash A : \mathit{Type} \qquad \Gamma, x : A \vdash t : B \qquad \Gamma, x : A \vdash B : \mathit{Type}}{\Gamma \vdash (\lambda x : A.\ t) : \Pi x : A.\ B}$$

Application

$$\frac{\Gamma \vdash M : (\Pi x : A.\ B) \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : [N/x]B}$$

Conversion

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash A : Type \qquad \Gamma \vdash B : Type \qquad A \equiv_\beta B}{\Gamma \vdash t : B}$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash A : Kind \qquad \Gamma \vdash B : Kind \qquad A \equiv_\beta B}{\Gamma \vdash t : B}$$

2.1.2 $\lambda\Pi$-modulo theory

$\lambda\Pi$-modulo theory extends $\lambda\Pi$ calculus by introducing *rewriting rules.* The major difference between $\lambda\Pi$ and $\lambda\Pi$ modulo is that in the former the context contains only variables, while in the latter the context contains variables *and* rewrite rules. Moreover, the latter introduces the concepts of *global* and *local* context. Global contexts encompass both variables and rewrite rules, whereas local contexts can only contain **declarations for object variables**. The first rule we introduce is the declaration of an object variable within a local context.
Declaration of an object variable in a local context

$$\frac{\Gamma \vdash \Delta \text{ local} \qquad \Gamma, \Delta \vdash A : Type}{\Gamma, \Delta \vdash x : A \text{ local}}$$

If Γ is a well-formed global context we denote by $\longrightarrow_{\beta\Gamma}$ the smallest relation, closed by context, such that if t rewrites to u for some rule in Γ, or if t β-reduces to u, then $t \longrightarrow_{\beta\Gamma} u$. With $\equiv_{\beta\Gamma}$ we indicate the reflexive-symmetric-transitive closure of the relation $\longrightarrow_{\beta\Gamma}$.
Before showing the typing rules for rewrite rules, we introduce the notion of **metavariables**. In the context of the $\lambda\Pi$ calculus modulo, metavariables are variables used as placeholders in expressions. Metavariables are often used to represent unknown terms, types, or contexts during the formulation of rules or theorems. They are placeholders that can be replaced with concrete expressions when needed. They provide a level of abstraction and generality when defining rules in formal systems, allowing the rules to apply to a wide range of specific instances.

$$\frac{\Gamma \text{ well-formed} \qquad \Gamma, \Delta \vdash l : A \qquad \Gamma, \Delta \vdash r : B \qquad A \equiv_{\beta\Gamma} B}{\Gamma \vdash l \longrightarrow^{\Delta} r \text{ well-formed}}$$

From the previous rule we can notice that rewrite rules preserve the typing. We need to rewrite the typing rules for conversion using the $\equiv_{\beta\Gamma}$ congruence.
Conversion with rewrite rules

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash A : Type \qquad \Gamma \vdash B : Type \qquad A \equiv_{\beta\Gamma} B}{\Gamma \vdash t : B}$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash A : Kind \qquad \Gamma \vdash B : Kind \qquad A \equiv_{\beta\Gamma} B}{\Gamma \vdash t : B}$$

Finally, we show some syntactical constraints on rewrite rules to ensure soundness and termination.

1. The right-hand side of a rewrite rule should contain a subset of the variables used in the left-hand side. This is done to ensure termination and avoid circular rewriting.
2. The head of the left-hand side of the rule must be a variable.

2.2 Dedukti

Dedukti [6] is a logical framework based on the λΠ-calculus modulo. `lapis-rs` is based on the syntax and implementative choices adopted by this framework. In this section I will outline the syntax of Dedukti, using some simple examples. In Dedukti, the comments are enclosed between `(; ... ;)`.
In Dedukti there are two kinds of declarations: declarations of static symbols, and declarations of definable symbols. Static symbols cannot appear at head of rewrite rules, and for this reason they are injective with respect to conversion. To declare static symbols, the following syntax is used.

```
(; Defining the type for natural numbers. ;)
Nat: Type.

(; Defining the zero for natural number ;)
z: Nat.
```

To declare definable symbols, the following syntax is used.

```
def one : Nat.

[ ] one --> S z.
```

The product is expressed with the syntax `A -> B`. If we are expressing a dependent type, the product can use the syntax `a: A -> B`. The following is a simple example for using product in both ways.

```
(; Definition of the successor function. ;)
def S : Nat -> Nat

(; Defining a type that depends on a natural number. ;)
Vec: Nat -> Type.
```

To express a rewrite rule Dedukti uses the syntax `[ ctx ] l --> r.` If we want to describe a set of rewrite rules, we only add the `.` at the end of the last rule. It is possible to omit the type for the variables in the context. Let's say we want to define the `plus` function between two numbers.

```
def plus: Nat -> Nat -> Nat.
[ n ] plus z n --> n
[ m, n ] plus (S m) n --> S (plus m n).
```

We can also express a rewrite rule with the syntax `def f := b.`. For example

```
def one: Nat := S z.
```

is just syntactic sugar for

```
def one: Nat.
[ ] one --> S z.
```

The abstractions are specified with `x: T := B` where `T` is an optional parameter. For example, the following code is correct.

```
def K := x: Nat => z.
def K2 : Nat -> Nat := x => z.
```

This code is syntactically and semantically correct. I will explain in detail why this code is peculiar, and how to address the problems it poses during the type checking process in section 2.4.
Another syntactic term introduced by Dedukti are theorems. Theorems are defined with the keyword `thm`. A theorem definition is *opaque* meaning that the defined symbol do not reduce to the body of the definition. This means that the rewrite rule is not added to the system. For example, for the code

```
thm three := S ( S ( S z ) ).
```

The rewrite rule `S ( S ( S z ) ).` is not added to the system, but only the definition of `three`.
Finally, Dedukti syntax accepts *wildcards*: when a variable is not used on the right-hand side of a rewrite rule, it can be replaced by an underscore on the left-hand side. The following example shows how wildcards can be used.

```
def mult: Nat -> Nat -> Nat
[ ] mult _ zero --> zero.
```

Now I want to show a simple example with dependent types to see Dedukti effectively in action. We have already defined the type for the "Array of size `n`". As for the code we have right now, we cannot instantiate this type, so we will inductively define a constructor for this type. The constructor will take a number `n`, a vector of size `n`, and will return a vector of size `n+1`.

```
(; Empty vector ;)
nil: Vec z.

(; Constructor ;)
cons: n: Nat -> Vec n -> Vec (S n).
```

Now, we can define a function to append one vector to another.

```
def append : n: Nat -> Vec n -> m: Nat -> Vec m -> Vec (plus n m).
[ v, m ] append z nil m v --> v
[ n, u, m, v ] append (S n) (cons n u) m v -->
cons (plus n m) (append n u m v).
```

This code can be very interesting to explain why Dedukti and similar tools can be useful: if we had written an erroneous code, the type checker would have failed, suggesting that the result would not actually be a `Vec (plus n m)`. This is very useful for critical scenarios, where the correctness of a program must be known with certainty. For example, if we had written the following code, instead of the correct one

```
def append : n: Nat -> Vec n -> m: Nat -> Vec m -> Vec (plus n m).
[ v, m ] append z nil m v --> v
[ n, u, m, v ] append n (cons n u) m v -->
cons (plus n m) (append n u m v).
```

Dedukti would throw the following error.

```
Error while typing
vec.cons (vec.plus n m) (vec.append n u m v)

...

---- Expected:
vec.Vec (vec.plus n m)
---- Inferred:
vec.Vec (vec.S (vec.plus n m))
```

Telling us that we are returning a `Vec S (plus n m)` instead of `Vec (plus n m)`. More interesting and detailed examples are shown in [6].

2.3 Syntax Directed

When implementing a judgement we need to understand if each slot position is an input or an output. For example, for the judgement $\Gamma \vdash e : A$, we should always know which of Γ, e, A are input and which are outputs. We will assume that everything before the : is the input, and everything after is an output. In the previous example, we would have Γ, e as input, and A as output. Syntax direction is an useful property whenever we want to obtain an algorithm starting from an inference system. It refers to the situation where the inputs conform to the conclusion of only one rule. However, the typing rules outlined in section 2.1.1 do not make our system syntax-directed. In particular, the conversion rule introduces ambiguity between input and output.

