Programming in homotopy type theory and erasing propositions

Gabe Dijkstra

M.Sc. thesis ICA-3354881
[*Supervisors*] Wouter Swierstra and Johan Jeuring

August 26, 2013

Universiteit Utrecht

Department of Computing Science

Abstract

In the recent years, homotopy type theory has become the subject of much study. Homotopy type theory studies the correspondence between the (propositional) equality in Martin-Löf's type theory and the concept of homotopy from topology. The correspondence roughly means that inhabitants of a type can be seen as points of a space and that a propositional equality $x \equiv y$ can be seen as a path $x \rightsquigarrow y$. At the time of writing, virtually all material on the subject is of a rather mathematical nature and focuses on its use in formalising mathematics. This thesis aims to provide an introduction to homotopy type theory geared toward programmers familiar with dependently typed programming, but unfamiliar with topology. We will present applications of homotopy type theory to programming, such as quotient types and dealing with views on abstract types. Furthermore, we will discuss the use of h-propositions to identify parts of a program that are not needed at run-time, compare it to existing methods present in Coq and Agda and discuss whether this can be used to optimise programs. Such an approach works in plain Martin-Löf's type theory. In homotopy type theory however, it does not work in general, but we can identify cases in which it still works.

Contents

Chapter 1

Introduction

One of the tricky things that comes up sooner or later when one designs a type system or a logic, is the defining a right notion of equality. When type checking a term, one needs a suitable concept of equality, e.g. when one type checks an application $f\ \ a$ and we know that $f : A \to B$ and we know that $a : X$, we have to check that A and X are equal in some way. In Martin-Löf's type theory [Martin-Löf, 1985], A and X need to be *definitionally equal* (denoted in this thesis by $A \overset{\Delta}{\equiv} X$): if we reduce both A and X to their normal forms, they need to be syntactically equal.

We also want to be able to reason about equality in the type theory itself: we want to use it in our programs (or proofs) written in the type theory language, e.g. to show that two programs behave in the same way, when given the same input. If we have a version of a meta-theoretical concept, such as definitional equality, that can be expressed in the language of type theory itself, we call such a version of the concept *internal*. The notion of equality internal to a type theory is called a *propositional equality* (in this thesis denoted by $\equiv$). In Martin-Löf's type theory, propositional equality is defined using the so called *identity types*: an inductive family with *refl* as its only constructor. This construction essentially imports definitional equality into the type theory. However, the resulting structure is not exactly definitional equality: as we will see at various points in this thesis, it is valid to add as axioms extra propositional equalities between terms that are not definitionally equal.

We can force the two notions to coincide by adding an *equality reflection* rule, i.e. a rule that states that if we have a proof $p : x \equiv y$ are propositionally equal, then $x \overset{\Delta}{\equiv} y$ also holds. Since type checking makes use of definitional equality, to show that two terms are definitionally equal, we may need to produce a proof of propositional equality first. This proof search means that type checking becomes undecidable. Even though it is undecidable in general, it still works out for enough cases to be useful, as is exemplified by Nuprl [Constable et al., 1986]. One advantage of adding equality reflection is that we can prove useful things such as function extensionality ($((x : A) \to f\ x \equiv g\ x) \to f \equiv g$), something that we cannot prove if we leave the equality reflection rule out.

The study of *intensional type theory*, i.e. type theory without the equality reflection rule, involves finding out why we cannot prove certain properties about propositional equality that are deemed to be natural properties for a notion of equality, such as function extensionality and uniqueness of identity proofs. This eventually led to the discovery of homotopy type theory, an interpretation of types and their identity types in the language of homotopy theory:

type theory	**homotopy theory**
A is a type	A is a space
$x, y : A$	x and y are points in A
$p, q : x \equiv y$	p and q are paths from x to y
$w : p \equiv q$	w is a homotopy between paths p and q
$\vdots$	$\vdots$

The discovery was that propositional equality behaves just like the homotopy we know from topology. This discovery spawned a lot of interest, as it meant that the language of type theory can be used to prove theorems about homotopy theory. It is also regarded as an interesting foundation of mathematics, as it makes working with isomorphic structures a lot more convenient than is the case when working with foundations based on set theory. There are already several introductions on the subject (e.g. Awodey [2012], Pelayo and Warren [2012] and Rijke [2012]). There has been a special year in 2012–2013 on the subject at the Institute of Advance Study in Princeton, which has culminated in a book [The Univalent Foundations Program, 2013], giving a very complete overview of the results. The focus of these materials is on homotopy type theory as a foundation of mathematics and its use in formalising mathematics.

The materials mentioned above assume the reader to have experience with homotopy theory and none with type theory. In this thesis instead assumes the reader to have experience with using a dependently typed language such as Agda as a programming language for certified programs, but have no background in homotopy theory. This leads us to the main research question of this thesis:

> What is homotopy type theory and why is it interesting to do programming in it?

In chapter 2 we give an introduction and overview of some of the main concepts of homotopy type theory. In this chapter we will also provide a very short introduction into topology and homotopy theory, to give a bit of intuition behind the terminology and where the concepts come from. In chapter 3 we discuss several applications of homotopy type theory to programming. In particular we look at how we can implement quotient types in homotopy type theory and contrast this to other ways to work with quotient types. Another application we consider is the use of univalence to deal with views on abstract types. We work out the examples given by Licata [2012] and extend the result to non-isomorphic views, using quotient types.

Homotopy type theory provides us with a notion of propositions, the so called h-propositions. In chapter 4 we compare this to similar notions found in Coq, Agda and Epigram. We investigate whether we can formulate an optimisation based on h-propositions in the spirit of the collapsibility optimisation proposed in Brady et al. [2004].

In the final chapter, chapter 5, we will discuss our answers to our research questions and propose directions of future research.

Since the focus of this thesis is on the programming aspects of homotopy type theory, as opposed to doing homotopy theory, we will not do any diagram chasing and instead will use Agda syntax throughout the thesis. As such, we will expect the reader to be familiar with this language.

Notation We will use Agda syntax for most of the code in this thesis, except for some parts in chapter 4. The code will not always be valid Agda syntax. We will use the notation $A : \mathit{Type}$ instead of $A : \mathit{Set}$, in order to avoid confusion between types and the homotopy type theory notion of h-sets. We will also refrain from mentioning levels and essentially assume that these are automatically inferred. For Σ-types, we will sometimes use the notation $\Sigma\ (x : A)\ .\ B\ x$ instead of $\Sigma\ A\ (\lambda x \to B\ x)$, for brevity.

Code The accompanying code can be found in the appropriate GitHub repository[1]. The file `index.agda` lists for each chapter the modules that contain code relevant to the chapter.

[1] `https://github.com/gdijkstra/hprop-erasibility` and for a browsable variant with syntax colouring: `http://gdijkstra.github.io/hprop-erasibility/`

Chapter 2

Homotopy type theory

As was briefly mentioned in chapter 1, homotopy type theory studies the correspondence between homotopy theory and type theory. As such, we will start out with a very brief sketch of the basic notions of topology and homotopy theory (section 2.1). After that, we will describe the notion of propositional equality in Martin-Löf's type theory using identity types (section 2.2). Having defined the identity types, we can explain the interpretation of Martin-Löf's type theory in homotopy theoretic terms, relating propositional equality to paths (section 2.3). In section 2.4 we describe how the idea of classifying spaces along their homotopic structure can be used in type theory to classify types. Section 2.5 and section 2.6 describe two extensions to Martin-Löf's type theory inspired by homotopy theory. This chapter is concluded by a discussion on the implementation issues of homotopy type theory (section 2.7).

