GIUSEPPE LOMURNO

VERIFIED GRAPH REWRITING

Dependent types in general programming

UNIVERSITÀ DI PISA
Dipartimento di Informatica
Corso di laurea triennale in Informatica

VERIFIED GRAPH REWRITING

Dependent types in general programming

Author:
GIUSEPPE LOMURNO

Supervisor:
PROF. FABIO GADDUCCI

Examiner:
PROF. ANDREA CORRADINI

Academic Year 2018–2019

ABSTRACT

We present REWRITE, a proof-of-concept application for directed graph rewriting, implemented with Idris, a language with support for dependent types, as a means to experience and motivate the use of dependent types in a non-trivial general programming context, other than theorem proving. We describe the process of gradually developing with dependent types, what are the pros and cons, and the future work.

SOMMARIO

Presentiamo REWRITE, una applicazione proof-of-concept per la riscrittura di grafi diretti, implementata in Idris, un linguaggio con supporto ai tipi dipendenti, come mezzo per fare esperienza e motivare l'utilizzo dei tipi dipendenti in contesti di programmazione generali non banali, oltre la verifica di teoremi. Descriviamo il processo graduale di sviluppo con i tipi dipendenti, quali sono i pro e i contro, e il lavoro futuro.

CONTENTS

LIST OF FIGURES

LISTINGS

1 INTRODUCTION

One of the most difficult part of software development is reasoning about programs and testing. Empirical testing requires extensive coding times and reasoning about edge cases. Static tools, covering analysis and property testing alleviate this burden but, often, it is not enough. Static typing, algebraic data types, ad-hoc and parametric polymorphism, generics, prevent a whole bunch of errors due to incompatible types and operations, however, it is difficult to incorporate more complex invariants, as we strive to make it impossible to represent invalid data, without introducing run-time checked preconditions and postconditions, particularly when developing libraries.

Another approach, to be used alongside regular testing, is the usage of formal methods, specification techniques and verification models, such as TLA$^+$ or Petri nets. However, this approach requires the same problem to be modeled both in the programming implementation and in the formal specification. Automatic extraction from a formal specification is a way of alleviating this problem. Extraction is not exempt from difficulties, prototyping and incremental development require a lot of effort to obtain the correct specification and not always the produced code is particularly readable for humans.

In this thesis, we explore the use of a more powerful type system based on dependent types, aiming at using it both for computation and verification. Due to the correspondence between dependent type theories and first-order intuitionistic logic, we can express, verify and incapsulate complex properties of custom data types directly in code, given that we are respecting some restrictions on the functions we are defining. Dependent type systems have the characteristic of allowing types indexed or parameterized not only by other types, as in higher kinded types, but also by instances of types, terms.

Extensive literature is available on programming languages, such as Coq or Agda, as proof assistants, instead we want to focus more on regular programming and using the theorem proving capabilities as a way of enforcing invariants in our data structures. For this purpose, we need to find a problem of simple formulation, of immediate understanding, but with a complex characterization of the structure of the terms. A problem with an extended mathematical description, an algebraic approach, may be preferable because, in general, the translation process from a mathematical representation to a computational representation is complex.

Therefore, a simple case study, but very relevant in computer science is on graph rewriting. A graph transformation, or rewriting, is a pro-

cedure that given a graph with some kind of substructure, a property, it builds a new graph in accordance with a chosen rule. The concept of graphs, in particular simple directed graphs, is simple however, multiple, distinctly different, representation of graphs can be chosen, from representations more akin to computational needs like adjacency lists or matrices, to algebraic approach such as sets with relations to represent edges, or function-based approach more familiar with category theorist. In order to define a rewrite rule we need a starting matching subgraph and a new graph to replace it with, however how we can preserve and verify certain properties of the original graph is non trivial. In this thesis, we work with multiple algebraic approaches taken from the categorical generalization of graph rewriting, although the category theory details are left to the reference material, as it is not the scope of this work. We want to discover if dependent types, in some cases, can simplify the implementation details, if we can express the complexity of algebraic approaches, if properties can be easily translated in code without having to deal with a huge amount of error checking code, and if regular non-dependent functional programming code can be mixed with dependent code, as a way of progressively augment the functionality of existing implementation and which architectural decisions are affected by dependent types.

The work in this thesis is organized in four main chapters:

CHAPTER 2 We introduce a formal system to describe basic concepts in dependent type theories, with comparison to more familiar, simpler, models such as the simply-typed lambda calculus.

CHAPTER 3 We introduce Idris, the implementation language, and a walkthrough, with a number of examples, on how to use dependent functionalities from regular algebraic data types to dependent types, proofs, and views.

CHAPTER 4 We describe the necessities and the rationale behind the program architecture. Each fundamental component is extensively described and we show how the functionalities portrayed in the previous chapter are adopted in the implementation.

CHAPTER 5 The conclusions argue the pros and cons of dependent types in the light of this case study, what would have been done differently and the future work on this project.

APPENDIX A Multiple rewrite rules are implemented in this work, we give a more complete overview of the rules, what are their components and what properties are checked in the implementation.

The documented source code, with some test cases, is available in a versioned repository at `https://github.com/ShinKage/rewrite`.

2 DEPENDENT TYPES

The origin and use of dependent type theories for software validation traces back to the discovery of the propositions-as-types principle by the mathematician and logician Haskell Curry. In 1958 [Cur34; CF58], working on intuitionistic and combinatorial logics, he found out that a certain kind of deduction system has a correspondence to a combinatorial model of computation, proposed by Curry himself and Moses Schönfinkel. Following his work, in 1969, although published in 1980, William Howard [How80] found the same kind of correspondence between the natural deduction proof system in intuitionistic logic and the simply-typed lambda calculus. This correspondence between a logic proof system and a model of computation would be known as the Curry-Howard isomorphism or correspondence [SU06].

These important results gave stimuli to new formal systems that would bridge the gap between computation and verification. Some of these models are the Martin-Löf [MLS84] predicative extensional type theory, the Calculus of Constructions by Thierry Coquand [CH86] and the more recent Homotopy Type Theory [Uni13], which is an intensional type theory. Various programming languages were born from these theories in the subsequent years, some more focused on the theorem proving aspect, most notably the Coq programming language, some other more focused on general programming, like Agda, which is heavily inspired by the Haskell programming language. Many Haskell programmers, as well as in other functional languages, are trying to incorporate dependent types in a more productivity-centered context. The work of this thesis is written in the Idris programming language [Bra13]. Like Agda, Idris is more focused on general programming and it is inspired, particularly in syntax, by Haskell.

Although the Curry-Howard isomorphism lets us interpret some types as propositions, there are some pitfalls. General recursion, necessary for Turing-completeness, makes the theory inconsistent, so calculations must halt and, as Alan Turing proved, the Halting problem is undecidable. Idris, like many other languages, provides a conservative termination checker which checks that each recursive call has structurally smaller arguments, eventually allowing only provably terminating calculations or false-negatives. Due to its general programming pedigree, Idris has opt-in termination checking, nonetheless, only provably terminating functions are allowed for type-level computations. Another way of dealing with some kind of general

recursion is corecursion or to use some type of monad to segregate the provably terminating pieces of code.

2.1 A FORMAL SYSTEM

Following the work on the Martin-Löf type theory of [MLS84; NPS90], the comprehensive introduction and semantics provided by [Hof97] and the implementations of [Nor07; Alt+10; Bra13], we provide a simplified formal system for the core concepts of dependent typing.

To define a formal system for dependent types we need to introduce the concept of context. Some notation first: uppercase greek letters, $\Gamma, \Delta, \ldots$, will be used for contexts; lowercase greek letters, $\sigma, \tau, \ldots$, for types; uppercase latin letters, $M, N, \ldots$, will be used for terms; lowercase latin letters, $x, y, \ldots$, for variables;

Definition 2.1 (Context)**.** A context is a list of type declarations

$$x_1 : \sigma_1, x_2 : \sigma_2, \ldots, x_n : \sigma_n$$

A context is well-formed if each σ_i is a type in the context

$$x_1 : \sigma_1, x_2 : \sigma_2, \ldots, x_{i-1} : \sigma_{i-1}$$

and the $x_i : \sigma_i$ are pairwise distinct.

There are six kinds of judgments that will be used in the typing rules:

Definition 2.2 (Fundamental judgments)**.**

$\vdash \Gamma$ context	Γ is a well-formed context
$\Gamma \vdash \sigma$ type	σ is a type in Γ
$\Gamma \vdash M : \sigma$	M is a term of type σ in Γ
$\vdash \Gamma = \Delta$ context	Γ and Δ are definitionally equal contexts
$\Gamma \vdash \sigma = \tau$ type	σ and τ are definitionally equal types in Γ
$\Gamma \vdash M = N : \sigma$	M, N are definitionally equal terms of type σ in Γ

Rule 1 (Context formation)**.**

$$\frac{}{\vdash \diamond \text{ context}} \qquad \frac{\Gamma \vdash \sigma \text{ type}}{\vdash \Gamma, x : \sigma \text{ context}}$$

$$\frac{\vdash \Gamma = \Delta \text{ context} \qquad \Gamma \vdash \sigma = \tau \text{ type}}{\vdash \Gamma, x : \sigma = \Delta, y : \tau \text{ context}}$$

The $\diamond$ symbol indicates an empty context and the variables x and y in these rules must undefined so far. The last rule introduces equality for context extension, if we have two definitionally equal contexts Γ, Δ and two definitionally equal types σ, τ, introducing a fresh variable $x : \sigma$

in the context Γ and introducing a differently named fresh variable $y : \tau$ in the context Δ produces definitionally equal contexts.

Rule 2 (Variables)**.**

$$\frac{\vdash \Gamma, x : \sigma, \Delta \text{ context}}{\Gamma, x : \sigma, \Delta \vdash x : \sigma}$$

Definitional equality must be an equivalence relation, reflexivity, symmetry, and transitivity follows from further judgements, simple enough to be omitted.

Rule 3 (Relation of typing and definitional equality)**.**

$$\frac{\Gamma \vdash M : \sigma \qquad \vdash \Gamma = \Delta \text{ context} \qquad \Gamma \vdash \sigma = \tau \text{ type}}{\Delta \vdash M : \tau}$$

$$\frac{\vdash \Gamma = \Delta \text{ context} \qquad \Gamma \vdash \sigma \text{ type}}{\Delta \vdash \sigma \text{ type}}$$

We also introduce a substructural rule for context weakening and substitution rules, where $\mathcal{J}$ ranges over one of four judgments, $M : \sigma$, σ type, $M = N : \sigma$, $\sigma = \tau$ type:

Rule 4 (Context weakening)**.**

$$\frac{\Gamma, \Delta \vdash \mathcal{J} \qquad \Gamma \vdash \rho \text{ type} \qquad x \text{ unbound}}{\Gamma, x : \rho, \Delta \vdash \mathcal{J}}$$

Rule 5 (Substitution)**.**

$$\frac{\Gamma, x : \rho, \Delta \vdash \mathcal{J} \qquad \Gamma \vdash U : \rho}{\Gamma, \Delta[x \coloneqq U] \vdash \mathcal{J}[x \coloneqq U]}$$

2.2 Π-types

In the simply-typed lambda calculus, for each pair of types σ and τ in a context Γ, it follows that, in the same context, it exists the type of functions from terms of type σ to terms of type τ:

Rule (Simply-typed lambda calculus function space)**.**

$$\frac{\Gamma \vdash \sigma \text{ type} \qquad \Gamma \vdash \tau \text{ type}}{\Gamma \vdash \sigma \to \tau \text{ type}}$$

A dependent function space follows from two types σ and τ, as for a simply-typed lambda calculus function space, however the latter can be indexed by a term of the former:

Rule 6 (Dependent function space)**.**

$$\frac{\Gamma \vdash \sigma \text{ type} \qquad \Gamma, x : \sigma \vdash \tau[x] \text{ type}}{\Gamma \vdash \Pi x\colon \sigma.\, \tau[x] \text{ type}}$$

$$\frac{\Gamma \vdash \sigma_1 = \sigma_2 \text{ type} \qquad \Gamma, x : \sigma \vdash \tau_1[x] = \tau_2[x] \text{ type}}{\Gamma \vdash \Pi x\colon \sigma_1.\, \tau_1[x] = \Pi x\colon \sigma_2.\, \tau_2[x] \text{ type}}$$

These two function spaces are similar, and with the help of the Rule 4 we easily demonstrate that the dependent function space is a generalization of regular function spaces and consequently the latter can be defined in terms of the former, $\sigma \to \tau \coloneqq \Pi x\colon \sigma.\, \tau$:

Rule 7 (Non-dependent function space)**.**

$$\frac{\Gamma \vdash \sigma \text{ type} \qquad \dfrac{\Gamma \vdash \sigma \text{ type} \qquad \Gamma \vdash \tau \text{ type}}{\Gamma, x : \sigma \vdash \tau \text{ type}}\ \text{Weakening}}{\Gamma \vdash \Pi x\colon \sigma.\, \tau}$$

Dependent functions correspond to the set-theoretic concept of cartesian product over a family of sets. If we define $I_n = \{1, 2, \ldots, n\}$ as a set of indices, tuples can be defined as a function from I_n that maps its value at i to the i-th element of the tuple, so the cartesian product, $X_1 \times X_2 \times \cdots \times X_n$, is the set of functions:

$$\{\, f\colon I_n \to X_1 \cup X_2 \cup \cdots \cup X_n \mid \forall i \in I_n.\, f(i) \in X_i \,\}$$

In the same way that the index of a dependent type determines the terms that are instances of that type, the element of the index set determines uniquely the codomain of the functions.

Dependent function spaces are isomorphic to first-order universal quantifications. If we have a predicate τ that we would like to prove on all the elements of a set σ, i. e. $\forall x \in \sigma.\, \tau(x)$, we need to prove that the predicate τ is satisfiable for each $x \in \sigma$. In the condition of the Curry-Howard correspondence, types are propositions and terms are proofs. If we have a type σ and a type τ dependent on σ, the latter can be interpreted as a predicate over σ and the type $\Pi x\colon \sigma.\, \tau[x]$ is instanced by terms that build a term of $\tau[x]$ from a term of σ. The terms of Π construct a proof of $\tau[x]$ from a proof of σ, which is exactly what is needed for universal quantification.

Forming a canonical element of type $\Pi x\colon \sigma.\, \tau[x]$ is pretty much the same as defining a regular lambda function in the simply-typed lambda calculus as can be seen from the rule:

Rule (Non-dependent function forming)**.**

$$\frac{\Gamma \vdash \sigma \text{ type} \qquad \Gamma \vdash \tau \text{ type} \qquad \Gamma \vdash M : \tau}{\Gamma \vdash \lambda x\colon \sigma.\, M : \sigma \to \tau}$$

We also need to define definitional equality judgements for canonical terms:

Rule 8 (Dependent function forming)**.**

$$\frac{\Gamma \vdash \sigma \text{ type} \qquad \Gamma, x : \sigma \vdash \tau[x] \text{ type} \qquad \Gamma, x : \sigma \vdash M : \tau[x]}{\Gamma \vdash \lambda x\colon \sigma.\, M : \Pi x\colon \sigma.\, \tau[x]}$$

$$\frac{\begin{array}{cc} & \Gamma, x : \sigma \vdash M_1 : \tau_1 \\ \Gamma \vdash \sigma_1 = \sigma_2 \text{ type} & \Gamma, x : \sigma \vdash M_2 : \tau_2 \\ \Gamma, x : \sigma \vdash \tau_1[x] = \tau_2[x] \text{ type} & \Gamma, x : \sigma \vdash M_1 = M_2 : \tau_1[x] \end{array}}{\Gamma \vdash \lambda x\colon \sigma_1.\, M_1 = \lambda x\colon \sigma_2.\, M_2 : \Pi x\colon \sigma_1.\, \tau_1[x]}$$

Rule 9 (Dependent function elimination)**.**

$$\frac{\Gamma \vdash M : \Pi x\colon \sigma.\, \tau[x] \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash \mathrm{App}(M, N) : \tau[N]}$$

$$\frac{\Gamma \vdash \lambda x\colon \sigma.\, M : \Pi x\colon \sigma.\, \tau[x] \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash \mathrm{App}(\lambda x\colon \sigma.\, M, N) = M[x \coloneqq N] : \tau[N]}$$

The application operator takes a dependent function and a term, and the dependent return type has that term bound as index. The same term is capture-free substituted to the body of the lambda function as represented by the $[x \coloneqq N]$ substitution. Regular function application does not affect the result type and it uses only the capture-free substitution on the argument term.

2.3 Σ-TYPES

We can extend the simply-typed lambda calculus with pairs, also called product types.

Rule (Product type)**.**

$$\frac{\Gamma \vdash \sigma \text{ type} \qquad \Gamma \vdash \tau \text{ type}}{\Gamma \vdash \sigma \times \tau \text{ type}}$$

With dependent types, we can generalize this concept and build a pair that holds a term M of type σ and a term of type $\tau[M]$, where τ is dependent on σ.

Rule 10 (Dependent sum)**.**

$$\frac{\Gamma \vdash \sigma \text{ type} \qquad \Gamma, x : \sigma \vdash \tau[x] \text{ type}}{\Gamma \vdash \Sigma x\colon \sigma.\, \tau[x] \text{ type}} \qquad \frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash N : \tau[M]}{\Gamma \vdash \langle M, N \rangle : \Sigma x\colon \sigma.\, \tau[x]}$$

In the same way, we can implement non-dependent function spaces via Π-types, we can implement product types via Σ-types simply using a non-dependent type as a second argument, i. e. $\sigma \times \tau \coloneqq \Sigma x\colon \sigma.\, \tau$:

Rule 11 (Non-dependent product type)**.**

$$\frac{\Gamma \vdash \sigma \text{ type} \qquad \dfrac{\Gamma \vdash \sigma \text{ type} \qquad \Gamma \vdash \tau \text{ type}}{\Gamma, x : \sigma \vdash \tau \text{ type}} \text{ Weakening}}{\Gamma \vdash \Sigma x\colon \sigma.\, \tau}$$

Dependent sums correspond to disjoint unions in set theory. If we have a family of sets X_i indexed by elements i of a set I, the disjoint union is defined as all the possible pairs of indices and elements of the set with that index:

$$\biguplus_{i \in I} X_i \coloneqq \bigcup_{i \in I} \{\, \langle i, x \rangle \mid x \in X_i \,\}$$

Elements of disjoint unions are pairs index-element, just as terms of dependent sums are pairs built from an index of a dependent type and a term of such type.

Dependent sums are isomorphic to first-order existential quantifications. To obtain a constructive proof that a value in a set σ satisfies a property τ, i. e. $\exists x \in \sigma.\, \tau(x)$, we need at least one of such values. As for dependent functions spaces, we can interpret $\langle M, N \rangle : \Sigma x\colon \sigma.\, \tau[x]$ as a pair where the first parameter is a witness of a proof and the second parameter, the dependent one, is the proof itself that the witness satisfies the predicate.

Rule 12 (Dependent sum elimination)**.**

$$\frac{\Gamma \vdash M : \Sigma x\colon \sigma.\, \tau[x] \quad \dfrac{\Gamma, z : \Sigma x\colon \sigma.\, \tau[x] \vdash \rho[z] \text{ type} \qquad \Gamma, x : \sigma, y : \tau[x] \vdash H : \rho[\langle x, y \rangle]}{}}{\Gamma \vdash R^{\Sigma}(H, M) : \rho[M]}$$

$$\frac{\dfrac{\Gamma, z : \Sigma\colon \sigma.\, \tau[x] \vdash \rho[z] \text{ type} \qquad \dfrac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash N : \tau[M]}{}}{\Gamma, x : \sigma, y : \tau[x] \vdash H : \rho[\langle x, y \rangle] \qquad \Gamma \vdash R^{\Sigma}(H, \langle M, N \rangle) : \rho[\langle M, N \rangle]}}{\Gamma \vdash R^{\Sigma}(H, \langle M, N \rangle) = H[x \coloneqq M, y \coloneqq N] : \rho[\langle M, N \rangle]}$$

The R^{Σ} eliminator takes two arguments: the first, H, dictates how we handle the dependent sum; the second is the dependent sum itself. H can depend on both types carried by the dependent sum. As an example, to grasp an intuition on how the R^{Σ} eliminator works, we define both projections. In the following rules, we assume the following judgements: $\Gamma \vdash \sigma$, $\Gamma, x : \sigma \vdash \tau[x]$ and $\Gamma \vdash M = \langle x, y \rangle : \Sigma x\colon \sigma.\, \tau[x]$.

Rule 13 (Dependent sum projections)**.**

$$\frac{\Gamma, z : \Sigma x\colon \sigma.\, \tau[x] \vdash \rho[z] \text{ type} \qquad \Gamma, z : \Sigma x\colon \sigma.\, \tau[x] \vdash \rho[z] = \sigma \text{ type}}{\Gamma \vdash R^{\Sigma}(x, M) : \sigma}\ \text{M.1}$$

$$\frac{\Gamma, z : \Sigma x\colon \sigma.\, \tau[x] \vdash \rho[z] \text{ type} \qquad \Gamma, z : \Sigma x\colon \sigma.\, \tau[x] \vdash \rho[z] = \tau[x] \text{ type}}{\Gamma \vdash R^{\Sigma}(y, M) : \tau[R^{\Sigma}(x, M)]}\ \text{M.2}$$

The first projection M.1 returns the witness of the dependent sum, defined as the variable x in the assumptions. The second projection M.2 returns the predicate on the witness defined as the variable y, notably the return type is indexed by the result of the first projection, i. e. the witness.

2.4 OTHER RELEVANT TYPES

Although definitional equality is incorporated in this formal system, it is expressed only as judgments and, thus, is not available as a type. We can build, in this context, a type for definitional equality, also called propositional equality.

Rule 14 (Propositional equality type)**.**

$$\frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash \mathrm{Id}_{\sigma}(M, N) \text{ type}} \qquad \frac{\Gamma \vdash M : \sigma}{\Gamma \vdash \mathrm{Refl}_{\sigma}(M) : \mathrm{Id}_{\sigma}(M, M)}$$

For each pair of terms of the same type, we have an $\mathrm{Id}_{\sigma}(M, N)$ type, although, it is not always inhabited. With the Refl_{σ} constructor, we obtain a proof that the equality defined by the Id_{σ} type is, in fact, reflexive, however, equalities needs to be also symmetric and transitive, these properties are consequences of the elimination rule for identity types.