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash A : Type \qquad \Gamma \vdash B : s \qquad A \equiv_{\beta\Gamma} B}{\Gamma \vdash t : B}\ s \text{ is a sort}$$

This rule overlaps with every other rule: this makes undecidable when the conversion rule should be triggered as the derivation trees are built. To solve the problem, we need to remove the conversion rules from the rule-set, so we focus on when the conversion rule should occur in the typing rules described in Section 2.1. Let's analyze the rule for application.

$$\frac{\Gamma \vdash M : (\Pi x : A.\ B) \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : [N/x]B}$$

In this case, A appears twice as output: we are implicitly requiring that the output of M is a product where the bound variable has the same type as N. Moreover, we are requiring that the output of $\Gamma \vdash M$ is exactly a product, so this too is an implicit use of the conversion rule. In particular, in $\lambda\Pi$ calculus we want that the output of $\Gamma \vdash M$ is a product up to weak head normal form. Hence, we denote with the judgement $u \twoheadrightarrow v$ the fact that the weak head normal form of u is v.
For this reason, we eliminate the conversion rules and slightly change the application rule.

$$\frac{\Gamma \vdash M : T \qquad T \twoheadrightarrow (\Pi x : A.\ B) \qquad \Gamma \vdash N : A' \qquad A \equiv_{\beta\Gamma} A'}{\Gamma \vdash MN : [N/x]B}$$

2.4 Bidirectional Typechecking

Regular typechecking poses the big problem of annotating each variable: the typechecking process assumes that every variable is correctly annotated, so that the check for a certain variable is always decidable. However, modern programming languages and frameworks let the programmer write a program flexible and clean code, omitting types when they are too complex or too naive, preserving the benefits of static typing. The technique that makes it possible, goes under the name of **bidirectional typing** or **bidirectional typechecking**.
Bidirectional type checking is an approach in type theory and programming language design that divides the process of type checking into two phases: checking and synthesis. Type checking is applied in scenarios where the type of an expression is known or can be

easily inferred: for example, when checking the arguments of a function call. Type inference is useful in many contexts, such as polymorphic functions, generic types, dependent types, and in general whenever the type can be determined based on how the expression is used. Clearly, using both techniques, allows for more flexible code, preserving the benefits of static type checking.
Dedukti uses bidirectional typechecking to use an elegant syntax, enabling the omission of types in some circumstances. One scenario where it is possible to omit types is with variables in a local context. In the following example, the variables `m, n` do not have an explicit type, so without bidirectional typing it couldn't be possible to correctly typecheck the program without introducing metavariables and unification, that make the algorithm more complex and the error messages hard to understand.

```
def plus : Nat -> Nat -> Nat
[ m, n ] plus (S m) n --> S (plus m n).
```

Another, non-trivial, scenario where bidirectional typing is useful is when we use abstractions. Let's consider the following example.

```
def K2 : Nat -> Nat := x => S (S z).
```

Dedukti accepts this kind of code. With bidirectional typechecking we know that `K2 : Nat -> Nat`, hence, because of injectivty of `->`, we must have `x : Nat`. With bidirectional typechecking, it is possible to catch this syntactic phenomenon and treat it correctly. Moreover, abstractions can appear in either side of the rule, so we can have more non-trivial cases where it is useful to not express explicitly thee type of the bound variables, like the following.

```
def ignore_abs : Nat -> (Nat -> Nat) -> Nat.
[ y ] ignore_abs y (x => plus y (S x)) --> y.
```

As described in [9], there are many ways to design a bidirectional typing system. The rules designed for the system will be shown in detail in Section 3.1.

2.5 De Brujin index

In lambda calculus, de Brujin indices are a way of representing variables. Instead of using variable names, de Brujin indices encode the information about the binding structure of variables directly into the syntax of terms.
The De Brujin index of a certain variable is defined as the number of binders that are in scope between the occurrence of the variable and its corresponding binder. For instance, the lambda expression

$$\lambda x.\lambda y.x$$

becomes

$$\lambda\lambda 2$$

The following example is slightly more intricate, but provides a better perspective in situations involving complex lambda functions.

$$\lambda z.(\lambda y.y(\lambda x.x))(\lambda x.zx)$$

$$\lambda(\lambda 1(\lambda 1))(\lambda 21)$$

De Brujin indices provide a way to avoid issues related to variable capture and renaming in substitution operations, as the indices capture the binding information explicitly.
De Brujin indices are often used in the context of α-equivalence in lambda calculus. α-equivalence is a notion of equivalence between lambda terms that takes into account variable renaming. Two lambda terms are considered α-equivalent if they only differ in the names of their bound variables. For example

$$\lambda x.\lambda y.xy \ \equiv_\alpha \ \lambda a.\lambda b.ab$$

Using de Brujin indices, we would check the trivial case

$$\lambda\lambda 21 \equiv_\alpha \lambda\lambda 21$$

Using de Brujin indices simplifies the process of checking α-equivalence because there are no variable names to compare directly. Instead, we can compare the structures of the terms and how they bind variables through their indices. In Dedukti, De Brujin indices are used for higher-order abstract representation (HOAS) of terms [4].
As we will see in section 2.7, an alternative approach to verify α-equivalence is employed by the sharing equality algorithm using term graphs.

2.6 Term Graphs

As we will see in Section 2.7, sharing equality is an algorithm for checking α-equivalence between terms. The sharing equality algorithm works on a data structure known as **term graphs**. In this section, we will explain what they are and will show some examples.
Term graphs are a representation technique used to model and analyze the structure of terms in mathematical expressions, programming languages, or formal systems. They provide a visual and efficient way to represent and manipulate complex data structures, expressions, or computations. Term graphs use a graph-based structure to represent terms. In this representation, nodes in the graph correspond to subterms, and edges represent the relationships or operations between these subterms. The graph structure provides a concise and visual way to capture the hierarchical and recursive nature of terms. One notable feature of term graphs is the ability to represent shared subterms

with a single node. This sharing mechanism helps eliminate redundancy in the representation of terms, making it more efficient in terms of memory usage and computational complexity. Term graphs are commonly used in functional programming languages where expressions can involve repeated subterms. By representing shared subterms as shared nodes in a graph, functional programming languages can optimize memory usage and improve the efficiency of certain operations. The use of term graphs is closely related to graph reduction strategies in functional programming languages and plays a significant role in optimizing the evaluation of expressions.
The λ-graphs presented in [8] represent an encoding of a λ term under the form of nodes of a directed graph. Graphically, λ-terms can be seen as syntax trees. We can distinguish between three kinds of node:

- Application: an application node has exactly two children, called *left* and *right*. We write `App(n, m)` for an application node whose left child is `n` and right child `m`.
- Abstraction: an abstraction node has exactly two children, called respectively its bound variable and its *body*. We write `Abs(v, n)` for an abstraction node whose bound variable is `v` and body is `n`.
- Variable: we can have two kinds of variable:
  - free: a free variable has no children, and is denoted by `Var()`.
  - bound: a bound variable has exactly one child called its *binder*. We write `Var(b)` for a variable term with binder `b`. The binding edge is represented with a dashed line.

One example of term graphs is shown in Figure 2.1.

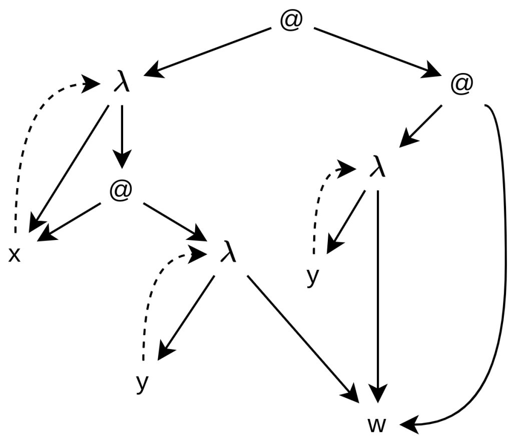


Figure 2.1: Term graph for $(\lambda x.x(\lambda y.w))((\lambda y.w)w)$

In the following example, we will show how term graphs are useful and efficient structure to reason about λ calculus. Suppose we have the term

$$(\lambda x.M)N$$

where M, N are λ expressions, and M contains at least once x. We want to compute a single step of β-reduction for this expression. With term graphs, we introduce a special edge (drawn as purple in Figure 2.2) from the bound variable (in this case, x) to the λ expression we substitute. Using term graphs, this operation only costs $O(1)$. The computations on the resulting graph will then take into account that the substitution has happened and will use the substituted expression N instead of the variable x. The computing of the previous λ expression can be found in Figure 2.2.