2.1 Topology and homotopy theory

Topology is the study of shapes (called spaces) and continuous functions between spaces. It generalises the familiar notion of continuity from calculus. In *homotopy theory* we are interested in studying continuous deformations . The simplest case of this is continuously deforming one point into another point, which is called a *path*. A path in a space X from point x to y is a continuous function $p : [0,1] \to X$, such that $p\ 0 = x$ and $p\ 1 = y$, also notated as $p : x \rightsquigarrow y$. The set of all paths in X can be also considered as a space. In this space, called the *path space* of X, we again can look at the paths. Suppose we have two paths $p, q : [0,1] \to X$ with the same begin and end points, then a path between p and q, called a *homotopy*, is a continuous function $\gamma : [0,1] \to [0,1] \to X$ where $\gamma\ 0 = p$ and $\gamma\ 1 = q$ (see fig. 2.1). Of course, we can also look at homotopies in these path spaces, and homotopies between these higher homotopies, ad infinitum.

These paths have an interesting structure: we can define operations acting on paths that satisfy certain laws: paths form a groupoid-like structure. If we have a path $p : a \rightsquigarrow b$ and a path $q : b \rightsquigarrow c$, we can compose these to form a path $p \circ q : a \rightsquigarrow c$. For every path $p : a \rightsquigarrow b$, there is a reversed path $p^{-1} : b \rightsquigarrow a$.

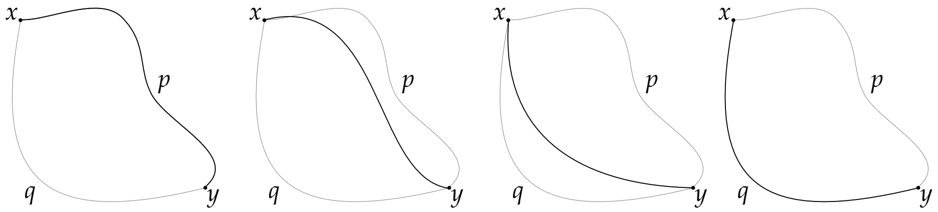

Figure 2.1: A homotopy between paths p and q

For every point a, there is the constant path $r_a : a \rightsquigarrow a$. One might wonder whether reversing a path acts as an inverse operation with r_a being the unit of path composition, i.e. whether we the following equations are satisfied:

- $p \circ (q \circ s) = (p \circ q) \circ s$
- $p \circ p^{-1} = r_a$
- $p^{-1} \circ p = r_b$
- $p \circ r_b = p$
- $r_a \circ p = p$

This happens to not be the case: the equations do not hold in the strict sense. However, both sides of the equations are homotopic to each other. These operations can also be defined on homotopies between paths, for which the same equations can be shown to hold up to higher homotopy. What we get is a tower of homotopies for which we have these groupoid-like structure at every level, in which the equations hold up to homotopy one level higher. This structure is called a ∞-groupoid structure. It was proposed in Grothendieck [1983] that homotopy theory should be the study of these ∞-groupoids, as these should capture all the interesting homotopy properties of a space.

2.2 Identity types of Martin-Löf's type theory

Martin-Löf [1985] introduced a notion of equality internal to his type theory, defined using *identity types*. These types can be formulated in Agda syntax as follows:

```
data Id (A : Type) : A → A → Type where
  refl : (x : A) → Id A x x
```

In order to type check $\mathit{refl}\ x : \mathit{Id}\ A\ x\ y$, the type checker needs to verify that x and y are definitionally equal. The *refl* constructor can be seen as a rule that definitional equality implies propositional equality. The converse does not need to hold: type theories (such as Martin-Löf's type theory) in which we do not have the *equality reflection rule*, that states that propositional equality implies definitional equality, are called *intensional* type theories. *Extensional* type theories are theories in which the equality reflection rule does hold.

If we want to do something with the inhabitants of an inductive type, other than passing them around or ignoring them, we must use the induction principle (or elimination operator) of the inductive type. The induction principle of the Id type is usually called J and has the following type:

$$\begin{aligned} J :\ & (A : \mathit{Type}) \\ & \to (P : (x\ y : A) \to (p : \mathit{Id}\ A\ x\ y) \to \mathit{Type}) \\ & \to (c : (x : A) \to P\ x\ x\ (\mathit{refl}\ x)) \\ & \to (x\ y : A) \to (p : \mathit{Id}\ A\ x\ y) \\ & \to P\ x\ y\ p \end{aligned}$$

Along with this type, we have the following computation rule:

$$J\ A\ P\ c\ x\ x\ (\mathit{refl}\ x) \triangleq c\ x$$

We will make use of a slightly different, but equivalent formulation of these types, due to Paulin-Mohring [1993], where the x is a parameter as opposed to an index, yielding a more convenient elimination principle:

$$\begin{aligned} & \textbf{data}\ \mathit{Id}'\ (A : \mathit{Type})\ (x : A) : A \to \mathit{Type}\ \textbf{where} \\ & \quad \mathit{refl} : \mathit{Id}'\ A\ x\ x \end{aligned}$$

with induction principle:

$$\begin{aligned} J' :\ & (A : \mathit{Type}) \\ & \to (x : A) \\ & \to (P : (y : A) \to (p : \mathit{Id}'\ A\ x\ y) \to \mathit{Type}) \\ & \to (c : P\ x\ x\ \mathit{refl}) \\ & \to (y : A) \to (p : \mathit{Id}'\ A\ x\ y) \\ & \to P\ x\ y\ p \end{aligned}$$

and computation rule:

$$J'\ A\ P\ c\ x\ \mathit{refl} \triangleq c$$

Since the x is a fixed base point, this elimination principle is also called *based path induction* [The Univalent Foundations Program, 2013].

To make things look more like the equations we are used to, we will for the most part use infix notation, leaving the type parameter implicit: $\mathit{Id}\ A\ x\ y$ becomes $x \equiv y$. In some cases we will fall back to the $\mathit{Id}\ A\ x\ y$ notation, when it is a bit harder to infer the type parameter.

Using the identity types and their induction principles, we can show that propositional equality is an equivalence relation, i.e. given $A : \mathit{Type}$ and $x\ y\ z : A$, we can find inhabitants of the following types:

- $\mathit{refl} : \mathit{Id}\ A\ x\ x$
- $\mathit{symm} : \mathit{Id}\ A\ x\ y \to \mathit{Id}\ A\ y\ x$
- $\mathit{trans} : \mathit{Id}\ A\ x\ y \to \mathit{Id}\ A\ y\ z \to \mathit{Id}\ A\ x\ z$

Another important property of propositional equality is that it is a congruence relation, i.e. we have a term with the following type:

$$ap : \{A\ B : Type\} \rightarrow (f : A \rightarrow B) \rightarrow \{x\ y : A\} \rightarrow x \equiv y \rightarrow f\ x \equiv f\ y$$

$ap\ f$ can be read as the (functorial) *action* on *paths* induced by f or the *application* of f on *paths*. If we want to generalise ap to also work on dependent functions $f : (a : A) \rightarrow B\ \ a$, we notice that we get something that does not type check: $f\ \ x \equiv f\ \ y$ does not type check because $f\ \ x : B\ \ x$ and $f\ \ y : B\ \ y$. However, if we have an equality between x and y, then $B\ \ x \equiv B\ \ y$, so we should be able to somehow transform something of type $B\ \ x$ to something of type $B\ \ y$. This process is called *transporting*:

$$transport : \{A : Type\}\ \{B : A \rightarrow Type\}\ \{x\ y : A\} \rightarrow x \equiv y \rightarrow B\ x \rightarrow B\ y$$

$transport$ is sometimes also called $subst$, as $transport$ witnesses the fact that if we have $x \equiv y$, we can substitute any occurrence of x in context B with y.

Using $transport$ we can now formulate the dependent version of ap:

$$\begin{aligned} &apd : \{A : Type\}\ \{B : A \rightarrow Type\}\ \{x\ y : A\} \\ &\quad \rightarrow (f : (a : A) \rightarrow B\ a) \rightarrow (\beta : x \equiv y) \\ &\quad \rightarrow transport\ \beta\ (f\ x) \equiv f\ y \end{aligned}$$

The resulting equality is an equality of between points in $B\ \ y$. Of course it does not matter if we $transport$ to $B\ \ x$ or $B\ \ y$, as propositional equalities are symmetric.

2.2.1 Difficulties of identity types

Even though at first glance the identity types have the right structure: they form equivalence relations on types, there are still some properties that cannot be proven, things that can be useful or seem to be natural properties of a notion of equality.