Rule 15 (Propositional equality elimination)**.**

$$\frac{\begin{array}{c}\Gamma \vdash \sigma \text{ type} \\ \Gamma \vdash M : \sigma \\ \Gamma \vdash N : \sigma \qquad\qquad \Gamma, z : \sigma \vdash H : \tau[z, z, \mathrm{Refl}_{\sigma}(z)] \\ \Gamma \vdash P : \mathrm{Id}_{\sigma}(M, N) \qquad \Gamma, x : \sigma, y : \sigma, p : \mathrm{Id}_{\sigma}(x, y) \vdash \tau[x, y, p] \text{ type}\end{array}}{\Gamma \vdash R^{\mathrm{Id}}(H, M, N, P) : \tau[M, N, P]}$$

$$\frac{\Gamma \vdash R^{\mathrm{Id}}(H, M, M, \mathrm{Refl}_{\sigma}(M)) : \tau[M, M, \mathrm{Refl}_{\sigma}(M)]}{\Gamma \vdash R^{\mathrm{Id}}(H, M, M, \mathrm{Refl}_{\sigma}(M)) = H[z \coloneqq M] : \tau[M, M, \mathrm{Refl}_{\sigma}(M)]}$$

To summarize, dependent type models are in direct correspondence with intuitionistic first-order logic. We described how dependent functions spaces correspond to universal quantification and how dependent sums correspond to existential quantification, however we need three more ingredients. The false formula can be interpreted as the data type without inhabitants, i. e. $\vdash \bot$ type, while the true formula can be interpreted as the data type with only one unique inhabitant, i. e. $\vdash \top$ type, $\vdash \mathrm{tt} : \top$. The last ingredient is implication which is in

First order logic	Dependent type model
Proposition	σ
Proof	$M : \sigma$
Predicate	$\tau[x]$
$\forall x \in \sigma.\, \tau$	$\Pi x\colon \sigma.\, \tau[x]$
$\exists x \in \sigma.\, \tau$	$\Sigma x\colon \sigma.\, \tau[x]$
$\sigma \Rightarrow \tau$	$\sigma \to \tau$
$\sigma \wedge \tau$	$\sigma \times \tau$
$\sigma \vee \tau$	$\sigma + \tau$
$\neg\sigma$	$\sigma \to \bot$
True	$\top$
False	$\bot$
$M = N$	$\mathrm{Id}_\sigma(M, N)$

Table 2.1: Correspondences between first order logic and a dependent type model.

correspondence with regular, non-dependent, function space. If we want to prove that $\sigma \Rightarrow \tau$ and have a proof of σ we need to construct a proof of τ. In regular function space, $\sigma \to \tau$, we transform a term of type σ, or in other terms a proof of σ, in a term of type τ. The other, classical, connectives of propositional calculus can be expressed in terms of only implications, as an example we can express $\neg\sigma$ as $\sigma \Rightarrow F$, and this translated to types becomes $\sigma \to \bot$. The addition of algebraic data types allows encoding conjunction and disjunction without the need for dependent functions spaces, in fact, regular product types are isomorphic to conjunction and regular sum types are isomorphic to disjunction.

3 IDRIS

3.1 LANGUAGE

Idris syntax is heavily inspired by the Haskell programming language with some differences. It is a strict evaluation language with opt-in laziness. We provide a short view of extensively used Idris features in this project, for a more detailed explanation of these and other features see [Bra13] and the language documentation available at http://docs.idris-lang.org/en/latest/.

Files are evaluated sequentially, as in OCaml, and a `mutual` block is provided for mutually recursive definitions. Pattern matching functionality is provided both at function level and expression level via the `case` expression, similarly to Haskell. Functions are implemented by pattern matching, however, since type inference is, in general, undecidable in a dependently typed context, top-level function must have a type declaration as shown in:

Listing 3.1: Function definition in Idris with pattern matching.

```
lefts : List (Either a b) -> List a
lefts [] = []
lefts (x :: xs) = case x of
  Left l  => l :: lefts xs
  Right r => lefts xs
```

Algebraic data types can be declared in two ways, the simpler syntax is identical to Haskell data type definition, where each constructor is separated by a pipe, building sum types, and each of them can have multiple parameters or indices, building product types:

Listing 3.2: Simple data type definition in Idris.

```
-- Definitions can be recursive
data Nat = Z | S Nat

-- Definition can have multiple paremeters
-- Either :: * -> * -> *
data Either a b = Left a | Right b
```

These definitions, however, are valid only if each constructor has the same return type. A syntax similar to Generalised Algebraic Data Types, available as a Haskell extension, offers more control over constructor definitions (from this point onward only this notation will be used in listings):

Listing 3.3: GADTs definition in Idris.

```
-- Simpler ADTs can be easily translated to GADTs
data Either : (a, b : Type) -> Type where
  Left  : a -> Either a b
  Right : b -> Either a b

-- GADTs allow differently indexed return types
data Elem : (x : a) -> (xs : List a) -> Type where
  Here  : Elem x (x :: xs)
  There : (later : Elem x xs) -> Elem x (y :: xs)
```

As can be seen from the `Elem` data type definition, the second index of the return type is different between the `Here` constructor and `There` constructor. Idris permits the omission of arguments which can be inferred, these are called implicit arguments and can be manually declared via curly braces. The `y` variable in the `There` constructor is not explicitly declared but it is automatically inferred to be `y : a`. By default, each lowercase parameter is an implicit argument unless explicitly prescribed. Implicit arguments can be used both in data type definition and function declaration, they can be explicitly passed as arguments and they can be pattern matched. Both declarations and the pattern matching of implicit parameters require curly braces instead of parenthesis.

Listing 3.4: Implicit arguments both in data types and functions. The `Fin n` data type represents the indices of a finite set with n elements; will be defined in Listing 3.17.

```
-- Implicit arguments in data type definition
data Elem : {a : Type} -> (x : a) -> (xs : List a) -> Type where
  Here  : {x : a} -> (xs : List a) -> Elem x (x :: xs)
  There : {x, y : a} -> {xs : List a} -> (later : Elem x xs)
       -> Elem x (y :: xs)

-- Implicit arguments in function declaration
index : {a : Type} -> {n : Nat} -> Fin n -> Vect n a -> a
index FZ     (x :: xs) = x
index (FS i) (x :: xs) = index i xs

-- Pattern matching an implicit argument and explicit usage
range : {len : Nat} -> Vect len (Fin len)
range {len = Z}   = []
range {len = S n} = FZ :: map FS range
```

Laziness must be explicitly annotated with some special constructors like `Lazy a` for lazy evaluation and `Inf a` for potentially infinitely recursive parameters. The compiler automatically wraps and unwraps data from and to these two constructors with no intervention from the developer, who can treat lazy parameters just like strict, default, parameters. Corecursive definitions use the `codata` keyword instead of

`data`. `codata` automatically wraps potentially infinite parameters with lazy `Inf` constructor, allowing infinitely recursive data structures that pass the termination checking as described in Chapter 2 introduction:

Listing 3.5: Corecursive data type definition and desugared version.

```
-- Corecursive definition
codata Stream : (a : Type) -> Type where
  (::) : a -> Stream a -> Stream a

-- Desugared to
data Stream : (a : Type) -> Type where
  (::) : a -> Inf (Stream a) -> Stream a
```

Types are first-class citizens and can be manipulated just like values. We can define functions that calculate types and use them both in type calculations and term calculations, and there is no explicit syntax for dependent and non-dependent function space, in fact, the latter can be always expressed in term of the former as shown in Section 2.2:

Listing 3.6: Definition of a function that calculates a type.

```
isFourtytwo : Nat -> Type
isFourtytwo n = if n == 42 then String else Bool

size : VSub xs -> Nat
extract : (xs : Vect n a) -> (sub : VSub xs) -> Vect (size sub) a
```

Dependent sums or pairs are implemented via the `DPair` data type:

Listing 3.7: Definition of dependent sums in Idris.

```
data DPair : (a : Type) -> (P : a -> Type) -> Type where
  MkDPair : (x : a) -> P x -> DPair a P
```

Since dependent sums are one of the fundamental constructs of a dependently typed language, Idris provide syntactic sugar for their manipulation: for example instead of writing `DPair Nat Even` we can write `(n : Nat ** Even n)` or `(n ** Even n)` if the type can be inferred, and at term level instead of writing `MkDPair 0 (EvenZ)` we can write `(0 ** EvenZ)`.

One of the most important features are typed holes. Introduced by a question mark, they signal an incomplete part of the implementation, however they are still valid code and can be queried by the interpreter, returning the type that the implementation must have, along with all the types in the context. Typed holes let us code incrementally and give a precious help telling developers what is needed to complete the implementations and what they have available in the context.

Listing 3.8: Typed hole and interpreter querying.

```
even : Nat -> Bool
even Z     = True
even (S k) = ?even_rhs

-- The interpreter can be used to query the holes
-- Idris> :t even_rhs
--   k : Nat
-- ------------------------------------
-- even_rhs : Bool
```

3.2 EXAMPLES

The classical way of defining a list in functional languages is to use an inductive definition parameterized, or indexed, by the type of the elements, no dependent types needed:

Listing 3.9: Definition of a inductive list data type.

```
data List : (elem : Type) -> Type where
  Nil  : List elem
  (::) : (x : elem) -> (xs : List elem) -> List elem
```

Lists are built via successive applications of inductive constructors, i. e. `1 :: 2 :: 3 :: Nil`. Idris provides syntactic sugar for list building, the previous example can be translated to `[1,2,3]`, more familiar to anyone who knows other languages like C or Java. This syntax requires a type with two constructors named exactly `Nil` and `(::)`, the former empty, the latter a binary constructor, recursive on the second argument. This definition is identical to implementations in other functional languages. The extraction of the first element is trivial via pattern matching, and even more complex functions, like list fusion or length determination, are of simple implementation:

Listing 3.10: Operations on inductive lists.

```
(++) : List a -> List a -> List a
(++) []        ys = ys
(++) (x :: xs) ys = x :: (xs ++ ys)

length : List a -> Nat
length []        = Z
length (x :: xs) = S (length xs)
```

However, not having a notion of length at compile time, we cannot assure that given an index, a given list is sufficiently long to contain a value at that index. The regular way of dealing with dynamic results is using some kind of type to differentiate between a valid and invalid result and the simpler approach is to implement such an index func-

tion using the `Maybe` monad and using natural numbers as defined in Listing 3.2:

Listing 3.11: Regular indexing function on lists.

```
index' : Nat -> List elem -> Maybe elem
index' Z     (x :: xs) = Just x
index' (S n) (x :: xs) = index' n xs
index' n     []        = Nothing
```

This implementation defers the verification at runtime and the return value must be pattern matched to check the result, even if we determined beforehand that `n` is less than the length of the list. To assure the compiler that we are in bounds we need a type, a proposition in the Curry-Howard isomorphism, dependent on a specific value of a `Nat` and a specific value of a `List elem`, and this is only possible with dependent types. We can see both the Idris definition and a possible translation in the system we described in the previous chapter[1]:

Listing 3.12: In bound proposition for list indexing.

```
data InBounds : (n : Nat) -> (xs : List elem) -> Type where
  InFirst : InBounds Z (x :: xs)
  InLater : InBounds n xs -> InBounds (S n) (x :: xs)
```

Rule 16 (`InBounds` formal translation.)**.**

$$\frac{\Gamma \vdash \texttt{Type}\ \text{type} \qquad \Gamma \vdash \texttt{Nat}\ \text{type} \qquad \Gamma, a : \texttt{Type} \vdash \texttt{List}[a]\ \text{type}}{\Gamma, a : \texttt{Type}, n : \texttt{Nat}, xs : \texttt{List}[a] \vdash \texttt{InBounds}[n, xs]\ \text{type}}$$

$$\frac{\Gamma \vdash \texttt{Z} : \texttt{Nat} \qquad \Gamma \vdash x : a \qquad \Gamma \vdash xs : \texttt{List}[a] \qquad \Gamma \vdash x :: xs : \texttt{List}[a]}{\Gamma \vdash \texttt{InFirst} : \texttt{InBounds}[\texttt{Z}, x :: xs]}$$

$$\frac{\dfrac{\Gamma \vdash n : \texttt{Nat}}{\Gamma, n : \texttt{Nat} \vdash S(n) : \texttt{Nat}} \qquad \dfrac{\Gamma \vdash x : a \qquad \Gamma \vdash xs : \texttt{List}[a]}{\Gamma \vdash x :: xs : \texttt{List}[a]} \qquad \Gamma \vdash L : \texttt{InBounds}[n, xs]}{\Gamma \vdash \texttt{InLater}(L) : \texttt{InBounds}[S(n), x :: xs]}$$

As we can see, the two constructors mimic the `index'` definition. The first constructor requires a list with at least one element and if the `n` is zero then we know that we are in bounds. The second constructor builds inductively, if we have a proof that `n` is in bounds to a given list then it is still in bounds in the same list with an added element in front, and consequently with an incremented length. Notably, there is not a

1 From previous listings, we can see that Idris has a type that is the universe of all the types and is called `Type`. However, Idris hides to the user the concept of cumulativity, in fact, there is not only `Type` but an infinite hierarchy of universes where each element of a given universe is also an element of all the higher level universes. The cumulativity is to prevent inconsistency in the type theory that would appear if each type would have been of type `Type`, one of such inconsistency is known as the Girard's paradox, a type theory equivalent of Russell's paradox.

constructor for an empty list because this proposition is trivially false when the list is empty. Following the Curry-Howard isomorphism, if we have an instance of a given type that is isomorphic to a proposition then we have a proof of that proposition. These proofs are easily built, if we want to prove that two is in bounds for this list `['a','b','c']` we construct the instance `InLater InFirst`. Now that we have a proof that an index is in bounds for a given pair of index-list, we can write a procedure to extract the element associated with this index:

Listing 3.13: List indexing procedure.

```
index : (n : Nat) -> (xs : List elem)
     -> (proof : InBounds n xs) -> elem
index Z     (x :: xs) InFirst       = x
index (S n) (x :: xs) (InLater prf) = index n xs
```

Pattern matching on the proof enables the interpreter to exclude impossible cases, in our case an empty list, as previously described. The matched patterns are the same as in `index'` and in the proof constructors however, this time, the compiler can guarantee that the return value is always an element of the list.