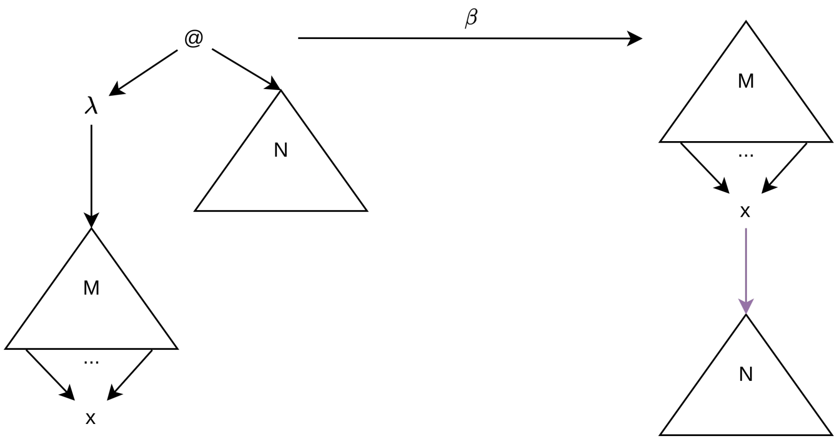


Figure 2.2: Term graph for $(\lambda x.x(\lambda y.w))((\lambda y.w)w)$

This definition for λ-graphs is given by [8] in the context of untyped λ calculus. Since this study is focused on $\lambda\Pi$-calculus modulo, we must extend this concept of term graphs on product terms and typed variables. In particular, we have these new kind of nodes:

- Product: a product node has two children, called respectively its bound variable and its *body*. We write `Prod(x, n)` for a product node whose bound variable is `x` and body is `n`.
- Variable: we redefine variable nodes. As before we have two kinds of variable:
  - free: a free variable has exactly one child, called its *type*; it is denoted by `Var(typ)` where `typ` is the type of the variable node.
  - bound: a bound variable has two children, called its *type* and its *binder*. We write `Var(typ, b)` for a bound variable whose type is `typ` and binder `b`.
- Sorts: we introduce the sorts $\mathit{Type}, \mathit{Kind}$ described in Section 2.1 as special kinds of node for typing correctly expressions.

Graphically we will denote the type of a variable with a red edge starting from the variable. An example of term graphs in the context of $\lambda\Pi$-calculus is shown in Figure 2.3.

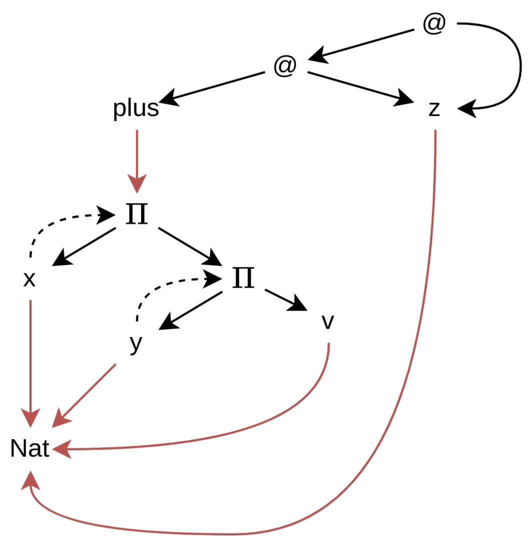


Figure 2.3: Term graph for $plus\ z\ z$ where $plus : \Pi x : Nat.\Pi x : Nat.Nat$ and $z : Nat$

2.6.1 Implementation details

2.6.2 Rust smart pointers

Smart pointers are structures that implement some traits for handling pointers, like dereferencing, deallocating and so on. Two of these smart pointers are very interesting: `Rc` and `Weak`: these pointers enable garbage collection in Rust with the technique of *reference count*. By default, Rust doesn't use garbage collection, but when using these pointers it is possible to enable it with the reference counting technique. As the name suggests, the `Rc` pointers maintain *strong* references, while `Weak` pointers maintain weak references. Smart pointers are particularly useful in handling circular data structures: without them, in fact, the reference counting technique would not recognize garbage and would run indefinitely.

2.6.3 Representation of nodes

For implementing the structure of the lambda-nodes described in [8], I used the smart pointer previously described. Each node, except for the sorts `Kind, Type` maintain the fields `undir, canonic, building, queue` used for checking the sharing equality of nodes presented in Section 2.7. The following snippet of code describes the structure of the `App` node implemented.

```rust
pub enum LNode {
  App {
    left: Rc<Self>,
    right: Rc<Self>,
    parent: RefCell<Vec<Weak<Self>>>,
    undir: RefCell<Vec<Weak<Self>>>,
    canonic: RefCell<Weak<Self>>,
    building: RefCell<bool>,
    queue: RefCell<VecDeque<Weak<Self>>>,
  },
}
```

As we can see, `left` and `right` fields are `Rc` pointers, because we want strong pointers to the children of the application. On the other hand, the parents of a node are `Weak` pointers, graphically represented previously with dashed edges. Note also that `LNode` is described as an `enum`: this is how Rust defines *algebraic data types*. The use of `enums` gives the programmer the possibility to use sophisticated and efficient ways to handle and recognize data, such as *pattern matching*.

2.6.4 Representation of λ-graphs

In this section we will show the implementation of the structure for λ-graphs. Before doing so, we introduce the concepts of **borrowing** and of lifetime in Rust.

In Rust, borrowing a variable corresponds to creating a reference to that variable. A borrow can either be *mutable* or *immutable*: Rust imposes some constraints on the use of these two kinds of borrowing to preserve its thread-safe nature. In particular:

1. There can be as many immutable borrows as possible: these are "read-only" references, so the thread-safe nature of Rust is preserved.
2. There can only be one mutable borrow at a time; moreover, if there is a mutable borrow, all other immutable borrows are freezed and it is not possible to create new immutable borrows

A borrow is denoted by the symbol `&` if it is immutable and by the symbol `&mut` if it is mutable.

A lifetime is a construct that the Rust compiler uses to ensure all borrows are valid. A variable's lifetime begins when it is created and ends when it is destroyed. Rust denotes lifetimes with the `<'a>` syntax. λ-graphs are encoded with the structure of `LGraph`. This structure contains the field `nodes`: this is an `HashSet` of references to `LNodes`.

```rust
pub struct LGraph<'a> {
    nodes: HashSet<&'a Rc<LNode>>,
}
```
This structure holds all the references to the sub-terms that compose a λ-graph, as we will see in Section 2.7 it is important to iterate over all the nodes of the graph.

2.7 Sharing Equality

In this section I will describe the algorithm presented in [8] for checking α-equivalence in the scenario of untyped λ calculus. The idea behind the algorithm of sharing equality is that having the terms structured in λ graphs, checking for equality reduces to check for the bisimulation relation between the two term graphs. In particular, the sharing equality algorithm computes the smallest bisimulation between the given term graphs. Before talking about the algorithm for sharing equality, I present some theoretical elements useful to fully understand the algorithm. The first notion introduced is that of *equivalence* between nodes.

Definition 2.7.1 (Homogeneous nodes) *Let n, m be nodes of a λ-graph G. We say that n and m are homogeneous if they are both application nodes, or they are both abstraction nodes, or they are both free variable nodes, or they are both bound variable nodes.*

Let R be a generic binary relation over the nodes of λ-graph: we call such relation homogeneous if it only relates pairs of homogeneous nodes. Sharing equivalence requires that it is closed under some other structural rules.

$$\frac{}{n \;\mathcal{R}\; n}\,\bullet \qquad \frac{n\mathcal{R}m}{m \;\mathcal{R}\; n}\,\leftrightarrow \qquad \frac{n\mathcal{R}m \qquad m\mathcal{R}p}{n \;\mathcal{R}\; p}\,\twoheadrightarrow$$

$$\frac{\mathrm{App}(n_1, n_2) \;\mathcal{R}\; \mathrm{App}(m_1, m_2)}{n_1 \;\mathcal{R}\; m_1}\,\swarrow \qquad \frac{\mathrm{App}(n_1, n_2) \;\mathcal{R}\; \mathrm{App}(m_1, m_2)}{n_2 \;\mathcal{R}\; m_2}\,\searrow$$

$$\frac{\mathrm{Abs}(n) \;\mathcal{R}\; \mathrm{Abs}(m)}{n \;\mathcal{R}\; m}\,\downarrow \qquad \frac{\mathrm{Var}(n) \;\mathcal{R}\; \mathrm{Var}(m)}{n \;\mathcal{R}\; m}\,\circlearrowleft$$

Figure 2.4: Sharing equivalence rules

- Equivalence rules: usual rules that characterize equivalence relations (reflexivity, symmetry and transitivity)
- Bisimulation rules:
  - Downward propagation rules: rules $\swarrow$, $\searrow$ are downward propagation rules; these rules state that if two application nodes are related, their children should also related. The same goes for the relation $\downarrow$ that states that if two abstraction nodes are related, their bodies should also related.
  - Scoping rule: the rule $\circlearrowleft$ states that if two bound variable nodes are related, then also their binders should be related.