Function extensionality When doing certified programming, we sometimes want to show one (more optimised) function to be equal to another (naively implemented) function. In these cases it is often necessary to have the principle of function extensionality:

$$\begin{aligned} functionExtensionality\ \ :\ \ & (A\ B : Type) \rightarrow (f\ \ g : A \rightarrow B) \\ & \rightarrow ((x : A) \rightarrow f\ \ x \equiv g\ \ x) \\ & \rightarrow f \equiv g \end{aligned}$$

However, in Martin-Löf's type theory there is no term of that type. The theory satisfies the so called canonicity property: if we have a judgement $\vdash p : \tau$, where τ is some inductive type, p normalises to a term built up solely of constructors. This means that if we have a propositional equality in the empty context, i.e. $\vdash p : x \equiv y$, we know that p must be canonical: it is definitionally equal to *refl*. In order for $\vdash$ *refl* $: x \equiv y$ to type check, we then know that x and y must be definitionally equal. Now consider the functions $f = \lambda n \rightarrow n + 0$ and $g = \lambda n \rightarrow 0 + n$, with the usual definition of $+ : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$ by recursion on the first argument, we can prove that $(n : \mathbb{N}) \rightarrow f\ n \equiv g\ n$, but not that $f \equiv g$, since that would imply they are definitionally equal, which they are not: one reduces to $\lambda n \rightarrow n$, whereas the other reduces to $\lambda n \rightarrow n + 0$.

Uniqueness of identity proofs The canonicity property implies that if we have $\vdash p : \mathit{Id}\ A\ x\ y$ and $\vdash q : \mathit{Id}\ A\ x\ y$, these proofs are both *refl*, hence they are equal to one another: $p \equiv q$. One would expect that it is possible to prove this inside Martin-Löf's type theory. Using dependent pattern matching [Coquand, 1992], we can easily prove this property in Agda, called uniqueness of identity proofs:

$$\begin{aligned}&\mathit{UIP} : (A : \mathit{Type})\ (x\ y : A)\ (p\ q : \mathit{Id}\ A\ x\ y) \rightarrow \mathit{Id}\ (\mathit{Id}\ A\ x\ y)\ p\ q\\ &\mathit{UIP}\ A\ x\ .x\ \mathit{refl}\ \mathit{refl} = \mathit{refl}\end{aligned}$$

Proving this using J instead of dependent pattern matching has remained an open problem for a long time and has eventually been shown to be impossible [Hofmann and Streicher, 1996] by constructing a model of Martin-Löf's type theory in which there is a type that violates uniqueness of identity proofs. This tells us that dependent pattern matching is a non-conservative extension over Martin-Löf's type theory[1].

As a complement to J, Streicher introduced the induction principle K:

$$\begin{aligned}K\ \ :\ \ &(A : \mathit{Type})\ (x : A)\ (P : \mathit{Id}\ A\ x\ x \rightarrow \mathit{Type})\\ \rightarrow\ &P\ \mathit{refl}\\ \rightarrow\ &(c : \mathit{Id}\ A\ x\ x)\\ \rightarrow\ &P\ c\end{aligned}$$

Using K we can prove the *UIP* property, and the other way around. We have also seen that dependent pattern matching implies K. The converse of this has also been established: we can rewrite definitions written with dependent pattern matching to ones that use only the induction principles and axiom K [Goguen et al., 2006].

In homotopy type theory, we give up K (and essentially dependent pattern matching), to allow for a more interesting structure of propositional equalities.

[1] This actually means that all the code we write, should be written using the elimination principles. Agda provides a `--without-K` flag that limits pattern matches to those cases that should be safe. The assumption is that every definition given by pattern matching that passes the `--without-K` check, can be rewritten using the elimination principles. As such, we will sometimes use pattern matching for our definition.

2.3 Homotopy interpretation

In the introduction (chapter 1), it was mentioned that homotopy type theory concerns itself with the following correspondence:

type theory	**homotopy theory**
A is a type	A is a space
$x, y : A$	x and y are points in A
$p, q : x \equiv y$	p and q are paths from x to y
$w : p \equiv q$	w is a homotopy between paths p and q
$\vdots$	$\vdots$

In section 2.1 we noted that homotopies have a ∞-groupoid structure. It is this structure that leads us to the correspondence between the identity types from Martin-Löf's type theory and homotopy theory. In Hofmann and Streicher [1996], the authors note that types have a groupoid structure. We have a notion of composition of proofs of propositional equality: the term $trans : Id\ A\ x\ y \rightarrow Id\ A\ y\ z \rightarrow Id\ A\ x\ z$, as such we will use the notation $\_ \circ \_$ instead of $trans$. The same goes for $symm : Id\ A\ x\ y \rightarrow Id\ A\ y\ x$, which we will denote as $\_^{-1}$. We can prove that this gives us a groupoid, i.e. we can prove the following laws hold:

Given $a, b, c, d : A$ and $p : a \equiv b$, $p : b \equiv c$ and $q : c \equiv d$ we have:

- Associativity: $p \circ (q \circ r) \equiv (p \circ q) \circ r$
- Left inverses: $p^{-1} \circ p \equiv \mathit{refl}$
- Right inverses: $p \circ p^{-1} \equiv \mathit{refl}$
- Left identity: $\mathit{refl} \circ p \equiv p$
- Right identity: $p \circ \mathit{refl} \equiv p$

The important thing to note is what kind of equalities we are talking about: the equations given above all hold up to propositional equality one level higher. The identity type $Id\ A\ x\ y$ is of course a type and therefore has a groupoid structure of its own. Every type gives rise to a tower of groupoids that can interact with each other: the presence of equations at one level can imply the presence of equations at a higher level. This is exactly the same as the way homotopies form an ∞-groupoid, hence we have the correspondence between types and spaces as mentioned earlier.

Having such an interpretation of type theory brings us several things. Since every proof we write in type theory corresponds to a proof of a statement in homotopy theory, we can use it to proof theorems of homotopy theory.

It also means that the intuition about homotopy theory can be applied to type theory. As such, we can use it to explain why one cannot prove K using J (section 2.3.1), using a couple of illustrations.

2.3.1 Interpreting uniqueness of identity proofs and K

Recall the elimination principle of identity types, J:

$$\begin{aligned}
&J : (A : \mathit{Type}) \\
&\quad \to (x : A) \\
&\quad \to (P : (y : A) \to (p : \mathit{Id}\ A\ x\ y) \to \mathit{Type}) \\
&\quad \to (c : P\ x\ x\ \mathit{refl}) \\
&\quad \to (y : A) \to (p : \mathit{Id}\ A\ x\ y) \\
&\quad \to P\ x\ y\ p
\end{aligned}$$

Interpreting propositional equalities as paths, we see that it tells us that if we want to prove that a predicate P on paths holds, we only have to show that it is satisfied for the constant path *refl*. Homotopically this can be motivated by the fact that P is a predicate on paths with a fixed starting point x and a y that can be chosen freely (see fig. 2.2). Any path $p : x \equiv y$ can be contracted along this path to the constant path $\mathit{refl} : x \equiv x$, so there is a homotopy between these two paths.

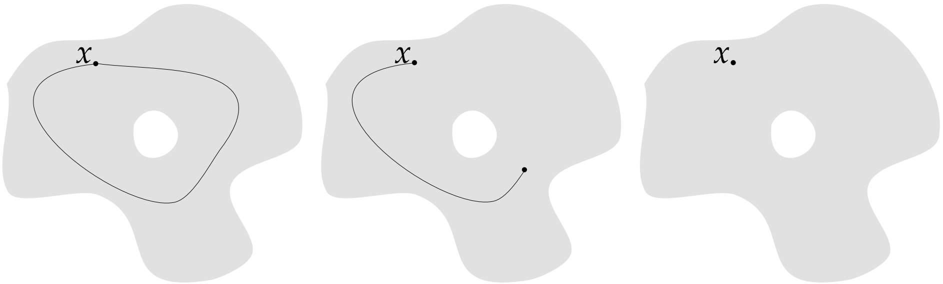

Figure 2.2: With J we have the freedom to move the end point around.