This solution is perfect at compile-time, but, if we are dealing with runtime data, like user input, we need a function that builds a proof, just like we intuitively would. First, we need a data type similar to the `Maybe` monad but that can carry with itself not only a proof that a given proposition holds but also a proof that the same proposition might not hold.

Listing 3.14: A data type for decidable properties.

```
data Dec : Type -> Type where
  Yes : (prf : prop) -> Dec prop
  No  : (contra : prop -> Void) -> Dec prop

-- A synonym for (prop -> Void) is (Not prop)
```

The `Yes` constructor holds a proof for a given proposition, the `No` constructor, instead, holds a contradiction. The `Void` type is the empty type, in the formal system we used the $\bot$ symbol for the empty type. Essentially, returning an instance of `Void` is impossible so, the `contra` parameter is telling us that a proof of the decidable proposition is impossible otherwise we could build an instance for `Void` and there is not a single constructor for it. If we would have such an instance then we could construct whatever instance we want, just like from false premises we can imply anything. The `contra` parameter, in our system, would be equivalent to $\Pi\varphi\colon \texttt{Type}.\,\bot$, that, as we can recall from Table 2.1, is isomorphic to $\neg\varphi$.

Listing 3.15: Decision procedure whether a list index is in bounds.

```
inBoundsContraProofHelper : (contra : InBounds k xs -> Void)
                         -> InBounds (S k) (x :: xs) -> Void
inBoundsContraProofHelper contra (InLater prf) = contra prf

inBounds : (n : Nat) -> (xs : List a) -> Dec (InBounds k xs)
inBounds n []        = No uninhabited
inBounds Z (x :: xs) = Yes InFirst
inBounds (S n) (x :: xs) with (inBounds n xs)
  inBounds (S n) (x :: xs) | Yes prf = Yes (InLater prf)
  inBounds (S n) (x :: xs) | No contra
    = No (inBoundsContraProofHelper contra)
```

The decision procedure checks for all possible cases, if we have an empty list then the type is trivially uninhabited, the compiler can determine this autonomously. If the list is not empty and we are searching for the first element then we found it. If the list is not empty, we must search for deeper recursively.[2]

At first glance, finding such procedure can seem tedious and complicated, however, the interpreter gives us serveral tools to help building the procedure incrementally, such as the case splitter and the automatic proof search, that in many cases such as this one, requires zero input from the developer.

Although using a combination of `List` and proofs is simple and functional, we may want to harness the power of dependent types to encode some properties directly to the list type. If we are dealing with the size we can add a parameter representing length to the list type:

Listing 3.16: Definition of static length lists in Idris.

```
data Vect : (len : Nat) -> (elem : Type) -> Type where
  Nil  : Vect Z elem
  (::) : elem -> Vect len elem -> Vect (S len) elem
```

The definition is equal to Listing 3.9 apart from the length parameter that we increment by one for each element we attach to the head of the list. Since the length is directly encoded in the `Vect` type, we can simply pattern match on the implicit argument instead of defining a new function. For indexing we use a different approach, as the length of the list is encoded in the type, we can define a new type that contains only the naturals up to that length.

Listing 3.17: Definition of a set with given length.

```
data Fin : (n : Nat) -> Type where
  FZ : Fin (S k)
  FS : Fin k -> Fin (S k)
```

2 The `with` block performs pattern matching, similarly to `case`, but if matching on a value affects what we know about the form of other values, `with` permits the use of this additional informations to further refine the pattern matching.

The `Fin` type is parameterized, or indexed, by a natural number representing the number of values that are inhabitants of this type. The inductive base constructor requires that the size is at least greater than zero, a set of size zero is the empty set and no value can be constructed. The inductive step constructs a new instance only if an instance of `Fin` exists with a decremented size. An example can offer a more intuitive explanation for this type. Imagine we want to build an instance of `Fin 3`, the first element is trivially `FZ` since we know that three is more than zero. The next instance would be `FS FZ`, from the definition of `FS` we know that `FZ` must be of type `Fin 2` and this is still trivially true. The same line of reasoning is applied to `FS (FS FZ)`, but if we try to build `FS (FS (FS FZ))`, which would be the fourth instance of `Fin 3`, we get a type checker error. Starting from the first `FS` constructor we require a valid `Fin 2` instance, then a valid `Fin 1` instance and eventually a valid `Fin 0` instance, but the base constructor cannot build such a value, so we got a type checker error and we are limited to three possible instances, that is exactly what we wanted. With the `Fin` data type we can define a safe indexing function:

Listing 3.18: Indexing procedure on static size lists.

```
index : (i : Fin len) -> (xs : Vect len elem) -> elem
index FZ     (x :: xs) = x
index (FS i) (x :: xs) = index i xs
```

Using `Fin` we guarantee to the type checker that we cannot pass an empty list and that we cannot use an out of bound index. The append function is identical to Listing 3.10 apart from the type signature, we can be more expressive and mark explicitly that the final length is the sum of the lengths of the components:

Listing 3.19: Append function on static length lists.

```
(++) : Vect n elem -> Vect m elem -> Vect (n + m) elem
(++) []        ys = ys
(++) (x :: xs) ys = x :: (x ++ ys)
```

With dependent types we can easily generalize `Vect` to a heterogeneous list, instead of using the size as a parameter, we can use a list of types of known length:

Listing 3.20: Definition of heterogeneous static size lists.

```
data HVect : (types : Vect k Type) -> Type where
  Nil  : HVect []
  (::) : t -> HVect ts -> HVect (t :: ts)

index : (i : Fin k) -> HVect ts -> index i ts
index FZ     (x :: xs) = x
index (FS i) (x :: xs) = index i xs
```

Once again, the inductive definition is similar both to Listing 3.9 and Listing 3.16, however, besides incrementing the size of the vector implicitly we also store the type of the element we append to the head of the parameterized list of types. The index function is similar to Listing 3.18, we return the value of indexing the vector of types to determine the type of the extracted element of the heterogeneous list.

There are some cases where, while the developer intuition is correct, the type checker cannot prove that the code is, in fact, correct. Imagine we are trying to write a function that intersperse an element to a `Vect` of the same type of elements. A first attempt would be:

```
intersperse : (sep : elem) -> (xs : Vect len elem)
           -> Vect (len + pred len) elem
intersperse sep []      = []
intersperse sep (x :: xs) = x :: intersperse' sep xs
  where
    intersperse' : elem -> Vect n elem -> Vect (n + n) elem
    intersperse' sep []        = []
    intersperse' sep (x :: xs) = sep :: x :: intersperse' sep xs
```

First, we check whether the list has at least one element, if it has, then we use a helper function that returns the list with the interspersed element in front of each remaining list item. The `where` block allows the definition of functions or values local to the function we are defining. However, when we try to type check this definition we get an error on the second definition of the helper. Specifically, we get:

```
Type mismatch between
        S (len + len)
and
        plus len (S len)Unification failure
```

The type checker cannot infer that the sum of a number and its successor is indeed the same as the successor of the sum of that number with itself. We need to aid the type checker with some kind of proof. To carry this proof we use a propositional equality data type, however, without some additional information, the only equality available for each type is when two instances are the same. This `(=)` data type is similar to the Id_σ type we introduced in Section 2.4 with the difference that the types of the two parameters can be heterogeneous:

Listing 3.21: Definition of propositional equality.

```
data (=) : (x : A) -> (y : B) -> Type where
  Refl : x = x
```

Next, we try to prove a more general property that we can use to solve the unification failure: $\forall n, m \in \mathbb{N}.\, S(n + m) = n + S(m)$ where $S : \mathbb{N} \to \mathbb{N}$ is the successor function on naturals.

Listing 3.22: A proof on sum of natural numbers.

```
plusSuccRightSucc : (n : Nat) -> (m : Nat)
                 -> S (n + m) = n + (S m)
plusSuccRightSucc Z m = Refl
plusSuccRightSucc (S n) m =
    rewrite plusSuccRightSucc n m in Refl
```

In the first matched case, the type checker reduces the expression enough to infer that the equality holds. In the second matched case if we use a typed hole we get the following required type:

```
-- plus n m = n + m
S (S (plus n m)) = S (plus n (S m))
```

We can see that inside the most exterior `S` constructor there is exactly the property that we are trying to prove but with the predecessor of the first parameter. So we use induction, calling the same function we are defining on the predecessor and we use the `rewrite` expression to tell the type checker that we have some kind of proof that he can use to further reduce the equality that is trying to unify. The `rewrite` expression needs a property `P : a -> Type` and a propositional equality `x = y`, searches all occurrences of `P y` in the context and replaces them with `P x`, following the identity of indiscernibles principle. After the rewrite process, the type checker has enough information to correctly identify that indeed the two expressions are always the same and we can use the `Refl` constructor. This process of rewriting uses the definitional equality judgements we defined earlier in Rule 3, with the proof carried by the propositional equality type.

Listing 3.23: Definition of interspersion on vectors.

```
-- pred : Nat -> Nat
-- The predecessors of a natural number. pred Z is Z.

intersperse : (sep : elem) -> (xs : Vect len elem)
           -> Vect (len + pred len) elem
intersperse sep []        = []
intersperse sep (x :: xs) = x :: intersperse' sep xs
  where
    intersperse' : elem -> Vect n elem -> Vect (n + n) elem
    intersperse'           sep []      = []
    intersperse' {n = S k} sep (x :: xs)
      = rewrite sym $ plusSuccRightSucc k k in
                sep :: x :: intersperse' sep xs
```

The filter function on `List` is trivial, if we try to directly translate this function to `Vect` we would expect an identical implementation:

```
filter : (elem -> Bool) -> List elem -> List elem
filter p [] = []
filter p (x :: xs) = if p x then x :: filter xs else filter xs
```

```
filter : (elem -> Bool) -> Vect len elem -> Vect len' elem
filter p [] = []
filter p (x :: xs) = if p x then x :: filter xs else filter xs
```

However, before runtime, we cannot know how many items will respect the filter predicate function, and the type checker cannot certify that the value picked for `len'` is, in fact, the correct value. To circumvent this problem we can use dependent pairs or sums introduced in Section 2.3. With dependent sums, we can return both the length of the filtered vector, the witness, and the filtered vector itself, the proof.