Finally, a sharing equivalence should not equate two different free variable nodes. Now we can formally define sharing equivalence.

Definition 2.7.2 ((Blind) sharing equivalence) *Let $\equiv$ be a binary relation over the nodes of a λ-graph G.*

- *$\equiv$ is a **blind sharing equivalence** if it is an equivalence relation, it is homogeneous and it is closed under the rules $\swarrow, \searrow, \downarrow$*
- *$\equiv$ is a **sharing equivalence** if it is a blind sharing equivalence, it is closed under the rule $\circlearrowleft$ and satisfies the open requirement for which $v \equiv w$ requires $v = w$ for every free variable nodes v, w.*

Another ingredient for sharing equivalence is the notion of query and of its spreading.

Definition 2.7.3 (Query) *A query Q over a λ-graph G is a binary relation over the root nodes of G.*

Every query Q induces a number of other equality requests, obtained by closing Q under the rules that every sharing equivalence has to satisfy.

Definition 2.7.4 (Spreading $R^{\#}$) *Let R be a binary relation over the nodes of a λ-graph G. The spreading $R^{\#}$ induced by R is the binary relation on the nodes of G inductively defined by closing R under the rules $\swarrow, \downarrow, \searrow$*

Now we have all the ingredients to describe the algorithm to check sharing equality. We check sharing equality in two phases:

- Blind check: building the propagated query $Q^{\#}$ and at the same time checks that it is a blind sharing equivalence.
- Variables check: verifying that $Q^{\#}$ is a sharing equivalence by checking the conditions for free and bound variables.

2.7.1 Blind check

In this section we introduce the concept behind the blind check algorithm, shown in Algorithm 1. For a detailed explanation on criterions such as completeness and correctness, along with the proof for the linearity of the algorithm, you can check [8].

The algorithm needs to enrich λ graphs with a few additional concepts, namely *canonic edges, undirected query edges, building flags* and *queues*, all grouped under the notion of *state*. We define a state $\mathcal{S}$ as either `Fail` or a tuple

```
(G, undir, canonic, building, queue)
```

where `G` is the λ graph and

- `undir` is is a multiset of undirected query edges, pairing nodes that are expected to be placed by the algorithm in the same sharing equivalence class. We denote by $\sim$ the binary relation over `G` such that $n \sim m$ iff the edge (n, m) belongs to `undir`.
- nodes may have one additional `canonic` directed edge pointing to the computed canonical representative of that node.

Algorithm 1: Blind check algorithm

```
Data: an initial state
Result: fail or final state
Procedure BlindCheck()
1       foreach node $n$ do
2           if canonic($n$) undefined then
3               BuildEquivalenceClass($n$)
            end
        end
    Procedure BuildEquivalenceClass($c$)
4       canonic($c$) := c
5       building($c$) := c
6       queue($c$) := {c}
7       while queue($c$) is non-empty do
8           $n$ := queue($c$).pop()
9           foreach parent $m$ of $n$ do
10              case canonic($m$) do
11                  undefined ⇒ BuildEquivalenceClass($m$)
12                  $c'$ ⇒ if building($c'$) then fail
                end
            end
13          foreach $\sim$ neighbour $m$ of $n$ do
14              case canonic($m$) do
15                  undefined ⇒ EnqueueAndPropagate($m$, $c$)
16                  c' ⇒ if $c' \neq c$ then fail
                end
            end
        end
    Procedure EnqueueAndPropagate($m$, $c$)
        case $m$, $c$ do
17          Abs($m'$), Abs($c'$) ⇒ create edge $m' \sim c'$
18          App($m_1, m_2$), App($c_1, c_2$) ⇒ create edges $m_1 \sim c_1$ and $m_2 \sim c_2$
19          Var($b$), Var($b'$) ⇒ ()
20          Var(), Var() ⇒ ()
21          _ , _ ⇒ fail
        end
22      canonic($m$) := c
23      queue($c$).push(m)
```

- nodes may have an additional boolean `building` that signals whether an equivalence class has or has not been constructed yet.
- nodes have a `queue` data structure that is used only on canonic representatives, and contains the nodes of the class that are going to be processed next.

Some interesting points to focus on the algorithm:

- Top-down recursive exploration: the algorithm can start at any node, but when processing a node n the algorithm recursively acts on the parents that have not been visited yet.
- Undir edges: the query is represented through undirected query edges between nodes and it is propagated on children nodes. The notion used for this kind of edges is $\sim$.
- Canonic edges: once the `BuildEquivalenceClass(c)` algorithm terminates, a canonic node is assigned to the node c, and it is representative of its sharing equivalence class.
- Building flag: each node has a boolean building flag that is mainly used to suggest failing and cyclic situations.

The procedure `BuildEquivalenceClass` is responsible for creating the sharing equivalence class for terms. It can be summarized in four steps:

1. Collect all the nodes in the sharing equivalence class of n, that is done in line 13 of the algorithm.
2. Set n as the canonical element of its class (line 4).
3. Propagate the query on the children (line 15)
4. Pushing a node in the queue, setting its canonic and propagating the query on its children through the procedure `EnqueueAndPropagate`.

2.7.2 Variables check

The variables check algorithm (Algorithm 2) takes in input the output of the blind check, and check if the `Var`-nodes of G satisfy the variable conditions for a sharing equivalence. After invocation of the `BlindCheck` algorithm, we can compare a node with its canonical representative of the class instead of checking it against all the nodes in its class. The check fails in two cases:

- When checking a free variable node: in this case, $Q^{\#}$ is not an open relation (lines $4, 5$).
- When checking a bound variable node: in this case, $Q^{\#}$ is not closed under (lines $6, 7$).

.

Algorithm 2: Variables Check

```
Data: canonic(·) representation of $Q^{\#}$
Result: is $Q^{\#}$ a sharing equivalence?
Procedure VarCheck()
1   foreach var-node $v$ do
2       $w \leftarrow$ canonic($v$)
3       if $v \neq w$ then
4           if binder($v$) or binder($w$) is undefined then
5               fail
            end
6           else if canonic(binder($v$)) $\neq$ canonic(binder($w$)) then
7               fail
            end
        end
    end
```

Chapter 3

Lapis-rs

This chapter contains the core concepts for the implementation of `lapis-rs`. In Section 3.1, we will talk in detail about the typing rules I implemented and in particular about my implementation of bidirectional typechecking. In Section 3.2 we will show the extension of the sharing equality algorithm introduced in Section 2.7 for the context of $\lambda\Pi$ calculus modulo. In Section 3.3, we will briefly explain the workflow that guided my study. In Section 3.5, we will talk about some implementation details that will show the possibility to further improve `lapis-rs`.

3.1 Rules for type checking

In this section we will enumerate the typing rules designed and implemented in the typechecker. Since we implemented bidirectional typing, we will use the judgement $x \Leftarrow A$ with the meaning of checking term `x` with type `A`; on the other hand, we use the judgement $x \Rightarrow A$ with the meaning of inferring type `A` from term `x`. Moreover, I will often use the notation $A \twoheadrightarrow A'$ with the meaning of "the weak head normal form of `A` is `A'`". Moreover, we will denote with x^T a variable x with known type T. Using term graphs described in Section 2.6, we don't need to use a context Γ to know the type of a specific variable: we have already encoded the information inside the nodes, so we will omit Γ and use the information computed. As for the rewrite rules, we assume that they are always known in the global context. The typing rules presented will be already syntax directed, and will incorporate conversion when in a context of application as described in Section 2.4.
Rewrite rule with inference on left hand-side term.

$$\frac{l \Rightarrow T \qquad r \Leftarrow T}{l \longrightarrow r}$$

Rewrite rule with inference on right hand-side term.

$$\frac{r \Rightarrow T \qquad l \Leftarrow T}{l \longrightarrow r}$$

This pair of rules makes it possible to check symbols where the left hand side of a rewrite rule is not annotated. One such rule is given by the following example

```
def K0 := x: Nat => zero.
```