In the case of axiom K, both the beginning and the end point are fixed:

$$\begin{aligned}
&K \ \ : \ \ (A : \mathit{Type})\ (x : A)\ (P : \mathit{Id}\ A\ x\ x \to \mathit{Type}) \\
&\quad \to P\ \mathit{refl} \\
&\quad \to (p : \mathit{Id}\ A\ x\ x) \\
&\quad \to P\ \ c
\end{aligned}$$

Homotopically this means that we are restricted to loops. If we want to contract a given path $p : x \equiv x$ to $\mathit{refl} : x \equiv x$, we cannot use the same trick as with J, as the end point is fixed. Contracting a loop to *refl* does not always work, as can be seen in fig. 2.3. If we have a hole in our space, then we can distinguish between loops that go around the hole and those that do not. This shows that because we can interpret type theory in homotopy theory, we can sometimes use our geometric intuition to answer problems from type theory.

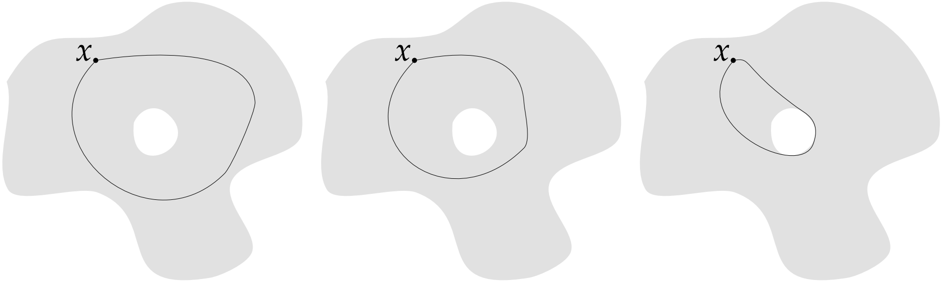

Figure 2.3: With K, we are restricted to loops

2.4 n-types

The tower of iterated identity types of a type can tell us all sorts of things about the type. For example, we can have a tower in which the identity types in a sense become simpler every iteration, until they reach a fixpoint, in which the identity types are isomorphic to the unit type, $\top$. In homotopy theory, spaces isomorphic (or rather, homotopic) to the "unit space", i.e. the space consisting of one point, are called contractible. One way to formulate this in type theory is with the following definition:

$$\begin{array}{l} \mathit{isContractible} : \mathit{Type} \to \mathit{Type} \\ \mathit{isContractible}\ A = \Sigma\ (\mathit{center} : A)\ .\ ((x : A) \to \mathit{Id}\ A\ \mathit{center}\ x) \end{array}$$

This can be interpreted as having a point center such that there is a path from center to any point x. Such an interpretation sounds more like the definition of *path connectedness*. In homotopy theory these two definitions do not coincide: contractibility implies path connectedness, but not the other way around. An example of this is the circle, which is path connected, but not contractible: going around the circle once is not homotopic to the constant loop. The key here is that the only functions that we can define in type theory are *continuous* functions, so $\mathit{isContractible}$ should really be interpreted as there being a point center such that we can construct paths from center to any point x *in a continuous manner*.

If the structure of the identity types peters out after n iterations, we call such a type an $(n-2)$-type, or $(n-2)$-truncated[2]:

$$\begin{array}{l} \mathit{is\text{-}truncated} : \mathbb{N}_{-2} \to \mathit{Type} \to \mathit{Type} \\ \mathit{is\text{-}truncated}\ (-2)\ \ A = \mathit{isContractible}\ A \\ \mathit{is\text{-}truncated}\ (S\ n)\ A = (x\ y : A) \to \mathit{is\text{-}truncated}\ n\ (\mathit{Id}\ A\ x\ y) \end{array}$$

These truncation levels have the property that every n-type is also an $(n+1)$-type, i.e. $\mathit{is\text{-}truncated}$ defines a filtration on the universe of types.

[2] The somewhat strange numbering, starting at -2 comes from homotopy theory, where they first considered groupoids without any higher structure to be 0-truncated and then generalised backwards.

The *contractible* types are the types that are isomorphic to $\top$ in the sense that a contractible type has an inhabitant that is unique up to propositional equality. In section section 2.5 we will see examples of contractible types that have more than one constructor.

Types of truncation level -1 are called h-propositions. (-1)-types are either empty ($\bot$) or, if they are inhabited, contractible, hence isomorphic to $\top$. They can be interpreted as false and true propositions. One can easily prove that h-propositions satisfy the principle of proof irrelevance:

$$\begin{aligned}&\mathit{proofIrrelevance} : \mathit{Type} \to \mathit{Type}\\&\mathit{proofIrrelevance}\ A = (x\ y : A) \to \mathit{Id}\ A\ x\ y\end{aligned}$$

The converse also holds: if a type satisfies proof irrelevance, it is an h-proposition. Showing this is a bit more involved, but it is a nice example of how one can proof things about equalities between equalities.

$$\begin{aligned}&proofIrrelevance{\Rightarrow}is\text{-}proposition : (A : \mathit{Type})\\&\qquad\qquad\to (p : \mathit{proofIrrelevance}\ A) \to is\text{-}hProp\ A\end{aligned}$$

We need to show that for every $x\ y : A$, $x \equiv y$ is contractible: we need to find a proof $c : x \equiv y$ and show that any other proof of $x \equiv y$ is equal to c. An obvious candidate for c is $p\ x\ y$. To show that $c \equiv p\ x\ y$, we use based path induction on c, fixing the y, so we need to prove that $\mathit{refl} \equiv p\ y\ y$. Instead of doing this directly, we first prove something more general:

$$\mathit{lemma} : (x\ y : A)\ (q : x \equiv y) \to p\ x\ y \equiv q\ \circ\ p\ y\ y$$

This can be done by based path induction on q, fixing y. The goal then reduces to showing that $p\ y\ y \equiv p\ y\ y$. Using the lemma we can show that $p\ y\ y \equiv p\ y\ y\ \circ\ p\ y\ y$. Combining this with $p\ y\ y\ \circ\ \mathit{refl}$ and the fact that $\lambda q \to p\ \circ\ q$ is injective for any p, we get that $p\ y\ y \equiv \mathit{refl}$.

The definition of h-proposition via proof irrelevance fits the traditional and classical (in the sense of classical logic) view of propositions and their proofs: we only care about whether or not we have a proof of a proposition and do not distinguish between two proofs of the same proposition.

Another important case are the 0-types, also called *h-sets*, which are perhaps the most familiar to programmers. These are the types of which we have that any two inhabitants x and y are either equal to each other in a unique way, or are not equal, i.e. h-sets are precisely those types that satisfy uniqueness of identity proofs. The simplest example of a type that is an h-set, but not an h-proposition is the type *Bool*:

$$\begin{aligned}&\textbf{data}\ \mathit{Bool} : \mathit{Type}\ \textbf{where}\\&\quad \mathit{True} : \mathit{Bool}\\&\quad \mathit{False} : \mathit{Bool}\end{aligned}$$

In fact, most types one defines in Agda are h-sets. One characteristic of h-sets is given by Hedberg's theorem [Kraus et al., 2013], which states that every type that has decidable equality (i.e. $(x\ y : A) \to x \equiv y + (x \equiv y \to \bot)$) also is an h-set. The only way to define a type that is not an h-set in Agda, is to add extra propositional equalities to the type by adding axioms. This is the subject of section 2.5.

Notation Sometimes we will use the notation $A : hProp$ to indicate that A is a type that is an h-proposition. In an actual implementation $hProp$ would be defined as $\Sigma\ (A : Type)\ .\ (is\text{-}truncated\ (-1)\ A)$. When we refer to A, we are usually not interested in an inhabitant of the Σ-type, but in the first field of that inhabitant, i.e. the $A : Type$. The same holds for the notation $A : hSet$.