Listing 3.24: Filter function on static size lists.

```
filter : (elem -> Bool) -> Vect len elem
      -> (len' : Nat ** Vect len' elem)
filter f [] = (0 ** [])
filter f (x :: xs) = case filter p xs of
  (len' ** xs') => if f x then (S len' ** x :: xs')
                          else (len' ** xs')
```

Finally, the last example of dependent types involves dependent pattern matching. As we have seen from previous examples, dependent pattern matching on a value can have effects on the form of other values. Using a purposely built data type we can force a specific form on another data type, a different view. We explore views building a type for traversing static size lists in reverse order:

Listing 3.25: View for list backward traversing.

```
-- Proof that the empty vector is the right identity of vectors
vectNilRightNeutral : (xs : Vect n a) -> xs ++ [] = xs
-- Proof that vector concatenation is associative
vectAppendAssociative : (xs : Vect xlen elem)
                     -> (ys : Vect ylen elem)
                     -> (zs : Vect zlen elem)
                     -> xs ++ ys ++ zs = (xs ++ ys) ++ zs

data SnocVect : (xs : Vect n a) -> Type where
  Empty : SnocVect []
  Snoc  : {x : a} -> {xs : Vect n a} -> (rec : SnocVect xs)
       -> SnocVect (xs ++ [x])

snocVectHelp : {xs : Vect n a} -> SnocVect xs -> (ys : Vect m a)
            -> SnocVect (xs ++ ys)
snocVectHelp {xs} x [] = rewrite vectNilRightNeutral xs in x
snocVectHelp {xs} x (y :: ys)
  = rewrite vectAppendAssociative xs [y] ys in
            snocVectHelp (Snoc x {x = y}) ys

snocVect : (xs : Vect n a) -> SnocVect xs
snocVect xs = snocVectHelp Empty xs
```

The `snocVect` function builds, in linear time, a view on the list parameter. If we pattern match on the view, automatically the type checker infers that the list must be built in the way carried by the view. Now that we have a way to traverse a list in the reverse order, we can build a reverse function using this view:

Listing 3.26: Reverse function on static length lists.

```
reverse : Vect n a -> Vect n a
reverse {n} xs with (snocVect xs)
  reverse {n = Z} [] | Empty = []
  reverse {n = (k + (S Z))} (xs ++ [x]) | Snoc rec
    = rewrite plusCommutative k 1 in
              x :: (reverseText xs | rec)
```

With the support of the `with`, block we pattern match on the built view. Matching on a non empty list allows the type checker to infer not only that the size of the vector must be some number plus one, but also to infer that the list is in fact represented in the same way that is explicitly coded in the view, namely a list with an element appended to the end. Some rewriting is needed to obtain exactly the return expression that we expected.

The syntax `(reverseText xs | rec)` is soliciting the compiler to reuse the view we already built instead of recreating a new one for the rest of the list, although it is not strictly required.

4 THE PROGRAM

4.1 INDUCTIVE DEFINITION

The development starts with the careful selection of the directed graphs data structure, preferring richness in representation of properties over performance measures. The chosen representation must also be easily translated from and to more common structures like adjacency matrices or already established exchange formats. The mathematical representation of directed graphs and the categorical approach for the rewriting rules is described in [Cor+97].

Definition 4.1 (Directed Graph)**.** A quadruple $G = \langle V, E, s, f \rangle$ is called a directed graph if V is a set of graph vertices, E a set of edges and $s, t : E \to V$ functions from edges to vertices with s representing the source vertex of each edge, and t representing the target vertex of each edge.

Two additional sets can be added: L_V and L_E with two unary functions $l_V : V \to L_V$ and $l_E : E \to L_E$, representing an alphabet of labels for each vertex and edge.

Remark. Since edges are objects on their own right, this directed graph definition allows multiple edges between the same vertices. This demands additional management, particularly when finding subgraphs as the same combination of vertices might be connected with different sets of edges.

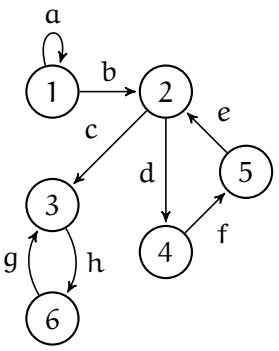
The Definition 4.1 can be easily translated in code, with one exception, although it is, in principle, possible to use Idris functions for s and t, they have some disadvantages: first they are difficult to build, particularly from user input, secondly Idris lacks the concept of extensional equality[1] so constructing proofs of equality on these functions requires to build a proof for each instance of the function domain or to postulate the extensionality requiring additional theoretic study to prove that extensionality does not create inconsistency in the underlying type system.

Instead, we explored an inductive definition where each inductive step constructs a single edge as shown in Listing 4.1, similar to the definition of `Vect` from Listing 3.16.

1 This means that given two functions `f, g : a -> b`, we can construct an equality like `(x : a) -> f x = g x` but we cannot simply assert that `f = g`. Setoids, sets that carry an equality function with themselves, could have been used, however, they would require to build different setoids with proofs for each set of vertices and edges, hindering the end-user usability.

Listing 4.1: Directed graph type definition.

```
data Graph : (n, m : Nat) -> {vertex, edge : Type}
          -> (vs : Vect n vertex) -> (es : Vect m edge)
          -> Type where
  Empty : Graph n Z vs []
  Edge  : (s : Fin n) -> (t : Fin n) -> Graph n m vs es
       -> Graph n (S m) vs (e :: es)
```



$V = \{1,2,3,4,5,6\}$
$E = \{a,b,c,d,e,f,g,h\}$
$s = \{(a,1),(b,1),(c,2),(d,2),(e,5),(f,4),(g,6),(h,3)\}$
$t = \{(a,1),(b,2),(c,3),(d,4),(e,2),(f,5),(g,3),(h,6)\}$

```
example : Graph 6 8 [1,2,3,4,5,6]
                    ['a','b','c','d','e','f','g','h']
example = Edge 1 1 $ Edge 1 2 $ Edge 2 3 $ Edge 2 4 $
          Edge 5 2 $ Edge 4 5 $ Edge 6 3 $ Edge 3 6 $
          Empty
```

Figure 4.1: An example of a directed graph with both representations.

The type is indexed by the number of vertices `n`, the number of edges `m`, the type of the vertices and edges labels, i. e. `vertex` and `edge`, and the vectors with the labels `vs` and `es`. The label types are given as implicit parameters as, most of the time, they can be inferred from the vertices and edges label vectors. In this context each vertex and edge is represented by a natural number between 0 and, respectively, n or m, that can be used to safely access the label vectors. The type constructors are inductive: the base step is a directed graph with some number of vertices and without edges, the inductive step adds an edge from the vertex with index `s` to vertex with index `t` to an already constructed directed graph. The `Fin n` type, already shown in Listing 3.17, is used to guarantee that the vertices are in the correct range, as an out of bound instance cannot be constructed. By default, a canonical representation is not enforced, since for our purpose and with our assumptions it is not necessary to have.

The implementation of many utility functions over directed graphs is fairly straightforward, for example finding the label of a vertex given its index or finding the in-degree of a vertex. Dependent types allow richer properties to be expressed, given the in-degree of a given vertex we can return a vector of all its predecessors guaranteeing at compile time that its length will be equal to the in-degree.

As we can see from Listing 4.2 the implementation of `indegree` follows the inductive definition of a directed graph and `predecessors` follows the same code path of `indegree`, it does not require additional information to help the type checker, as normalization is enough to

Listing 4.2: Example of dependent return type function.

```
indegree : (v : Fin n) -> (g : Graph n m vs es) -> Nat
indegree v Empty = 0
indegree v (Edge _ t g) with (decEq v t)
  indegree v (Edge _ v g) | Yes Refl = S (indegree v g)
  indegree v (Edge _ _ g) | No _     = indegree v g

predecessors : (v : Fin n) -> (g : Graph n m vs es)
            -> Vect (indegree v g) (Fin n)
predecessors v Empty = []
predecessors v (Edge s t g) with (decEq v t)
  predecessors v (Edge s v g) | Yes Refl = s :: predecessors v g
  predecessors v (Edge _ _ g) | No _     = predecessors v g
```

Listing 4.3: Predicate assessing whether a directed graph contains the given edge between two vertices.

```
data HasEdge : (s, t : Fin n) -> (g : Graph n m vs es)
            -> Type where
  Here  : HasEdge s t (Edge s t g)
  There : HasEdge s t g -> HasEdge s t (Edge s' t' g)
```

prove that in fact, we are adding elements to the vector exactly whenever we would increment the in-degree. Although some boilerplate is necessary, the case split functionality provided by the Idris compiler reduces sensibly the need of manually typing all the cases. Notable is the use of the `decEq` function that returns an equality proof instead of a traditional boolean value. The proof can be used by the type checker for normalization and rewriting since it guarantees that, in fact, the target vertex is the same as the vertex we are calculating the predecessors of, as can be observed by the `Yes` branch where the `with` blocks allow to use the same variable for both the searched index and the index inside the `Edge` constructor as proven by the `Refl` constructor.

More complex procedures can be built with auxiliary types like `HasEdge` (see Listing 4.3), which gives a proof that a directed graph contains a specified edge, and its instances can be used to retrieve the first of these edges.

The proofs are built inductively following the definition of a directed graph. At first, if we meet a different edge we go further until we find the correct one. If there is none, then the type is inhabited and proves, under the Curry-Howard correspondence, that there is not an edge of that specification. Given a construction of this predicate, it can be used to retrieve the index of the edge, recurring on the instance, as can be seen in Listing 4.4:

Some functions, however, provide quite a challenge, shifting the burden from debugging to the conception itself of the function. Examples

Listing 4.4: Function that retrieves the index of an edge given its source and target. The `auto` keyword instructs the compiler to search the required value in the context of the caller.

```
edgeIndex : (s, t : Fin n) -> (g : Graph n m vs es)
         -> {auto prf : HasEdge s t g} -> Fin m
edgeIndex s t (Edge s t g) {prf = Here} = FZ
edgeIndex s t (Edge _ _ g) {prf = There prf}
  = FS (edgeIndex s t g)
```

are the removal functions, in particular, the multiple removal function, like a vertex removal that requires to remove all connected edges. Removing multiple edges requires not only the correct distinct indices, but also that the final graph is of the correct length, without providing it in a dependent pair, and the number of edges to be removed must be less or equal to the whole number of edges.

The first approach was to use a `Vect k (Fin m)` and remove the given edges with recursion over the vector. There are multiple ways of ensuring that `k` is really less or equal to `m`: one is to impose that the number of edges is `k + m` and the vector becomes `Vect k (Fin (k + m))`, the other method is to provide a proof carried by the predicate `LTE k m`, which has instances if the natural `k` is less or equal to the natural `m`. However, vectors do not carry with themselves any proof that the indices are distinct and even after imposing this condition as informal precondition, the signatures of all functions involved were convoluted and not intelligible.

Another approach, the chosen one, is to build a type indexed by a vector, which states for each element if it is in or out, effectively splitting the vector into two subsets easily buildable with recursion over this type and the vector (see Listing 4.5 for the definition of such type). The manipulation functions treat the subset like a set, implementing union, intersections, and subtractions just like sets. Also, proof is provided that the subset size is really less than or equal to the size of the vector, and finally the `extract` function returns a new vector with the correct size and only the elements marked as `In`, fulfilling our needs.

4.2 MORPHISMS AND PUSHOUTS

Morphisms

The rewriting procedure requires searching correspondences between directed graphs, this leads to the definition of morphisms between directed graphs.