It is important to note that the rules are not complete: with this set of rules, in fact, we encountered some problems in type checking Dedukti code. Investigating into the matter, we found out that Dedukti uses higher order patterns for unification, so we still can't capture those cases.
Application.

$$\frac{M \Rightarrow A \qquad A \twoheadrightarrow (\Pi x^{A'}.B) \qquad N \Leftarrow A'}{\Gamma \vdash MN \Rightarrow [N/x]\, B}$$

Abstraction where s is a sort.

$$\frac{x \Rightarrow A \qquad t \Rightarrow B \qquad A \Rightarrow \mathit{Type} \qquad B \Rightarrow s}{\lambda x.t \Rightarrow \Pi x^{A}.B}$$

Product where s is a sort.

$$\frac{x \Rightarrow A \qquad A \Rightarrow \mathit{Type} \qquad B \Rightarrow B' \qquad B' \Rightarrow s}{\Pi x^{A}.B \Rightarrow B'}$$

Variable.

$$\frac{x^{A}}{x \Rightarrow A}$$

Type sort.

$$\frac{}{\mathit{Type} \Rightarrow \mathit{Kind}}$$

Checking abstraction

$$\frac{A \twoheadrightarrow \Pi x^{A'}.B \qquad x \Leftarrow A' \qquad t \Leftarrow B}{\lambda x.t \Leftarrow A}$$

Checking untyped variable. In this case we denote the assignment of the type A to x with $\mathit{Typ}(x) \leftarrow A$.

$$\frac{}{x \Leftarrow A}\ \mathit{Typ}(x) \leftarrow A$$

Checking generic term

$$\frac{t \Rightarrow A' \qquad A \equiv_{\beta\Gamma} A'}{t \Leftarrow A}$$

With this design, bidirectional typechecking is coherent with the specifications of Dedukti: it is in fact possible to express patterns like the following for inferring the type for abstractions without having to annotate the variable `x`.

```
def K0 : Nat -> Nat := x => zero.
```

Note that this code typechecks only if `K0` is annotated.

3.2 Sharing Equality in λΠ-calculus modulo

As we have already discussed in Section 2.6, when we are in the context of Π calculus we extend the set of nodes with the `Prod` node. For this reason, we slightly change the the procedure `EnqueueAndPropagate` to accept nodes of this type (Procedure Enqueue-AndPropagate).

Procedure EnqueueAndPropagate

```
case $m$, $c$ do
    Abs($m'$), Abs($c'$) $\Rightarrow$ create edge $m' \sim c'$
    Prod($m'$), Prod($c'$) $\Rightarrow$ create edge $m' \sim c'$
    App($m_1, m_2$), App($c_1, c_2$) $\Rightarrow$ create edges $m_1 \sim c_1$ and $m_2 \sim c_2$
    Var($b$), Var($b'$) $\Rightarrow$ ()
    Var(), Var() $\Rightarrow$ ()
    Type, Type $\Rightarrow$ ()
    Kind, Kind $\Rightarrow$ ()
    _ , _ $\Rightarrow$ fail
end
canonic($m$) := c
queue($c$).push(m)
```

As for the `VarCheck` procedure, it is not necessary to apply any change to the procedure itself: in λΠ calculus, the variables can be bound to to an `Abs` node, or to a `Prod` node. One problem arises from this property: in fact, the relation $\circlearrowright$ could not be satisfied since while typechecking we can in fact have situations for which two bound variables with different binders can be in relation with each other. For example, in the following Dedukti code, we would need to check sharing equivalence between `n1` and `n2`: the first has a product as binder, while the latter has an abstraction.

```
succ: Nat -> Type.

def X: n1: Nat -> m1: succ n1 -> Nat := n2: Nat => m2: succ n2 => zero.
```

This problem is properly described and solved in Section 3.5.3. For now, it suffices to say that the `VarCheck` procedure is in fact correct also in this scenario.

3.3 Workflow

The workflow of the `lapis-rs` is illustrated in Figure 3.1. The source code of the project is publicly available at `https://github.com/nicolopizzo/lapis-rs`.

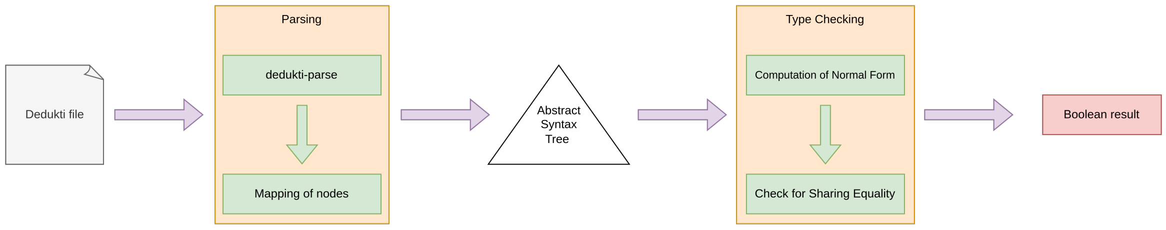
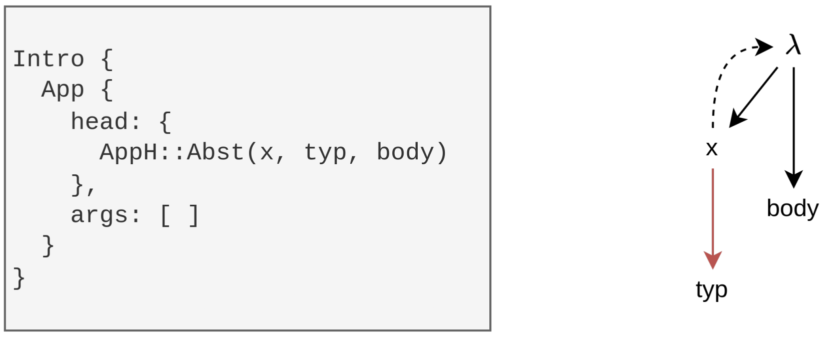


Figure 3.1: Workflow of the tool. The purple arrows represent the flow of the input/output of each phase. The orange boxes represent phases of the tool. The green boxes represent the sub-modules of the tools. The green arrows represent the dependency of the sub-modules.

The flow of the system can be descripted in 2 crucial steps:

1. Parsing: the parsing process is done with the library `dedukti-parse`. It is one of the component of `kontroli-rs`, and it parses a Dedukti file, building the corresponding syntax tree. After parsing, we implemented a function to map the nodes of the syntax tree to λ nodes. An example of this mapping is shown in figure 3.2.
2. Type checking: the typechecker represents the original part of this thesis. The typechecker includes the algorithm for reducing terms to normal form and the algorithm to check for alpha equivalence.



```
Intro {
  App {
    head: {
      AppH::Abst(x, typ, body)
    },
    args: [ ]
  }
}
```

Figure 3.2: Example for mapping an Abstraction parsed by dedukti-parse into a lambda node. On the left there is the encoding of dedukti-parse; on the right, the term created by the mapping.

3.4 Implemented functions

In this section I will briefly introduce some of the functions I implemented and will explain some of the side effect they have. This introduction is useful when we will talk about implementative details in Section 3.5 and reason about the possible efficiency and design solutions to apply to the system, in order to gain the possibility to compare `lapis-rs` with existing solutions.

3.4.1 Enqueue and Propagate

This function is the implementation of `EnqueueAndPropagate` procedure shown in Algorithm 1.

```
pub fn enqueue_and_propagate(&self, m: &Rc<LNode>, c: &Rc<LNode>)
```

The type of `self` is that of a `LGraph`. The main extension of this algorithm consists in adding a side-effect: since in Dedukti it is possible to define wildcard metavariables (see Section 2.2), we could try to check sharing equality between two different terms. For example, we might try to compare `Succ (plus n m)` with `Succ _`. In this case, the algorithm for sharing equivalence would fail, because we would try to compare an application (`plus n m`) with a meta-variable (`_`). For this reason, we slightly change the algorithm, so that it accepts this syntax and instantiate `_` with `plus n m`. Note that, this way, we can also achieve first order unification, since instead of `Succ _` we could write `Succ x`.

3.4.2 Deep clone

This function is used when computing weak head and strong normal forms. The goal of the function consists in cloning the term graphs as needed, preserving the sharing.

```
pub fn deep_clone(subs: &mut HashMap<usize, Rc<LNode>>, node:
 ↪  &Rc<LNode>) -> Rc<LNode>
```

The function takes as input a substitution map `subs` and a term `node` to be cloned. The result of the function is the cloned term, and there is no side-effect.
This function is particularly useful when computing normal forms: in triggering rewrite rules, we instantiate some metavariables for unification purposes. If we didn't clone correctly the nodes, we would not be able to reuse those rules, as metavariables would already be instantiated. This function also preserves the sharing.