2.4.1 Truncations

It may happen that we sometimes construct a type of which the identity types have too much structure, e.g. it is a 2-type but we want it to be a 0-type. In homotopy type theory, we have a way to consider a type as though it were an n-type, for some n we have chosen ourselves, the so called n-truncation of a type. Special cases that are particularly interesting are the (-1)-truncation, i.e. we force something to be an h-proposition, which is particularly useful when we want to do logic, and 0-truncation, i.e. we force something to be an h-set. The idea is that we add enough extra equalities to the type such that the higher structure collapses. This can be done using higher inductive types (section 2.5). The general construction is rather involved and not of much interest for the purposes of this thesis: we will only encounter the (-1)-truncation and 0-truncation.

2.5 Higher inductive types

We have seen a counterexample of a space in which the interpretation of K and uniqueness of identity proofs fails: a space with a hole in it. The question is then if we can construct such counterexamples in the type theory itself. Since we are asking for a type $A : Type$ for which there is an inhabitant $x : A$ with a term $p : Id\ A\ x\ x$ such that $p \equiv refl \rightarrow \bot$, we know that we cannot do this is normal Martin-Löf's type theory without adding axioms as this would violate the canonicity property.

Higher inductive types extend inductive types with the possibility add *path constructors* to the definition of a type: instead of giving constructors for the points of a space, we may also give constructors for paths between points, and paths between paths, and so on. Using higher inductive types we can now describe familiar spaces, such as the circle (see also fig. 2.4):

```
data Circle : Type where
  base : Circle
  loop : base ≡ base
```

The above is of course not (yet) valid Agda syntax. We can simulate higher inductive types by adding the extra path constructors (in this case *loop*) as postulates. We can also hide the constructors in such a way, that we can use them (indirectly) to construct terms of type *Circle*, but without allowing pattern matching.

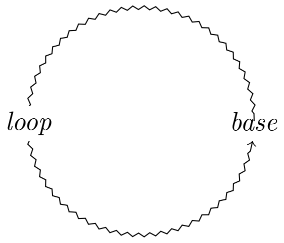

Figure 2.4: The circle as a higher inductive type

Instead of pattern matching, we specify how we can eliminate inhabitants with the principle presented below. Roughly speaking we need to ensure that all the points get mapped in such a way that all the (extra) equalities are respected. In the case of the circle this looks as follows:

$$\begin{aligned}
&\mathit{Circle\text{-}rec} : \{B : \mathit{Set}\}\\
&\quad \to (b : B)\\
&\quad \to (p : b \equiv b)\\
&\quad \to \mathit{Circle} \to B
\end{aligned}$$

with computation rule:

$$\mathit{Circle\text{-}rec}\ b\ p\ \mathit{base} \overset{\Delta}{\equiv} b$$

We also need a computation rule for the paths, to witness that the loop indeed gets mapped onto the specified path $p : b \equiv b$ by ap:

$$\mathit{ap}\ (\mathit{Circle\text{-}rec}\ b\ p)\ \mathit{loop} \overset{\Delta}{\equiv} p$$

It might seem a bit silly that we need to provide a path $b \equiv b$, as this type is always inhabited by refl. However, we sometimes do want p to be different from refl: in order to write the identity function on Circle, we also want loop to be preserved by this map.

Apart from a non-dependent elimination principle, we also need a dependent version:

$$\begin{aligned}
&\mathit{Circle\text{-}ind} : \{B : \mathit{Circle} \to \mathit{Set}\}\\
&\quad \to (b : B\ \mathit{base})\\
&\quad \to (p : \mathit{transport}\ B\ \mathit{loop}\ b \equiv b)\\
&\quad \to (x : \mathit{Circle}) \to B\ x
\end{aligned}$$

Using the dependent elimination principle, we can show that this type violates uniqueness of identity proofs, i.e. we can prove that $\mathit{loop} \equiv \mathit{refl}$ does not hold. In fact, the type $\mathit{Id}\ \mathit{Circle}\ \mathit{base}\ \mathit{base}$ is isomorphic to the integers $\mathbb{Z}$, where transitivity maps to addition on integers [Licata and Shulman, 2013]. This might seem a bit strange, because at first glance Circle seems to be a contractible type: we have only have one constructor base and an equality $\mathit{base} \equiv \mathit{base}$, so it seems to fit the definition. However, trying to prove $(x : \mathit{Circle}) \to x \equiv \mathit{base}$ will not work, as the only functions we can define in type theory are *continuous* functions. While it is true in homotopy theory that for every point on the circle, we can find a path to the base point, we cannot do so in a continuous way.

If we add a path constructor connecting two points x and y, we do not only get that specific path, but all the paths that can be constructed from that path using transitivity and symmetry. If we start out with a type with only two constructors x and y, we get a type isomorphic to the booleans (see fig. 2.5), a 0-type. Adding one path constructor $p : x \equiv y$ gives us the interval (see fig. 2.6 and section 2.5.2), which is a contractible type (it is a (-2)-type) and hence isomorphic to the unit type $\top$. If we add yet another path constructor $q : x \equiv y$, we get a type isomorphic to *Circle*, which is a 1-type.

Figure 2.5: Booleans

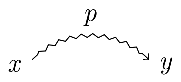

Figure 2.6: Interval

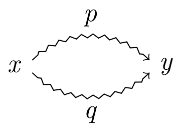

Figure 2.7: Circle

2.5.1 Coherence issues

Equalities at different levels interact with each other: if we add equalities at one level, e.g. paths between points, it may also generate new paths at other levels, e.g. new homotopies between paths that previously did not exist. One example of this is (-1)-truncation, or propositional truncation, via the following higher inductive type:

```
data (−1)-truncation : (A : Type) : Type where
  inhabitant : A → (−1)-truncation A
  all-paths : (x y : (−1)-truncation A) → x ≡ y
```

We have seen in section 2.4 that this type indeed yields a proposition, as it satisfies proof irrelevance, since we have added paths between all points x and y. This collapses all higher structure of the original type $A : \mathit{Type}$.

The converse can also happen: instead of collapsing the structure at higher levels, we might gain new structure at those levels, which sometimes may be undesirable. If this is the case, the resulting type is not coherent enough. *Coherence properties* are properties that state that certain equalities between equalities must hold. Suppose we want to consider words generated by some alphabet $A : \mathit{Set}$. This can be done with the following type:

```
data FreeSemigroup (A : Set) : Type where
  elem : A → FreeSemigroup A
  _·_ : FreeSemigroup A → FreeSemigroup A → FreeSemigroup A
```

Clearly, *FreeSemigroup* A is a set. Suppose we want $\_\cdot\_$ to be associative, so we add the following path constructor:

$$\mathit{assoc} : \{ a\ b\ c : \mathit{FreeSemigroup}\ A \} \to (a \cdot b) \cdot c \equiv a \cdot (b \cdot c)$$

Adding these equalities breaks the h-set property. The coherence property that we want to hold here is uniqueness of identity proofs. One example for which this fails, is that the following diagram (the so called *Mac Lane pentagon*) does not commute:

$$\begin{tikzcd}
((a \cdot b) \cdot c) \cdot d \arrow[r, rightsquigarrow, "ap\ (\lambda x \to x \cdot d)\ assoc"] \arrow[d, rightsquigarrow, "assoc"'] & (a \cdot (b \cdot c)) \cdot d \arrow[r, rightsquigarrow, "assoc"] & a \cdot ((b \cdot c) \cdot d) \arrow[d, rightsquigarrow, "ap\ (\lambda x \to a \cdot x)\ assoc"] \\
(a \cdot b) \cdot (c \cdot d) \arrow[rr, rightsquigarrow, "assoc"'] & & a \cdot (b \cdot (c \cdot d))
\end{tikzcd}$$

This shows us that the interaction of propositional equalities at the different levels can be quite subtle. For this reason one often truncates a higher inductive type, to be sure that it is coherent enough, e.g. that it is really an h-set.

2.5.2 Interval

Another example of a space from homotopy theory is the interval. At first glance this might seem like a rather uninteresting space to study, as it is homotopy equivalent to the space that consists of one point. The following presentation of the interval as a higher inductive type has some interesting consequences.