Listing 4.5: Subset predicate upon static size vectors and size proof with extraction.

```
data VSub : Vect n a -> Type where
  Empty : VSub []
  In    : VSub xs -> VSub (x :: xs)
  Out   : VSub xs -> VSub (x :: xs)

lteSuccRight : LTE n m -> LTE n (S m)

sizeLTE : (xs : Vect n a) -> (sub : VSub xs) -> LTE (size sub) n
sizeLTE []        Empty     = LTEZero
sizeLTE (x :: xs) (In sub)  = LTESucc (sizeLTE xs sub)
sizeLTE (x :: xs) (Out sub) = lteSuccRight (sizeLTE xs sub)

extract : (xs : Vect n a) -> (sub : VSub xs) -> Vect (size sub) a
extract []        Empty     = []
extract (x :: xs) (In sub)  = x :: extract xs sub
extract (x :: xs) (Out sub) = extract xs sub
```

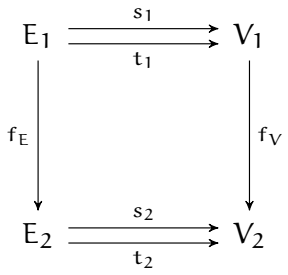


Figure 4.2: Commutative diagram of a morphism between two graphs, as described by Equation 4.1.

Definition 4.2 (Graph Morphism)**.** Let be $G_1 = \langle V_1, E_1, s_1, t_1 \rangle$ and $G_2 = \langle V_2, E_2, s_2, t_2 \rangle$ two directed graphs, $f : G_1 \to G_2$ is called a morphism between G_1 and G_2 if it is a pair of functions $f = \langle f_V, f_E \rangle$ with $f_V : V_1 \to V_2$ and $f_E : E_1 \to E_2$ such that:

$$f_V \circ s_1 = s_2 \circ f_E \quad \text{and} \quad f_V \circ t_1 = t_2 \circ f_E \tag{4.1}$$

Given $G = \langle V, E, s, t \rangle$, the identity morphism $id_G : G \to G$ is defined as $id_G = \langle id_V, id_E \rangle$.

Given three directed graphs G_1, G_2, G_3 and the pair of morphisms $f : G_1 \to G_2$ and $g : G_2 \to G_3$, $g \circ f = \langle g_V \circ f_V, g_E \circ f_E \rangle$ is a graph morphism $g \circ f : G_1 \to G_3$.

The required properties guarantee that after the morphism is applied, the structure of the original directed graph is preserved, in other contexts this kind of morphism is called a homomorphism.

The Definition 4.2 can be translated easily to Idris code, although two additional requirements are added. Due to the lack of extensional

Listing 4.6: Morphism between two directed graphs.

```
data Morphism : (g  : Graph n  m  vs  es)
             -> (g' : Graph n' m' vs' es')
             -> Type where
  Morph : {g : Graph n m vs es} -> {g' : Graph n' m' vs' es'}
       -> (vmap : Vect n (Fin n')) -> (emap : Vect m (Fin m'))
       -> Morphism g g'

data PreserveSource : (g  : Graph n  m  vs  es)
                   -> (g' : Graph n' m' vs' es')
                   -> (mor : Morphism g g') -> Type where
  Empty : PreserveSource Empty g' mor
  Edge  : source e g' = index s vmap
       -> PreserveSource g            g' (Morph vmap emap)
       -> PreserveSource (Edge s t g) g' (Morph vmap (e :: emap))

data Homomorphism : (g  : Graph n  m  vs  es)
                 -> (g' : Graph n' m' vs' es')
                 -> (mor : Morphism g g') -> Type where
  Homo : PreserveSource g g' mor -> PreserveTarget g g' mor
      -> Homomorphism g g' mor
```

equality and the difficulties of dealing with functions manipulation, a simpler representation of the functions f_V and f_E is used. If we have a directed graph `G1` with its corresponding set of vertices `V1` of size `n` and a directed graph `G2` with set of vertices `V2` of size `n'` then the function f_V is represented with a vector `fv` of size `n` of natural numbers between `0` and `n'` (using the `Fin n'` type). Indexing `fv` with a natural `i` is the same as $f_V(v)$ with v the i-th vertex in some defined order (see the `Morphism` data type in Listing 4.6).

The Equation 4.1 is codified in two predicates, one for sources and one for targets. They follow quite literally the mathematical definition, recurring on the edges of the domain directed graph, except for indexing in place of function application (see the `PreserveSource` in Listing 4.6 for one of the implementations).

Finally, the `Homomorphism` is a product type of the two properties, indexed by the directed graphs involved in the morphism, and the chosen morphism itself. As we have seen in Section 2.4 the product of types is in correspondence to the logical conjunction.

Besides all the predicate definitions we need a dynamic procedure for determining if a given predicate is true or false with a proof. All custom predicates that deal with directed graphs have a dynamic procedure implemented and they are used extensively, particularly to deal with user input at runtime but without losing the power of the type checker.

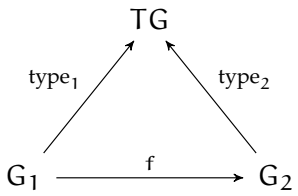


Figure 4.3: Commutative diagram of the definition of typed morphism between two typed directed graphs, as described by Equation 4.2.

Typed graphs and morphisms

All but one rewrite rule use regular directed graphs with regular morphisms. To code this rule we must first define what is a typed directed graph.

Definition 4.3 (Typed Graph)**.** Let us call a regular directed graph TG, a type graph. We define typed graph of type TG, a pair $G^T = \langle G, type \rangle$ where G is a regular directed graph and $type : G \to TG$ is a morphism from the directed graph G to the type graph TG.

Definition 4.4 (Typed Morphism)**.** Let TG be a directed type graph, $G_1^T = \langle G_1, type_1 \rangle$ and $G_2^T = \langle G_2, type_2 \rangle$ two directed graphs of type TG. A morphism $f : G_1^T \to G_2^T$ between the two typed directed graph is a morphism from G_1 to G_2 such that:

$$type_2 \circ f = type_1 \tag{4.2}$$

The identity morphism and the rule of composition are the same as regular morphisms.

Although typed graphs with typed morphisms form a category in its own right, for simplicity sake, we define rewrite rules that involve typed graphs, simply by requiring all the necessary typing morphism. No additional data structure is required for typed graph, nonetheless, its implementation would be trivially the product type of a graph and its typing morphism, with the type indexed by the type graph.

Rewrite Rules

After defining the concept of morphism, we can define some rewrite rules to be implemented. All the rules are implemented in single namespaces inside the `Data.Graph.Pushout` package. The rules are based on double pushout in the category of directed graphs or typed directed graphs. Some simplification have been taken:

- The type of the labels must be the same for each directed graph involved in the process of rewriting, this is reflected by the definition of the rules, nonetheless, the labels of the vertices and the labels of the edges can be different.

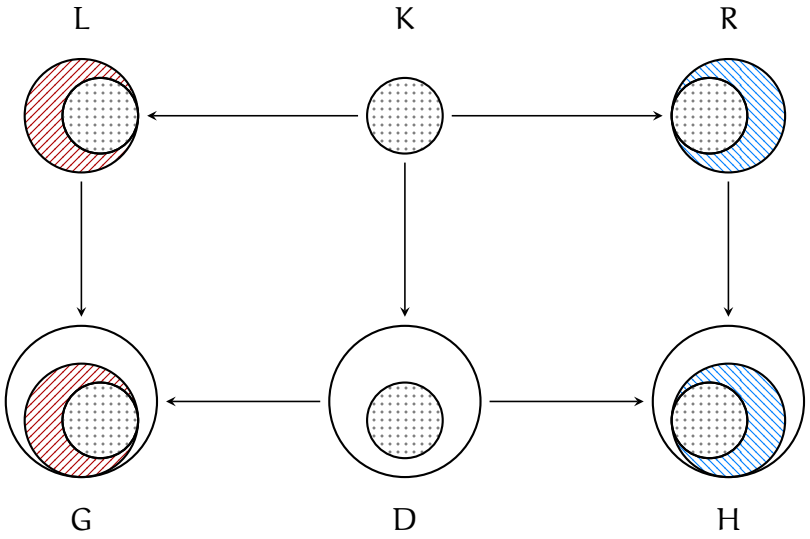


Figure 4.4: Visualization of a standard rewrite rule using a double pushout in the category of directed graphs. The K subgraph is preserved during all the process, conversely the ▨ highlighted part in the L subgraph is substituted by the ▨ highlighted part in the R subgraph.

- We assume that the morphism from the preserved part of the matched subgraph to the whole subgraph (in the rules defining the morphism is marked as $K \to L$) is injective, this guarantees that the morphism for the preserved subgraph of the rule to the matched preserved subgraph of the graph is deterministic, essentially making the whole process deterministic.

- Although the underlying implementation reflects the definition of morphism as a pair of mapping functions, we defined the rules in terms of only the vertex mapping function, precalculating, if possible, the edge mapping function to respect the Equation 4.1. In the context of a simple client frontend, this assumption makes the input process much simpler and almost always produces the same edge mapping function that would have been inserted manually by the user.

- Optionally, we could accept only directed graphs without multi-edges. Following the previous assumption, a client must specify, when multiedges are present, the edge mapping functions of involved morphisms, otherwise we may apply rewrite rules to unintended edges.

Some support data structures are defined for each rule, they contain all the graphs involved and the definition of the morphisms with some additional information that can be precalculated to aid the rewrite process. For example, the simplest of the rewrite rules (as can be seen in Listing 4.7), contains a reference to all the graphs

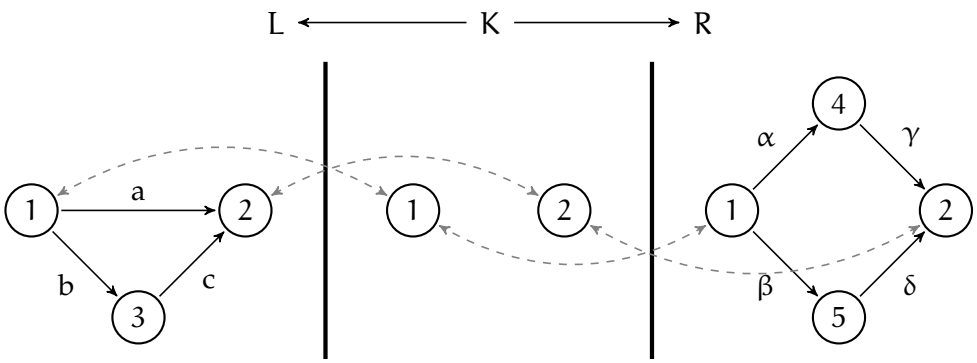


Figure 4.5: Example of a simple rewrite rule. In the context of this visualization, if two vertices have the same number then they are mapped to each other, the same applies to each edge with the same letter.

involved, the morphisms and an additional function that specifies how each preserved vertex in the subgraph that needs to be attached, is correlated to the same vertex in the subgraph that needs to be removed. These additional data structures can all be precalculated without the user involvement.

Listing 4.7: Simple rewrite rule for directed graphs.

```
data Rewrite : Type -> Type -> Type where
  MkRule : (l : Graph ln lm {vertex} {edge} lvs les)
        -> (r : Graph rn rm {vertex} {edge} rvs res)
        -> (k : Graph kn km {vertex} {edge} kvs kes)
        -> (ktol : Vect kn (Fin ln))
        -> (ktor : Vect kn (Fin rn))
        -> (mapping : Vect rn (Maybe (Fin ln)))
        -> Rewrite vertex edge
```

4.3 SUBGRAPHS

The first step, after choosing the rewrite rule, is to find a matching subgraph to apply the rule. Finding such a subgraph is a problem of finding an isomorphic subgraph, thus we can formalize it in the following way:

Definition 4.5 (Graph Isomorphism)**.** Given $G = \langle V_G, E_G, s_G, t_G \rangle$ and $H = \langle V_H, E_H, s_H, t_H \rangle$ directed graphs, we say that G is isomorphic to H, $G \cong H$, if it exist a pair of morphisms $f : G \to H$ and $f^{-1} : H \to G$ such that:

$$f \circ f^{-1} = id_H \quad \text{and} \quad f^{-1} \circ f = id_G \tag{4.3}$$

Definition 4.6 (Subgraph)**.** Given two graph $G = \langle V_G, E_G, s_G, t_G \rangle$ and $S = \langle V_S, E_S, s_S, t_S \rangle$, S is a subgraph of G if $V_S \subseteq V_G$, $E_S \subseteq E_G$ and $s_S, t_S : E_S \to V_S$ such that:

$$\forall e \in E_S . s_S(e) = s_G(e) \wedge t_S(e) = t_G(e) \tag{4.4}$$

The subgraph isomorphism problem is $\mathcal{NP}$-complete, however, multiple heuristic-based algorithms provides polynomial time-complexity most of the times, such as the Ullmann algorithm [Ull76] or the VF2 algorithm [Cor+04].