3.4.3 Weak head normal form

This function computes the weak head normal form of a term labelled `node`.

```
pub fn weak_head(node: &Rc<LNode>, rules: &RewriteMap) -> Rc<LNode>
```

The function accepts as a second argument the set of rewrite rules `rules` because, in the context of λΠ-calculus modulo, as we have seen in Section 2.1, computing the weak head normal form of a term may involve rewriting it to some other term. This function propagates the side-effect of the `matches` function, since it is used to check if `node` matches the left hand side of a rewrite rule up to weak head.

3.4.4 Matches

This function is used to test structural equality between a `term` and a node `pattern`. The set of rewrite rules `rules` is passed as argument because, in some circumstances, `term` can match `pattern` up to weak head normal form.

```
pub fn matches(term: &Rc<LNode>, pattern: &Rc<LNode>, rules:
↪  &RewriteMap) -> bool
```

This function returns a boolean value. If `pattern` is a meta-variable, and has no substitution, the function turns it into `term`. For this reason, the function is applied on *deep cloned* left-hand side and right-hand side pairs of rewrite rules.

3.4.5 Type infer and type check

The following two codes are used to implement bidirectional typechecking.

```
fn type_infer(node: &Rc<LNode>, rules: &RewriteMap) ->
↪  Result<Option<Rc<LNode>>>
```

The `type_infer` function takes as input a `node` and the rewrite rules `rules` of the context. The function tries to infer the type of `node` based on the information computed at time of checking. If the function cannot infer the type for `node`, the result of the function is `Ok(None)`; on the other hand, if there it is impossible to correctly type `node`, the result of the function is `Err(...)`. The main side effect of the function consists in the substitution of a bounded var with an application argument if the `node` is an `App`.

```
fn type_check(term: &Rc<LNode>, typ_exp: &Rc<LNode>, rules: &RewriteMap)
↪  -> Result<()>
```

The `type_check` function takes as input a `term`, an expected type `typ_exp` for the term, and the set of rewrite rules `rules`. This function is responsible for checking that the type of `term` is equal to `typ_exp` using the sharing equality algorithm shown in Section 2.7. This function introduces side-effects in two different cases:

- Sharing equality: the function checks the sharing equality between two types, so it propagates the side effects of the `enqueue_and_propagate`.
- Inference: when the term is an untyped variable, the function sets the type of `term` to `typ_exp`.

3.4.6 Check rule

```
fn check_rule(lhs: &Rc<LNode>, rhs: &Rc<LNode>, rules: &RewriteMap) ->
↪  Result<()>
```

This function is responsible for type checking a rewrite rule of the form `lhs --> rhs`. This function restores the state of the meta-variables, removing the substitution happened during the check of the rule. This acts as a countering measure for the side-effects generated by the `enqueue_and_propagate` and `matches` functions.
The following two lines of code are executed at the end of the typechecking of the rule.

```
lhs.unsub_meta();
rhs.unsub_meta();
```

3.5 Implementation Details

3.5.1 Structure of the rewrite map

The choice of the data structure for maintaining the rewrite rules has been very important and impacted significantly on the efficiency of the code. In fact, during the normalization steps, the algorithm checks if any subterm has to be rewritten using rewrite rules. As the size of subterms and rewrite rules grows, it is important to keep the costs for checking if a sub-term has to be rewritten as low as possible.
Throughout the implementation of the system, the structure of the rewrite map has undergone three changes:

1. Vector of rewrite rules: the naive version of the map consisted, for a brief period, in a simple vector. Clearly such implementation was highly inefficient when the rewrite rules were too many.
2. Naive `HashMap`: the second implementation of this structure was a naive HashMap where the key of an entry was the memory address of the head of the term, and the value the vector of the rewrite rules associated with that head. This change highly improved the time efficiency of the system, leading to acceptable times of execution
3. Refined `HashMap`: in the final implementation of the structure, the key of the HashMap has changed. The key has type `(usize, usize)`: the first member of the pair is, as before, the memory address of the head of the term, while the second is the number of argumnets that are being applied to the head; before this change, in fact, if a variable had 4 arguments applied, and 3 possible rewrite rules, we would check for a match $4 \times 3 = 12$ times. For example, if the rule `plus` is implemented as follows

```
def plus: Nat -> Nat -> Nat.
[ n ] plus zero n --> n
[ m, n ] plus (S m) n --> S (plus m n).
```

and we wanted to check if `plus zero (S zero)` matches a rewrite rule, we would have checked twice for `plus` and `plus zero`, and only in the end we would have matched `plus zero (S zero)` with `plus zero n --> n`, performing 5 match checks instead of 1. With this change, the performance improved drastically.

3.5.2 Inference for variables

As we have already said, Dedukti gives the possibility to the programmer to omit types in some circumstances. In particular, inference can happen in three circumstances:

- Local metavariables
- Wildcards
- Variables bound to abstractions (when possible)

As we have already introduced in Section 3.4, we adapted the code for checking sharing equality, adding an explicit substitution to cover first-order unification. The following snippet of code is extracted from the implementation of the `enqueue_and_propagate`.

```
1   fn enqueue_and_propagate(&self, m: &Rc<LNode>, c: &Rc<LNode>) ->
     ↪  Result<(), String> {
2     match (&**m, &**c) {
3       ...,
4       (BVar { subs_to, is_meta, .. }, _) => {
5         let sub = &mut *subs_to.borrow_mut();
6         if let Some(sub) = sub {
7           return self.enqueue_and_propagate(&sub, c);
8         } else if *is_meta {
9           *sub = Some(c.clone());
10        } else {
11          ...
12        }
13      (BVar { subs_to, is_meta, .. }, _) => {
14        let sub = &mut *subs_to.borrow_mut();
15        if let Some(sub) = sub {
16          return self.enqueue_and_propagate(m, &sub);
17        } else if *is_meta {
18          *sub = Some(m.clone());
19        } else {
20          ...
21        }
22      }
23      ...
24    }
25    ...
26  }
```

As it is possible to see, lines 9 and 18 are responsible for adding the explicit substitution. On the other hand, for implementing correctly bidirectional type checking and inferring, when possible, the type for variables, the following code is added in the `type_check` function. Lines 5 and 8 implement explicit type inference for variables.

```rust
fn type_check(term: &Rc<LNode>, typ_exp: &Rc<LNode>, rules: &RewriteMap)
↪  -> Result<()> {
    match &**term {
      ...
      LNode::Var { ty, .. } if ty.borrow().is_none() => {
        *ty.borrow_mut() = Some(typ_exp.clone());
      }
      LNode::BVar { ty, is_meta, .. } if ty.borrow().is_none() => {
        *ty.borrow_mut() = Some(typ_exp.clone());
      }
      ...
    }
}
```

3.5.3 Binding to the context

As we have already introduced in Section 3.2, in the context of the $\lambda\Pi$ calculus the sharing equivalence is not naively closed under the $\circlearrowleft$ relation. One such example is the following

```
succ: Nat -> Type.

def X: n1: Nat -> m1: succ n1 -> Nat := n2: Nat => m2: succ n2 => zero.
```

When checking the sharing equality between `n1, n2`, the varcheck would fail: in fact, we would find that the binder of `n1` is the product `n1: Nat -> ...`, while the binder of `n2` is the abstraction `n2: Nat => ...`. To solve this kind of problem we implemented the notion of **binding to the context**: when checking the type of an abstraction with a product, we unbind the variable of the product. Moreover, we substitute the variable bounded to the abstraction with the variable bounded to the product. For the example shown before, this would then result in checking the sharing equivalence $n1 \equiv n1$, and not $n1 \equiv n2$: with this implementation, sharing equivalence is closed under the $\circlearrowleft$ relation, hence the `var_check` procedure remains correct. Following is the code that implements the binding to the context.

```rust
fn type_check(term: &Rc<LNode>, typ_exp: &Rc<LNode>, rules: &RewriteMap)
↪  -> Result<()> {
  match &**term {
    LNode::Abs { bvar: lbvar, body: lbody, .. } => {
      let typ_exp = weak_head(&typ_exp, rules);
      if let LNode::Prod { bvar: pbvar, body: pbody, .. } = &*typ_exp {
        ...
        pbvar.bind_to_context(); // removes binder
        lbvar.subs_to(&pbvar);

        type_check(&lbody, &pbody, rules)?;
        ...
      }
      ...
    }
    ...
  }
}
```