The interval $[0, 1]$ can be seen, from a homotopy theory perspective, as a space with two points, 0 and 1, and a path between them. As a higher inductive type, this can be presented as follows:

```
data Interval : Type where
  zero : Interval
  one  : Interval
  segment : zero ≡ one
```

A map from *Interval* to some type $B : \mathit{Type}$ must map *zero* and *one* to points in $a\ b : B$ such that $a \equiv b$:

```
Interval-rec : {B : Type}
  → (b₀ b₁ : B)
  → (p : b₀ ≡ b₁)
  → Interval → B
```

with computation rules:

$$\begin{aligned}
&\textit{Interval-rec}\; b_0\; b_1\; p\; \textit{zero} \triangleq b_0\\
&\textit{Interval-rec}\; b_0\; b_1\; p\; \textit{one}\; \triangleq b_1\\
&\textit{ap}\; (\textit{Interval-rec}\; b_0\; b_1\; p)\; \textit{seg} \triangleq p
\end{aligned}$$

Having an interval type means that we have a different way to talk about equalities: any path $p : \textit{Id}\ A\ x\ y$ can be seen as a map $\textit{Interval} \to A$:

$$\begin{aligned}
&\equiv\, \Rightarrow\! \textit{Interval} : \{A : \textit{Type}\}\ \{x\ y : A\} \to x \equiv y \to \textit{Interval} \to A\\
&\equiv\, \Rightarrow\! \textit{Interval}\ \{A\}\ \{x\}\ \{y\}\ p\ i = \textit{Interval-rec}\ \{A\}\ x\ y\ p\ i
\end{aligned}$$

The other way around can also be done:

$$\begin{aligned}
&\textit{Interval}\!\Rightarrow \equiv\ : \{A : \textit{Set}\} \to (p : \textit{Interval} \to A) \to (p\ \textit{zero}) \equiv (p\ \textit{one})\\
&\textit{Interval}\!\Rightarrow \equiv\ \ p = \textit{ap}\ p\ \textit{seg}
\end{aligned}$$

Using this we can now manipulate propositional equalities in such a way that we can prove function extensionality. Suppose two functions $f\ g : A \to B$ and a term $\alpha : (x : A) \to f\ x \equiv g\ x$. To remove the dependency in the type, we can use $\equiv\, \Rightarrow\! \textit{Interval}$:

$$\lambda a \to \equiv\, \Rightarrow\! \textit{Interval}\ (\alpha\ a) : A \to \textit{Interval} \to A$$

If we flip the arguments of that term, we get a function $\textit{Interval} \to A \to A$, which then can be turned into the desired $f \equiv g$. The whole term looks as follows:

$$\begin{aligned}
&\textit{ext} : (A\ B : \textit{Type})\ (f\ g : A \to B)\ (\alpha : (x : A) \to f\ x \equiv g\ x) \to f \equiv g\\
&\textit{ext}\ A\ B\ f\ g\ \alpha = \textit{Interval}\!\Rightarrow \equiv\ \ (\textit{flip}\ (\lambda a \to \equiv\, \Rightarrow\! \textit{Interval}\ (\alpha\ a)))
\end{aligned}$$

2.6 Equivalence and univalence

Martin-Löf's type theory satisfies the property that everything you construct in the theory is invariant under isomorphism. Consider for example the definition of a monoid:

$$\begin{aligned}
&\textit{Monoid} : \textit{Type} \to \textit{Type}\\
&\textit{Monoid}\ A =\\
&\quad \Sigma\ (\textit{unit} \quad\quad\ : A)\ \ .\\
&\quad \Sigma\ (\_\cdot\_ \quad\quad\ \ : A \to A \to A)\ \ .\\
&\quad \Sigma\ (\textit{assoc} \quad\ \ \ : (x\ y\ z : A) \to x \cdot (y \cdot z) \equiv (x \cdot y) \cdot z)\ \ .\\
&\quad \Sigma\ (\textit{unitleft} \ \ : (x : A) \to \textit{unit} \cdot x \equiv x)\ \ .\\
&\quad \Sigma\ (\textit{unitright} : (x : A) \to x \cdot \textit{unit} \equiv x)\ \ .\ \top
\end{aligned}$$

If we have two types $A\ \ B : \mathit{Type}$ with an isomorphism $f : A \rightarrow B$ and a proof $ma : \mathit{Monoid}\ \ A$, then it is straightforward to produce a $\mathit{Monoid}\ \ B$ using only $\mathit{Monoid}\ \ A$ and the isomorphism f, by applying f and $f^{\ -1}$ to the fields of $ma : \mathit{Monoid}\ \ A$. The resulting instance of $\mathit{Monoid}\ \ B$ can then also be shown to be isomorphic to ma. This is similar to the situation with *transport* and *apd*: if we have a proof $p : A \equiv B$, then we can use *transport* to create an inhabitant of $\mathit{Monoid}\ \ B$ using ma and p. We can then prove that the resulting instance of $\mathit{Monoid}\ \ B$ is propositionally equal to ma using *apd*. However, writing *transport* and *apd* functions that works with isomorphisms instead of propositional equalities will not work in Martin-Löf's type theory. If we try to write the following functions:

$$
\begin{aligned}
&\mathit{transport\text{-}iso} : \{A : \mathit{Type}\}\ \{B : A \rightarrow \mathit{Type}\}\ \{x\ y : A\} \\
&\qquad\qquad \rightarrow x\ \simeq\ y \rightarrow B\ x \rightarrow B\ y \\
&\mathit{apd\text{-}iso} : \{A : \mathit{Type}\}\ \{B : A \rightarrow \mathit{Type}\}\ \{x\ y : A\} \\
&\qquad \rightarrow (f : (a : A) \rightarrow B\ \ a) \rightarrow (\beta : x\ \simeq\ y) \\
&\qquad \rightarrow \mathit{transport}\ \beta\ (f\ \ x)\ \simeq\ f\ y
\end{aligned}
$$

we will find ourselves stuck. We want to write a type-generic program that applies the isomorphism at the right places, but we cannot access the information about how the types are constructed.

Univalence gives us an internal account of the principle that everything we construct is invariant under isomorphism. It roughly says that isomorphic types are propositionally equal, so all the tools to manipulate propositional equalities now also can be applied to isomorphisms. But before we can formulate the univalence axiom, we need to introduce some new terminology. We can define the notion of a function $f : A \rightarrow B$ being an isomorphism as follows:

$$
\begin{aligned}
&\mathit{isIsomorphism} : \{A\ B : \mathit{Type}\}\ (f : A \rightarrow B) \rightarrow \mathit{Type} \\
&\mathit{isIsomorphism}\ f = \Sigma\ (g : B \rightarrow A)\ \ .\ \ ((x : B) \rightarrow f\ (g\ x) \equiv x\ \ \times \\
&\quad (x : A) \rightarrow g\ (f\ x) \equiv x) \\
&\_\simeq\_ : (A\ B : \mathit{Type}) \rightarrow \mathit{Type} \\
&A\ \simeq\ B = \Sigma\ (f : A \rightarrow B)\ (\mathit{isIsomorphism}\ f)
\end{aligned}
$$

We want the type $\mathit{isIsomorphism}\ f$ to be an h-proposition, which it is when A and B are h-sets, but it can fail to be an h-proposition when A and B are n-types with $n > 0$. Instead we introduce the notion of *equivalence* :

$$
\begin{aligned}
&\mathit{isEquivalence} : \{A\ B : \mathit{Type}\}\ (f : A \rightarrow B) \rightarrow \mathit{Type} \\
&\mathit{isEquivalence}\ f = \Sigma\ (g : B \rightarrow A)\ \ .\ \ ((x : B) \rightarrow f\ (g\ x) \equiv x) \\
&\quad \times \qquad\qquad\quad\ \ \Sigma\ (g : B \rightarrow A)\ \ .\ \ ((x : A) \rightarrow h\ (f\ x) \equiv x)
\end{aligned}
$$

This definition does satisfy the property that $\mathit{isEquivalence}\ f$ can hold in at most one way (up to propositional equality). We can also show that $\mathit{isIsomorphism}\ f \rightarrow \mathit{isEquivalence}\ \ f$ and $\mathit{isEquivalence}\ \ f\ \rightarrow\ \mathit{isIsomorphism}\ \ f$, i.e. the two types are *coinhabited*.