Eventually, the VF2 algorithm was chosen as it offers a good improvement with respect to Ullmann but with a much lower memory footprint, furthermore it is pretty easy to code with implementation in many other languages that have been used as reference points.

The implementation is pretty straightforward and does not use a whole lot of dependent type features, primarily bound checking with dependent length vectors. The matching function returns, if possible, a list of pairs of morphisms that define the isomorphism between the directed graph and the searched subgraphs, at this point in time, a coded proof that it is indeed an isomorphism is not given, relying only on the correctness of the underlying algorithm. The current state of the algorithm is retained in a dependent record type[2] indexed by the number of vectors of the searched graph and the number of vectors of the subgraph. The two indices prevent runtime bound checking procedures.

Subgraph extraction is more tedious, particularly if we try to force the graph calculations on the type level; inference with recursion is not enough when we have non trivial type functions, and proving that we are constructing the correct instance requires a huge number of equality to be proven and rewritten during typechecking, compile time slows down and debugging is harder. In this case, a more precise return type is not worth the effort and dependant pairs are far easier to work with and are already required to deal with user input. The implementation is found in the `subgraph` function inside the `Data.Graph.Subgraph` module. The list of vertices to be extracted is required in form of a subgraph predicate on the vector of vertices and the extraction is comprised of multiple vertex removal calls.

Besides subgraph extraction we need also glued extraction, implemented in the `gluedGraph` function. The operation is similar to subgraph extraction, given a subgraph we extract the inverse of the subgraph plus a number of vertices and edges glued to the subgraph that needs to be preserved. During the rewrite procedure we need to extract and replace the precondition subgraph, however some of the frontier vertices may be preserved and the postcondition graph

2 Dependent records are desugared to regular dependent product types with namespaced extraction function and a single constructor.

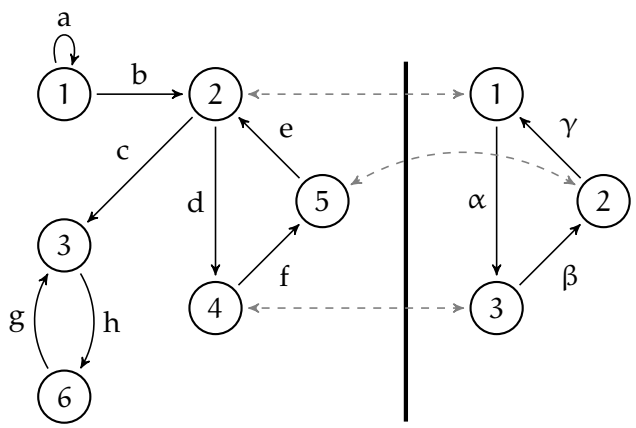


Figure 4.6: Example of a isomorphic subgraph.

merged upon these vertices. As end result, the glued graph and a morphism from the glued graph to the starting graph is returned.

At last, a merging function is needed, implemented in the `merge` function. It appends a graph to another, given which vertices are the same in both the graphs. It returns the new graph and morphism from the single components to the new composed graph. The function is more complex than other manipulation functions, multiple helper functions have been written and also custom proofs were required by the type checker, in particular for limit cases like when we deal with empty graphs. The totality checker cannot automatically infer that the merge function is total, since some of the helper functions arguments are not structurally reducing. Future work on this project would focus towards a proven total and easier to understand merge function.

4.4 GRAPH REWRITING

The rewrite procedure requires an input graph, that for simplicity has only string labels, a rewrite rule (all the available ones are in Appendix A) and an index that selects to which subgraph to apply the rule. At this stage, the possible indices need to be precalculated. The example frontend provides a command that returns all the matches with their indices. The output of each procedure is either a directed graph, in dependent pair form, or an error message.

Each rule is implemented in its own module and all share some utility functions in the `Rewrite.Utils` module. The steps needed can be roughly summarized with some common phases:

1. The injectivity of the morphism from the subgraph to match and the preserved part in the rewrite rule is checked to guarantee determinism. In future, this phase can be prechecked and the proof would be required by the rewrite function.

2. All morphisms involved in the rewrite rule are checked for the Equation 4.1 (or Equation 4.2 if we are dealing with typed directed graphs). Again, this phase can be, in the future, decoupled from the rewrite process.

3. The subgraph that needs to exist from the rewrite rule is matched to the input directed graph. The input index is used to choose which subgraph applies the rule.

4. The matched subgraph is extracted from the input graph, with all the morphisms to the subgraph checked for Equation 4.1.

5. The preserved subgraph and the vertices outside the rule are extracted from the input graph, with all the relative morphism checked.

6. The end subgraph in the rule is merged to the preserved subgraph, and the last morphisms are checked.

7. If a rewrite rule needs additional morphisms, they are checked in this phase.

8. At last, all commutativity rules are checked.

As an example we see the implementation of the simple rewrite rule (Appendix A):

Listing 4.8: Implementation of simple rewrite.

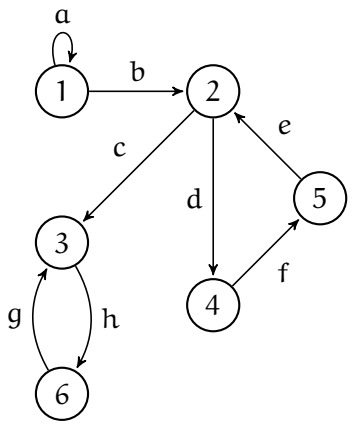
```
-- ExGraph is graph built with dependent pairs
singleRewrite : (graph : Graph n m {vertex=String} {edge=String} vs es)
             -> (rule : Rewrite String String) -> (i : Nat)
             -> Either RewriteError (ExGraph String String)
singleRewrite {vs} g (MkRule l r k ktol ktor kmap) i = do
  -- Checking injectivity and that the rule is valid
  checkInjective "K" "L" ktol
  (ktolMorph ** ktolPrf) <- findMorphism "K" "L" k l ktol
  (ktorMorph ** ktorPrf) <- findMorphism "K" "R" k r ktor

  -- Subgraph matching and morphism
  (ltog', _) <- indexCheck i (match g l)
  ltog <- convertVect "L" "G" ltog'
  (ltogMorph ** ltogPrf) <- findMorphism "L" "G" l g ltog

  -- Subgraph and glued extraction
  let lsubset = vectToSubset ltog'
  let ksubset = fromFins {xs = vs} $ toList $ ltog . ktol
  let kedges = toList $ ltog .*. ktol .*. (edges k)
  let (_ ** _ ** _ ** _ ** sub) = subgraph lsubset g
  let (_ ** _ ** _ ** _ ** (d, dtog))
    = gluedGraph lsubset ksubset kedges g

  -- Morphism from the preserved subgraph
  ktod <- convertVect "K" "D" $ dtog .! ltog . ktol
  (ktodMorph ** ktodPrf) <- findMorphism "K" "D" k d ktod
```

```
(dtogMorph ** dtogPrf) <- findMorphism "D" "G" d g dtog

-- Merging with morphisms
let (_ ** _ ** _ ** _ ** (h, rtoh, dtoh))
  = merge (mergeMapping ktor ktod) d r
rtoh <- convertList "R" "H" rtoh
(rtohMorph ** rtohPrf) <- findMorphism "R" "H" r h rtoh
(dtohMorph ** dtohPrf) <- findMorphism "D" "H" d h dtoh

-- Commutative check
checkPath "K -> L -> G" "K -> D -> G" (ltog . ktol) (dtog . ktod)
checkPath "K -> R -> H" "K -> D -> H" (rtoh . ktor) (dtoh . ktod)

-- Pushout definition
let dp = DP ltogPrf ktodPrf rtohPrf ktolPrf ktorPrf dtogPrf dtohPrf
pure (_ ** _ ** _ ** _ ** h)
```

The implementation roughly follows the steps described earlier. We rely on the do notation to reduce the boilerplate needed for error management, each function may return an error where the `RewriteError` is the sum type with all of them; the strings are required for a more complete error output. First, we check the injectivity of the rule, which is needed for determinism. Next, given a vertex mapping function, `findMorphism` tries to produce the complete morphism definition and checks that, indeed, it respects the definition described in Equation 4.1. The `(.)` operator defines vertex morphism composition, mirroring function composition. Edge morphism composition is implemented with the `(.*.)` operator. When computing some morphisms, for example the morphism from the glue in the rewrite rule to the corresponding glue in the starting graph, we may need to invert an already computed morphism. If the rewrite rule follows our assumptions, all the involved morphisms are uniquely identifiable. However, in general, morphisms are not bijective and so we need to prove that, with composition, we are restricting the image of the morphism to only the invertible vertices. Since, proving this property and the uniqueness of the morphisms is particularly hard, we return the morphism in a `Maybe` monad and subsequently we extract it from the monad. If the extraction process is unsuccessful then the rewrite rule must be incorrect and we return an error. Vertex morphism composition wrapped in `Maybe` is implemented by the `(.!)` operator. After subgraph extraction, merging and computing the required morphisms, the `checkPath` function is used to check that the defined morphism commute as described by the commutative diagram in Figure A.1.

4.5 INPUT/OUTPUT

Although the `Graph` data type is easily manipulated with recursion and can be used to prove properties of directed graphs, it is not particularly intelligible for humans, yet it is another format in a plethora of other




```
digraph { 1 -> 1 [label=a]; 1 -> 2 [label=b];
          2 -> 3 [label=c]; 2 -> 4 [label=d];
          5 -> 2 [label=e]; 4 -> 5 [label=f];
          6 -> 3 [label=3]; 3 -> 6 [label=h] }
```

```
example : Graph 6 8 [1,2,3,4,5,6]
                ['a','b','c','d','e','f','g','h']
example = Edge 1 1 $ Edge 1 2 $ Edge 2 3 $
          Edge 2 4 $ Edge 5 2 $ Edge 4 5 $
          Edge 6 3 $ Edge 3 6 $ Empty
```

Figure 4.7: Example of a graph coded both in the DOT format (up) and the project data structure (down).

standards. Therefore, the example frontend implements parsing and pretty-printing of a subset of the DOT format [GKN06], already available and compatible with other applicatives. The DOT format is much easier to read for humans (see Figure 4.7 for comparison between the two formats), it is concise and supports a wide range of attributes and it is the format supported by the Graphviz tool, a free open-source tool that uses DOT format for automatically arranging and producing representations of graphs in various formats, including PDF, SVG and PNG. The Graphviz command-line tools can be used to make good images of the graphs and rules without human effort, making it a good choice for clients.

The frontend is structured as a standalone process that responds to a single request at a given time, miming the command line tools in UNIX systems. The exchange protocol uses the JSON format [Bra17] both for input and output. The input must be provided in a file passed as argument of the command, the output is printed to the standard output. This simpler style of management is due to a lack of a library ecosystem in many experimental languages. A more complete approach would be an objective of future development, but at this point in time more focus was spent on the rewrite and data structures aspects of the program.