3.5.4 Computation of Strong Normal Form

For lack of time, the strong normal form of a term is currently computed inefficiently: once the strong normal form is computed for an instantiated variable, it is saved into a field of the and it can be reused. This implementation can be very inefficient, as it leads to computing and expanding terms even if it is not useful. By useful we mean that the reduced form of the term is not a β-redex. For example, in the following reduction

$$yxx\,[x \leftarrow t] \longrightarrow^{\beta} ytt$$

the resulting term ytt is not a β-redex, hence it is not useful to expand the term. On the other hand, a useful scenario would be the following

$$xN\,[x \leftarrow \lambda z.M] \longrightarrow^{\beta} (\lambda z.M)N$$

In this case, $(\lambda z.M)N$ is a β-redex, hence it can be useful to expand the term.
This results in big normal forms, impacting indirectly also on the sharing equality algorithm, as will be shown in Section 4.2. The following code shows the implementation of the strong normal form for a generic term.

```
pub fn snf(term: &Rc<LNode>, rules: &RewriteMap) -> Rc<LNode> {
  let term = weak_head(term, rules);
    match &*term {
      LNode::Prod { bvar, body, .. } => {
          let bvar = snf(bvar, rules);
          let body = snf(body, rules);
          LNode::new_prod(bvar, body)
      }
      LNode::Abs { bvar, body, .. } => {
          let bvar = snf(bvar, rules);
          let body = snf(body, rules);
          LNode::new_abs(bvar, body)
      }
      LNode::App { left, right, .. } => {
          let left = snf(left, rules);
          let right = snf(right, rules);
          LNode::new_app(left, right)
      }
      LNode::BVar {
          subs_to,
          normal_forms,
          ..
      } if subs_to.borrow().is_some() => {
          let subs_to = subs_to.borrow().clone().unwrap();
          let NormalForms(wnf_computed, computed_snf) =
           ↪  normal_forms.borrow().clone();
          computed_snf.unwrap_or_else(|| {
              let snf_term = snf(&subs_to, rules);
              *normal_forms.borrow_mut() = NormalForms(wnf_computed,
               ↪  Some(snf_term.clone()));

              snf_term
          })
      }
      LNode::BVar { ty, .. } => {
          let ty_b = ty.borrow().clone();
          let ty_b = ty_b.map(|ty| snf(&ty, rules));
          *ty.borrow_mut() = ty_b;

          term
      }
      _ => term,
  }
}
```

3.5.5 Deep Cloning variables

One implementative choice we have taken consists in treating rewrite rules defined in one line such as

```
def one: Nat := S zero.
```

the same as

```
def one: Nat.
[ ] one --> S zero.
```

Deep cloning this kind of terms can be very inefficient, as we would compute and expand many times the same rule. An improvement could be to treat these variables as constant variables with a substitution, in order to always use it and memorize the normal forms for future re-usability.

Chapter 4

Conclusions

In this section we will summarize the work and give the results obtained. Section 4.1 presents the summary of the work done for this thesis. Section 4.2 will show the performance of `lapis-rs`, and will briefly investigate some interesting key points on which we could focus for future works. Section 4.3 will show some of the ideas we had for improving the performance and compatibility of `lapis-rs` with Dedukti.

4.1 Summary

In this section, I will summarize the work I have done in terms of time and implementation effort. This work started in July 2023, when together with professor Claudio Sacerdoti Coen, we planned the implementation of this tool. In July and August, I deeply researched for bibliographic references, understanding the topics found and planning the workflow for implementing `lapis-rs`. In the months of September, October and November, I implemented the code for `lapis-rs`, while reading and researching new topics for future works and efficient implementations of the typechecker. The implementation proved to be a hard task: understanding some mechanisms of Rust and tweaking some efficiency parameters for the typechecker took a lot of time. In the end, the current implementation of `lapis-rs` counts 2145 lines of code that also include the implemented tests. As a comparison `kontroli` counts 2237 lines of code without `dedukti-parse` module, while `Dedukti` counts 4385 lines of code that also include the parser.

4.2 Results

The benchmark has been done on a subset of a small library called `matita_basics_logic.dk`. We had efficiency issues on such a small library, so we haven't tried to test `lapis-rs` on bigger libraries. Moreover, as we will explain in Section 4.3, there are still some portions of Dedukti that `lapis-rs` does not recognize, so we think it is best to postpone the comparison of `lapis-rs` with the existing tools at a later time. Even though the system is inefficient, it shows some important properties of the sharing equality algorithm extended to the λΠ-calculus modulo context.

Figures 4.1 and 4.2 show that the time for computing the sharing equivalence is, in fact,

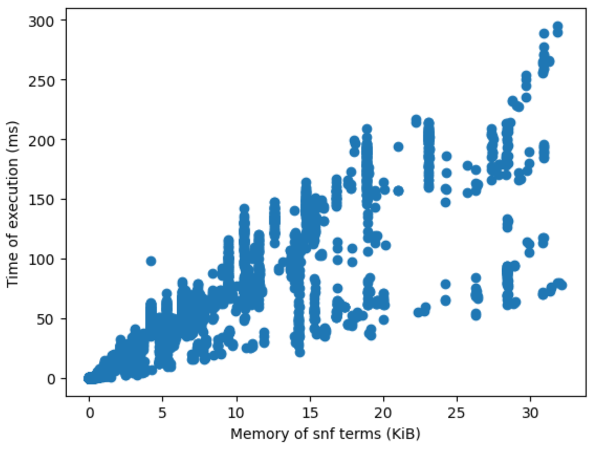


Figure 4.1: Time of execution of sharing equality algorithm in function of the size of the terms in strong normal form

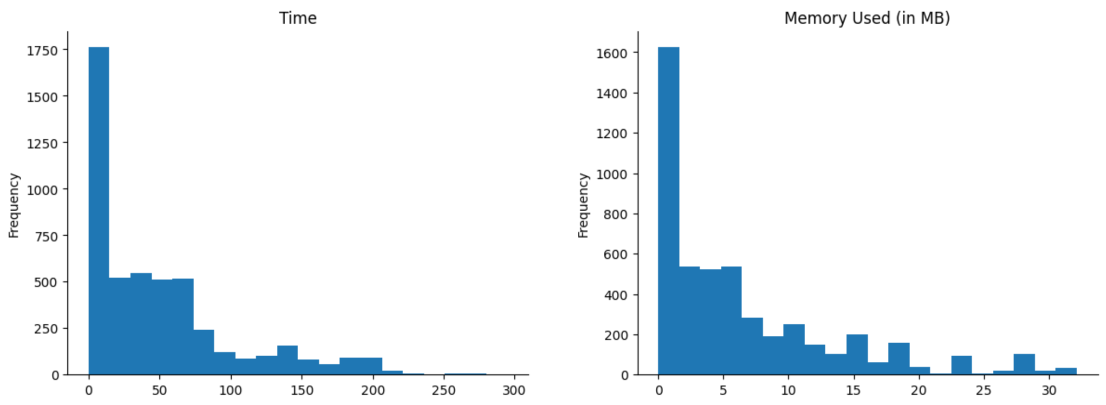


Figure 4.2: Frequencies of time executions (a) and memory usage (b)

linear w.r.t. the size of the term in strong normal form. We further investigated why the algorithm is inefficient, and we've found that the computed strong normal forms can be very big (Figure 4.3), even though the computing times are very efficient. It would be interesting to further investigating the reasons behind the high size of the terms in strong normal form. As we already shown in Section 3.5, the computation of the normal form of the terms is not linear, so this is sure to play an important role.