Using this definition of what it means to be an equivalence, we can define the following relation on types, analogous to what we did with isomorphisms:

$$
\begin{aligned}
&\_\cong\_ : (A\ B : \mathit{Type}) \rightarrow \mathit{Type} \\
&A\ \cong\ B = \Sigma\ (f : A \rightarrow B)\ \ .\ \ (\mathit{isEquivalence}\ f)
\end{aligned}
$$

It is easy to show that if two types are propositional equal, then they are also equivalent, by transporting along $\lambda X \rightarrow X$:

$$\equiv\Rightarrow\simeq\ : (A\ B : \mathit{Type}) \rightarrow A \equiv B \rightarrow A\ \cong\ B$$

A universe of types is called a *univalent* universe if equivalences and propositional equalities are equivalent, e.g. in the case of the universe Type, this would look as follows:

$$(A\ B : \mathit{Type}) \rightarrow \mathit{isEquivalence}\ (\equiv\Rightarrow\simeq\ A\ B)$$

It has been shown that in a popular model of homotopy theory, the category of simplicial sets, the universe of spaces is indeed univalent [Kapulkin et al., 2012]. One important consequence of this property is that we have the following:

$$\mathit{univalence} : (A\ B : \mathit{Type}) \rightarrow A\ \cong\ B \rightarrow A \equiv B$$

which should satisfy the following computation rule:

$$\begin{aligned} &\mathit{uacomp} : \{A\ B : \mathit{Type}\} \\ &\qquad \{f : A \rightarrow B\} \\ &\qquad \{\mathit{eq} : \mathit{isEquivalence}\ f\} \\ &\qquad \{x : A\} \\ &\quad \rightarrow \mathit{transport}\ (\lambda X \rightarrow X)\ (\mathit{univalence}\ A\ B)\ x \equiv f\ x \end{aligned}$$

Univalence means that we can now generalise the Monoid example mentioned to any $B : \mathit{Type} \rightarrow \mathit{Type}$, since $\mathit{transport}$ and apd can now be used for isomorphisms as well.

If we have univalence, the universe of h-sets is not a h-set, as is exhibited by the isomorphisms $\mathit{Bool} \rightarrow \mathit{Bool}$. There are two different such isomorphisms: id and not. Using $\mathit{univalence}$, these isomorphisms map to different proofs of $\mathit{Bool} \equiv \mathit{Bool}$. id maps to refl and not to something that is not equal to refl. This means that the universe of h-sets violates uniqueness of identity proofs. It can be shown to be a 1-type instead. In fact, the universe of n-types is not an n-type but an $(n+1)$-type [Kraus and Sattler, 2013].

2.7 Implementation

Currently, the way to "implement" homotopy type theory, i.e. Martin-Löf's type theory with univalence and higher inductive types, is to take an existing implementation of Martin-Löf's type theory such as Agda or Coq and add univalence and the computation rules for univalence as axioms. This approach is sufficient when we want to do formal mathematics, since in that case we only are interested in type checking our developments. If we want to run the program, terms that make use of univalence then get stuck as soon as it hits an axiom.

The computational interpretation of univalence is one the biggest open problems of homotopy type theory. Several attempts have been made at a computation interpretation for truncated versions of homotopy type theory: Licata and Harper [2012] show that if we restrict ourselves to a univalent universe of h-sets, we can achieve canonicity. The article however does not present a decidability result for type checking. Sozeau et al. [2013] internalise homotopy type theory in Coq and also restrict themselves to the two-dimensional case, i.e. uniqueness of identity proofs need not hold, but equalities between equalities are unique.

A question one might ask is why we cannot add an extra constructor to the definition of Id for univalence. Doing this means that we end up with a different elimination principle: if we want to prove something about propositional equalities, we also need to account for the case when it was proven using univalence. Apart from making it more difficult to prove things about propositional equalities, it is also has some undesirable properties. We can prove that a proof of equality constructed using the univalence constructor is never equal to refl. There are cases in which we want this to be the case, e.g. when we apply univalence to the identity isomorphism.

The conjecture is that full canonicity will probably not hold, but only canonicity "up to propositional equality": it is conjectured [Voevodsky, 2011] that there is a terminating algorithm that takes an expression $t : \mathbb{N}$ and produces a canonical term $t' : \mathbb{N}$ along with a proof that $t \equiv t'$. The proof of equality may use the univalence axiom.

Higher inductive types can also be implemented by adding axioms for the extra paths. The elimination principles also can be implemented by adding the computation rules for paths as axioms. One then has to be careful not to pattern match on higher inductive types. In Agda one can hide things in such a way that one can export an elimination principle in which the computation rules for the points hold definitionally and the other rules propositionally, while also making direct pattern matching impossible from any other module that imports the module containing the higher inductive type [Licata, 2011]. However, one still has to be careful not to use the absurdity pattern, $()$, when dealing with higher inductive types, as that can be used to prove $\bot$ [Danielsson, 2012].

Chapter 3

Applications of homotopy type theory

In chapter 2 we introduced homotopy type theory and the two extensions to Martin-Löf's type theory it brings us: univalence and higher inductive types. We have seen how higher inductive types can be used to prove function extensionality and how univalence makes it a lot easier to deal with isomorphic types in programs. This chapter is devoted to other applications of homotopy type theory to programming. In section 3.1 we show how higher inductive types can be used to define quotient types and argue whether we need such a construction and contrast this approach to the setoid approach. We consider some of the difficulties that higher inductive types usually bring with them (so called coherence issues) and show how to write binary operations on quotients as an example of how one uses the elimination principles of quotients.

We also consider the application of univalence to views on abstract types (section 3.2), as proposed by Licata [2012]. We work out the computations in detail to show how this works out and extend the approach to also work with non-isomorphic views. The resulting construction is a nice application of quotient types.

3.1 Quotient types

In mathematics, one way to construct new sets is to take the *quotient* of a set X by an equivalence relation R on that particular set. The new set is formed by regarding all elements $x, y \in X$ such that xRy as equal. An example of a quotient set is the set of rationals $\mathbb{Q}$ constructed from the integers as follows: we quotient out $\mathbb{Z} \times \mathbb{Z}$ by the relation $(a, b) \sim (c, d)$ if and only if $ad = bc$.

In programming, such a construction can also be very useful, as it often happens that we have defined a data type that has more structure than we want to expose via the interface. An example of this is encoding sets as lists: we want to regard two lists as encoding the same set if they contain the same elements, no matter what the multiplicity of every element is or how the list is ordered. Other examples where quotient types can be useful are the situations in which we want to encode our data in such a way that certain operations on the data can be implemented more efficiently. An example of this is implementing a dictionary with a binary search tree: there are multiple binary search trees that represent the same dictionary, i.e. contain the same key-value pairs. If we pass two different trees representing the same dictionary to an operation, we want the operation to yield the same results.

To make the above more precise, suppose we have defined a data type of binary search trees, $BST : Type$, along with a relation $\_{\sim}\_ : BST \to BST \to hProp$ such that $x \sim y \equiv \top$ if and only if x and y are comprised of the same key-value pairs, and $x \sim y \equiv \bot$ otherwise. Suppose we have an insertion operation $insert$ of type $KeyValuePair \to BST \to BST$ and a lookup function $lookup : Key \to BST \to Maybe\ Value$. We can formulate the properties that should hold:

- $(a : KeyValPair)\ (x\ y : BST) \to x \sim y \to insert\ a\ x \sim insert\ a\ y$
- $(a : Key)\ (x\ y : BST) \to x \sim y \to lookup\ a\ x \equiv lookup\ a\ y$

Note that for insertion, returning the same results means that we want them to represent the same dictionary: it is perfectly allowed to return differently balanced binary search trees. For $lookup$, we want the results to be propositionally equal, as we do not have any other relation available that holds on the result type, $Maybe\ Value$.