The DOT format parsing procedure is implemented with a simple monadic parser with the `Either` monad used for reporting results, similarly to other solutions in popular functional programming languages like Haskell or OCaml. The final result could be an error message, with errors that immediately halt the execution and do not propagate further, or a directed graph in dependent pair form, preserving the indices of the directed graph data type. Interleaving common functional style code with dependent code was easy, although, verbose at times.

5 CONCLUSIONS

We presented dependent types in the introduction, as a mean of checking complex invariants and the correctness of code. In this project, however, we tried to explore types as a planning tool. First of all, we write the type of functions and data structures as a specification of what they should do, then we provide implementations, sometimes incomplete, particularly with the help of type holes. Instead of inference, we rely on the type checker and on contexts to guide us towards an implementation that satisfies the required specification. With dependent types, in particular, this approach is, in some respects, required as dependent constructs hinder type inference, so it cannot be prevalent as in other type systems, such as the Hindley-Milner type system at the heart of Haskell core language.

Sometimes, encoding all the required properties is not simple. This process requires many successive iterations, so it is important that the type system allows a combination of extended dependent typing and regular non-dependent typing. The most important data structure in this work is the `Graph` data type. While exploring how to implement it, we focused on two axis: which types allowed more properties to be expressed and which types allowed simpler reasoning and required the least amount of work. For the `Graph` data type, the algebraic definition was simple and rich enough to be translated easily and with good results, in particular in regards to translation to more common representations. However, the auxiliary functions required more iterations, in particular dealing with the vectors of vertices and edges. In our experience, ancillary data types, the likes of `HasEdge` or `VSub`, required a little bit of time for their definition, but helped a lot down the road. In particular, `VSub` significantly reduced the amount of rewrite proofs required and worked well with the tools provided by the compiler, since it followed the same deductive reasoning implemented by the other surrounding data types.

As more complete data structures require less testing past implementation, they need more pondering and more work as stubs and dynamic testing is more difficult since the type checker enforcement is stricter, barring subversion functions like `assert_total` and `believe_me`, that must be handled with extreme care. Nonetheless, this shift in the amount of work required for implementation in place of testing and the incorporated invariants do not eliminate the need of testing since the compiler cannot second guess what are the needs of the developer. Also, attention must be paid regarding the totality of functions as the conservativeness of the totality checker can be prone to false-negatives.

The function `assert_smaller` and `assert_total` can subvert the totality checker however it is preferable to use predicate data types that restrict the domain of general recursion functions to the subset of terms that lead to termination (for further reference see [BC05]). In any case, at least the covering check must always be passed for pattern matching.

Theorem proving can be a powerful tool to aid regular testing techniques, however, when dealing with calculated types, it is often required the usage of proofs to pass the type checker. Although, the deductive reasoning required by dependent type theories is often natural to reason about, some proofs not only could be hard to navigate but often require a lot of rewriting for immediate steps, e. g. many proofs involving sum or multiplication on natural numbers require commutativity or associativity rewrites. Proof assistants provide the concept of tactics which, given a goal, are used to automatically split it into subgoals and try to apply lemmas to resolve the subgoal. Advanced tactics can make decisions based on cases and either the success or the failure of other base tactics. They can be used to simplify the process of dealing with complex and long proofs that require a lot of successive case splitting, although, they are not always intelligible and are more oriented towards the proof assistant kind of usage.

Working with existing code could be difficult. Refining types on a complete iteration of a project has a ripple effect on the whole codebase. The compiler ensures that no incompatible code passes the type-checking phase, this requires a lot of work upfront to make the new refined version compatible with the old code. Tooling, in many dependent type languages, is still incomplete. Refactoring a big chunk of code requires a lot of work directly by the developer, as editor support can be hit or miss. E. g. the Idris compiler provides an interface that coding environments can use to automate case splitting, base definition from signatures, type documentation and simple expression search, however IDE-like features common in more popular languages, such as renaming or extraction, are still unavailable or immature. The experimental and research nature of many dependently typed languages leads also towards an immature ecosystem of libraries and lacking standard libraries, as well as not properly optimized code and long type checking and compile times.

Dependent types can be a powerful tool to augment the expressivity of our code, however, developing with a more complex type system requires a trade-off between easiness of use and correctness. While some mission-critical systems require absolute safety that dependent types could offer, other applications can mix dependent and non-dependent type features in a middle ground that offers both quick development time and an additional amount of safety, for example via state machines or domain-specific languages.

Future work on this project will be focused primarily on performance and removing the simplicity assumptions listed in the previous

chapters. For performance purpose there are three different avenues that can be followed: usage of lesser computationally intensive algorithms and data structures, such as an optimized version of the subgraph isomorphism algorithm and faster version of the extraction and merge functions; stricter control on runtime erasure of proofs; experimentation with extension to the type system such as, for example, linear types [Atk18].

A REWRITE RULES

SIMPLE RULE

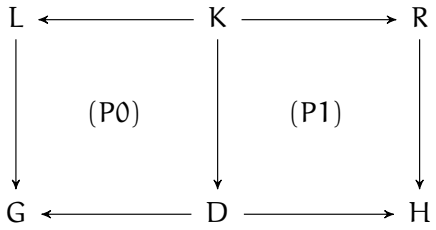


Figure A.1: Commutative diagram of a simple rewrite rule.

The Figure A.1 diagram assumes we are working in the category of directed graphs as described by the Definition 4.1 and Definition 4.2.

The Figure 4.4 provides a visual interpretation of this rule, here we provide a short summary of each graph or subgraph involved:

L The subgraph that needs to be matched and rewritten according to the rule.

K A subgraph of L that must be preserved by the rewrite procedure.

R The subgraph that will be merged to the input graph to produce the final graph, with the preserved parts specified by the K subgraph.

G The starting input graph.

D The subgraph of G corresponding to the K subgraph that will be preserved by the rewrite process.

H The final output graph.

The initial input is composed of the rule, described by the three graphs L, K, and R and the starting input graph G. D and H are calculated and checked during the rewrite process, together with the commutativity paths P0 and P1.

OTHER RULES

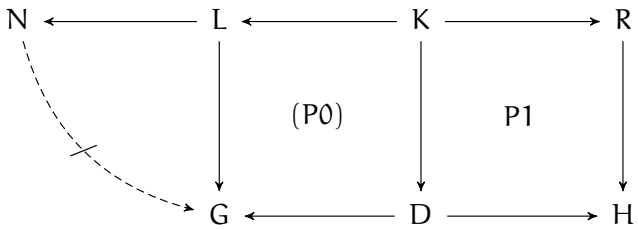


Figure A.2: Commutative diagram of a negative property rewrite rule.

The rule described by Figure A.2 extends the simple rewrite rule with a directed graph N that represents some additional properties of the L subgraph. A morphism from N to G must not exist. This means that this rule can be applied only if the starting graph does not have the property described by N.

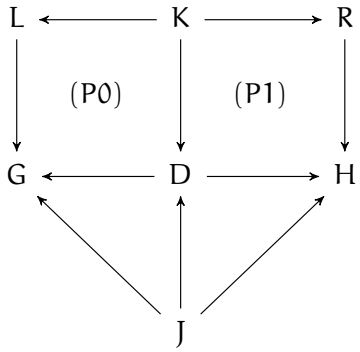


Figure A.3: Commutative diagram of a rewrite rule with an interface.

The rule described by Figure A.3, also, extends the simple rewrite rule with a directed graph J, that represents some additional properties of the G directed graph. A morphism from J to G and the derived graphs D and H must exist. This means that this rule can be applied only if the starting graph has some specific property attached to the part that is not rewritten and it must be preserved during the whole rewrite process.

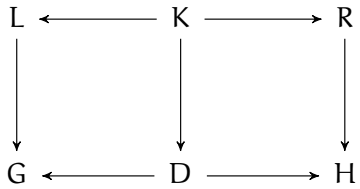


Figure A.4: Commutative diagram of a typed rewrite rule.

The rule described by Figure A.4 is the same rule as the simple rule but in the category of typed graph with some type TG, according to Definition 4.3 and Definition 4.4.

BIBLIOGRAPHY

[Alt+10] Thorsten Altenkirch, Nils Anders Danielsson, Andres Löh, and Nicolas Oury. "ΠΣ: Dependent types without the sugar". In: *International Symposium on Functional and Logic Programming*. Springer. 2010, pp. 40–55 (cit. on p. 4).

[Atk18] Robert Atkey. "Syntax and Semantics of Quantitative Type Theory". In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. ACM. 2018, pp. 56–65 (cit. on p. 39).

[BC05] Ana Bove and Venanzio Capretta. "Modelling General Recursion in Type Theory". In: *Mathematical. Structures in Comp. Sci.* 15.4 (Aug. 2005), pp. 671–708 (cit. on p. 38).

[Bra13] Edwin Brady. "Idris, a general-purpose dependently typed programming language: Design and implementation". In: *Journal of functional programming* 23.5 (2013), pp. 552–593 (cit. on pp. 3, 4, 11).

[Bra17] Tim Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 8259. RFC Editor, 2017. URL: https://tools.ietf.org/rfc/rfc8259.txt (cit. on p. 36).

[CH86] Thierry Coquand and Gérard Huet. "The calculus of constructions". PhD thesis. INRIA, 1986 (cit. on p. 3).

[Cor+04] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. "A (sub) graph isomorphism algorithm for matching large graphs". In: *IEEE transactions on pattern analysis and machine intelligence* 26.10 (2004), pp. 1367–1372 (cit. on p. 32).

[Cor+97] Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, Reiko Heckel, and Michael Löwe. "Algebraic Approaches to Graph Transformation - Part I: Basic Concepts and Double Pushout Approach". In: *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. 1997, pp. 163–246 (cit. on p. 23).

[Cur34] Haskell B. Curry. "Functionality in Combinatory Logic". In: *Proceedings of the National Academy of Sciences* 20.11 (1934), pp. 584–590 (cit. on p. 3).

[CF58] Haskell B. Curry and Robert Feys. *Combinatory Logic*. Combinatory Logic v. 1. North-Holland Publishing Company, 1958 (cit. on p. 3).

[GKN06] Emden Gansner, Eleftherios Koutsofios, and Stephen North. *Drawing graphs with dot*. Tech. rep. AT&T Research, 2006. URL: http://www.graphviz.org/pdf/dotguide.pdf (cit. on p. 36).

[Hof97] Martin Hofmann. "Syntax and semantics of dependent types". In: *Extensional Constructs in Intensional Type Theory*. Springer, 1997, pp. 13–54 (cit. on p. 4).

[How80] William Alvin Howard. "The formulae-as-types notion of construction". In: *To HB Curry: essays on combinatory logic, lambda calculus and formalism* 44 (1980), pp. 479–490 (cit. on p. 3).

[MLS84] Per Martin-Löf and Giovanni Sambin. *Intuitionistic type theory*. Vol. 9. Bibliopolis Naples, 1984 (cit. on pp. 3, 4).

[NPS90] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory*. Oxford University Press, 1990 (cit. on p. 4).

[Nor07] Ulf Norell. "Towards a practical programming language based on dependent type theory". PhD thesis. SE-412 96 Göteborg, Sweden: Department of Computer Science and Engineering, Chalmers University of Technology, 2007 (cit. on p. 4).

[SU06] Morten Heine Sørensen and Pawel Urzyczyn. "Lectures on the Curry-Howard isomorphism". In: *Studies in Logic and the Foundations of Mathematics*. Vol. 149. Elsevier, 2006 (cit. on p. 3).

[Ull76] Julian R. Ullmann. "An Algorithm for Subgraph Isomorphism". In: *J. ACM* 23.1 (Jan. 1976), pp. 31–42 (cit. on p. 32).

[Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: https://homotopytypetheory.org/book, 2013 (cit. on p. 3).