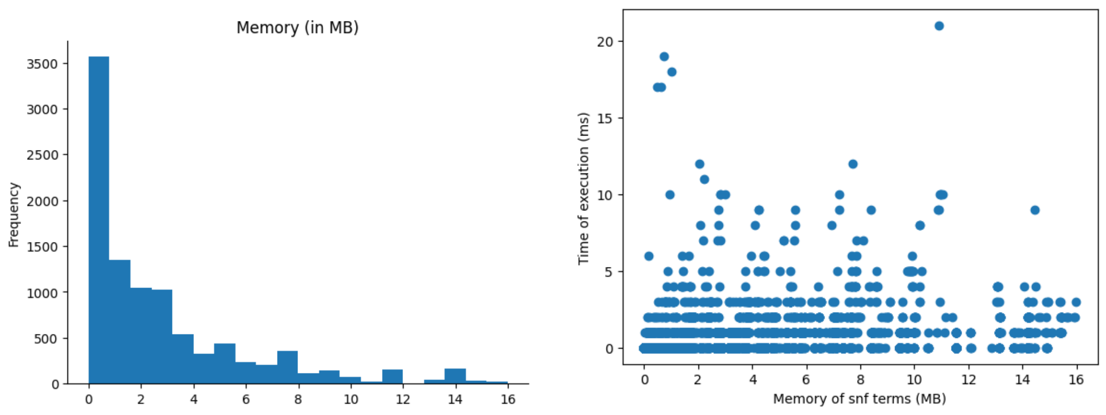


Figure 4.3: Frequencies of size for terms in strong normal form (a) and time execution for computing it (b)

File	Dedukti	Lapis	Lapis*
matita_basics_logic.dk	0.02s	269.50s	0.65s

Table 4.1: Comparison for time execution between Dedukti and Lapis. Lapis* is Lapis without the check for sharing equivalence.

4.3 Future Works

In this section we will present some of the possible future works on `lapis-rs`. There are many ideas and algorithms that can be implemented to improve the efficiency of `lapis-rs`, and it could be interesting, in future, to be able to effectively compare this tool with `dedukti` and `kontroli`.

4.3.1 Further investigations

As it is clear from the obtained results, there is much room for improving `lapis-rs`. One future work could consist in investigating which components of the system make it inefficient. As it is shown in Figure 4.4 the most expensive operation is the check for sharing equality: nonetheless, even disabling the check, the system is slow if compared with `dedukti`. Table 4.1 shows the times of execution, and we can clearly see that, even if `dedukti` executes the check for alpha equivalence, they are still faster than `lapis-rs` with the check for sharing equivalence disabled.

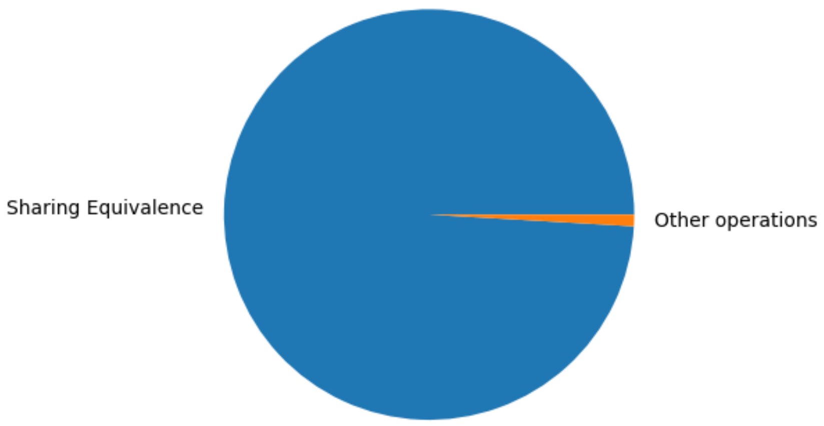


Figure 4.4: Time of execution of Sharing Equality over total time execution

4.3.2 Crumbling Abstract Machines

As described in Section 3.5, the algorithm for computing the strong normal form of a term implemented is the highly inefficient. One way it could be better implemented consists in computing the weak head of the term only when it is needed, as it is the main approach used in literature. Moreover, in [7] has been described an interesting algorithm to compute the strong normal form of a term, that is compatible with the structure of term graphs. It would be interesting to implement abstract crumbling machines and compare the results obtained with the ones we have now. It could also be interesting to further study the phenomenon of implosive sharing described in [10].

4.3.3 Higher-order patterns

During the implementation of the system, we have found out that Dedukti uses higher order patterns in rewrite rules. A higher order pattern consists in the possibility to apply arguments to a meta-variable. The following example in Dedukti shows a real case we have incurred in.

```
[s1 : Sort, s2 : Sort, a : Univ s1, b : (Term s1 a -> Univ s2)]
Term _ (prod s1 s2 a b) --> x : Term s1 a -> Term s2 (b x).
```

Here `b` is a meta-variable, but it also appears as the head of an application (`b x`). This kind of pattern has not been covered by our work, and many rules used this rewrite rule specifically. We have encountered some problems with checking sharing equivalence for these terms, so it is clear that it would be much more interesting to implement the possibility handle these patterns, or at least those occurring in Miller's pattern fragment [1].

Bibliography

[1] Dale Miller. "A logic programming language with lambda-abstraction, function variables, and simple unification". In: *Journal of logic and computation* 1.4 (1991), pp. 497–536.

[2] Thierry Coquand. "An algorithm for type-checking dependent types". In: *Science of Computer Programming* 26.1-3 (1996), pp. 167–177.

[3] Adam Chlipala, Leaf Petersen, and Robert Harper. "Strict bidirectional type checking". In: *Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation.* 2005, pp. 71–78.

[4] Mathieu Boespflug, Quentin Carbonneaux, and Olivier Hermant. "The $\lambda\Pi$-calculus Modulo as a Universal Proof Language." In: *PxTP.* 2012, pp. 28–43.

[5] The Univalent Foundations Program. "Homotopy Type Theory: Univalent Foundations of Mathematics". In: Institute for Advanced Study: `https://homotopytypetheory.org/book`, 2013. Chap. Type theory, pp. 17–34.

[6] Ali Assaf et al. "Dedukti: a logical framework based on the $\lambda\Pi$-calculus modulo theory". In: *Manuscript http://www. lsv. fr/˜ dowek/Publi/expressing. pdf* (2016).

[7] Beniamino Accattoli et al. "Crumbling Abstract Machines". In: *CoRR* abs/1907.06057 (2019). arXiv: `1907.06057`. URL: `http://arxiv.org/abs/1907.06057`.

[8] Andrea Condoluci, Beniamino Accattoli, and Claudio Sacerdoti Coen. "Sharing Equality is Linear". In: *CoRR* abs/1907.06101 (2019). arXiv: `1907.06101`. URL: `http://arxiv.org/abs/1907.06101`.

[9] Jana Dunfield and Neel Krishnaswami. "Bidirectional Typing". In: *CoRR* abs/1908.05839 (2019). arXiv: `1908.05839`. URL: `http://arxiv.org/abs/1908.05839`.

[10] Beniamino Accattoli, Andrea Condoluci, and Claudio Sacerdoti Coen. "Strong Call-by-Value is Reasonable, Implosively". In: *CoRR* abs/2102.06928 (2021). arXiv: `2102.06928`. URL: `https://arxiv.org/abs/2102.06928`.

[11] Michael Färber. “Terms for Efficient Proof Checking and Parsing”. In: *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2023. Boston, MA, USA: Association for Computing Machinery, 2023, pp. 135–147. ISBN: 9798400700262. DOI: `10.1145/3573105.3575686`. URL: `https://doi.org/10.1145/3573105.3575686`.

Ringraziamenti

Un profondo ringraziamento va al mio relatore, il professore Claudio Sacerdoti Coen. Senza la sua guida, non sarei riuscito a ideare e portare a termine questo lavoro, nè ad appassionarmi a tal punto a questo ambito di ricerca, che costantemente mi mette alla prova, portandomi a dare il meglio di me.
Un altro grande ringraziamento va ai miei affetti più cari. Sabrina, che con la tua gioiosa presenza hai sempre reso le mie giornate più leggere nonostante la distanza: la tua stima ed il tuo profondo affetto mi hanno motivato ogni giorno a mettermi in gioco, e sono state tesoro prezioso di questo percorso. Alla mia famiglia, che mi ha sempre sostenuto e che da sempre mi ha insegnato quanto sia importante l'educazione come principio fondamentale nella vita di una persona.
Un altro ringraziamento va ai miei compagni di viaggio: Bianca, Filippo, Alfonso, Isabella e Gabriele. Abbiamo condiviso studio, serate e tante risate. Senza un ambiente così amichevole, leggero e stimolante, questa esperienza non sarebbe stata la stessa. Custodisco ogni momento passato insieme come memoria indelebile di questo percorso formativo e soprattutto personale.
Infine, un grande ringraziamento va ai miei amici di Palermo, troppo numerosi per essere nominati uno ad uno: mi avete sempre dimostrato la vostra vicinanza, la vostra stima ed il vostro affetto; riuscirci a ritrovare per un caffè o per una sera, in quei brevi intervalli da questo percorso, mi hanno dato sempre più forza e voglia di dare tutto me stesso.