A type that comes equipped with an equivalence relation, such as BST along with $\_{\sim}\_$, is called a *setoid*. Its disadvantages are that we have to formulate and check the properties ourselves: there is no guarantee that a function out of a setoid respects the relation from the setoid. As can be seen in the binary search tree example, we have to be careful to use the right relation (propositional equality or the setoid's equivalence relation) when we want to talk about two inhabitants being the same. Homotopy type theory provides us with the machinery, namely higher inductive types, to enrich the propositional equality of a type, so we can actually construct a new type in which propositional equality and the provided equivalence relation coincide.

3.1.1 Do we need quotients?

Before we look at the quotient type construction with higher inductive types, we will determine whether we actually need such a thing. In the case of the dictionary example, we might consider making the BST data type more precise such that the only inhabitants are trees that are balanced in a certain way, e.g. by using a cleverly indexed type, so we do have a unique representation for every dictionary.

The question then is whether such a construction always exists: can we define a type that is in some sense equal to the quotient? To be able to answer this question, we need to define what it means to be a quotient and what notion of equality we want.

Altenkirch et al. define a quotient, given a setoid $(A, \_\!\sim\!\_)$ as a type $Q : Type$ with the following:

- a projection function $[\_] : A \to Q$
- a function $sound : (x\ y : A) \to x \sim y \to [x] \equiv [y]$
- an elimination principle:

$$\begin{aligned} Q\text{-}elim : &(B : Q \to Type) \\ &(f : (x : A) \to B\ [x]) \\ &((x\ y : A)\ (p : x \sim y) \to (transport\ (sound\ x\ y\ p)\ (f\ x)) \equiv f\ y) \\ &(q : Q) \to B\ q \end{aligned}$$

A quotient is called definable if we have a quotient Q along with the following:

- $emb : Q \to A$
- $complete : (a : A) \to emb\ [a] \sim a$
- $stable : (q : Q) \to [emb\ q] \equiv q$

We can view these requirements as having a proof of $[\_]$ being an isomorphism, with respect to the relation $\_\!\sim\!\_$ on A instead of propositional equality.

The result of Altenkirch et al. is that there exist quotients that are not definable with one example being the real numbers constructed using the usual Cauchy sequence method. Adding quotients as higher inductive types to our type theory, does not make the real numbers definable. Adding quotients is still useful in that we only have to work with propositional equality, as opposed to the confusion as to what relation one should use that arises from the use of setoids.

3.1.2 Quotients as a higher inductive type

Using higher inductive types, we can define the quotient of a type by a relation as follows:

$$\begin{aligned} &\textbf{data}\ Quotient\ (A : Type)\ (\_\!\sim\!\_ : A \to A \to hProp) : Type\ \textbf{where} \\ &\quad [\_] : A \to Quotient\ A\ \_\!\sim\!\_ \\ &\quad sound : (x\ y : A) \to x \sim y \to [x] \equiv [y] \end{aligned}$$

To write a function $Quotient\ A\ \_\!\sim\!\_ \to B$ for some $B : Type$, we need to specify what this function should do with values $[x]$ with $x : A$. This needs to be done in such a way that the paths added by $sound$ are preserved. Hence the recursion principle lifts a function $f : A \to B$ to $\widetilde{f} : Quotient\ A\ \_\!\sim\!\_ \to B$ given a proof that it preserves the added paths:

$$\begin{aligned} Quotient\text{-}rec : &(A : Type)\ (\_\!\sim\!\_ : A \to A \to hProp) \\ &(B : Type) \\ &(f : A \to B) \\ &((x\ y : A) \to x \sim y \to f\ x \equiv f\ y) \\ &Quotient\ A\ \_\!\sim\!\_ \to B \end{aligned}$$

If we generalise this to the dependent case, we get something that fits perfectly in the requirement of a type being a quotient given earlier:

$$\begin{aligned}\mathit{Quotient\text{-}ind} : {} & (A : \mathit{Type})\ (\_\!\sim\!\_ : A \to A \to \mathit{hProp}) \\ & (B : \mathit{Quotient}\ A\ \_\!\sim\!\_ \to \mathit{Type}) \\ & (f : (x : A) \to B\ [x]) \\ & ((x\ y : A)\ (p : x \sim y) \to (\mathit{transport}\ (\mathit{sound}\ x\ y\ p)\ (f\ x)) \equiv f\ y) \\ & (q : \mathit{Quotient}\ A\ \_\!\sim\!\_) \to B\ q\end{aligned}$$

Note that we do not require a proof of $\_\!\sim\!\_$ being an equivalence relation. Instead, the quotient should be read as identifying inhabitants by the smallest equivalence relation generated by $\_\!\sim\!\_$: adding the path constructor means we get $x \equiv y$ for every $x \sim y$, but also paths constructed from these new paths using *trans* and *sym*, possibly in combination with paths that already existed.

3.1.3 Coherence issues

One thing we glossed over is the question whether $\mathit{Quotient}\ A\ \_\!\sim\!\_$ is actually an h-set, given the fact that A is an h-set. This need not be the case, as is exhibited by the case where A is taken to be $\top$ and $\_\!\sim\!\_$ is the trivial relation. The resulting quotient is equivalent to the circle, which is not an h-set: the loop $\mathit{sound}\ \mathit{tt}\ \mathit{tt}\ \mathit{tt} : [\mathit{tt}] \equiv [\mathit{tt}]$ is not equal to $\mathit{refl} : [\mathit{tt}] \equiv [\mathit{tt}]$.

In order to get an h-set, we therefore need to take the 0-truncation of the quotient, which can be done with the following higher inductive type:

$$\begin{aligned}& 0\text{-}\mathit{truncation}\ (A : \mathit{Type}) : \mathit{Type}\ \mathbf{where} \\ & \quad \mathit{inhabitant} : A \to 0\text{-}\mathit{truncation}\ A \\ & \quad \mathit{uip} : \{x\ y : 0\text{-}\mathit{truncation}\ A\} \to (p\ q : x \equiv y) \to p \equiv q\end{aligned}$$

The elimination principle tells us that any function $A \to B$, with $A\ B : \mathit{Type}$ can be lifted to $0\text{-}\mathit{truncation}\ A \to B$ if it respects the additional paths of $0\text{-}\mathit{truncation}\ A$. If B happens to be an h-set, then these conditions are automatically satisfied. In the dependent case, we have to supply a family of types $B : 0\text{-}\mathit{truncation}\ A \to \mathit{Type}$ and a function $f : (x : A) \to B\ (\mathit{inhabitant}\ x)$ such that, again, the additional paths of $0\text{-}\mathit{truncation}\ A$ are respected. If we have that for every $x : 0\text{-}\mathit{truncation}\ A$, $B\ x$ is an h-set, then we are done. The precise formulations of the elimination principles, both dependent and non-dependent, are rather technical and involved and not of interest for our purposes. In the examples we consider, we eliminate into h-sets, so we do not need to explicitly check the additional conditions.

If the relation $\_\!\sim\!\_$ happens to be an equivalence relation, using the truncated quotient also gives us that we have (using univalence) $a \sim b \equiv ([a] \equiv [b])$, for every $a\ b : A$, supporting our previous statement that the propositional equality of $\mathit{Quotient}\ A\ \_\!\sim\!\_$ is the smallest equivalence relation generated by $\_\!\sim\!\_$.

3.1.4 Binary operations on quotients

We have seen how to lift a function $f : A \to B$ to $\widetilde{f} : \mathit{Quotient}\ A\ \_{\sim}\_\ B$ given a proof of $(x\ y : A) \to x \sim y \to f\ x \equiv f\ y$, using *Quotient-rec*. Suppose we want to write a binary operation on quotients, then we want to have a way to lift a function $f : A \to A \to B$ satisfying $(x\ y\ x'\ y' : A) \to x \sim x' \to y \sim y' \to f\ x\ y \equiv f\ x'\ y'$ to $\widetilde{f} : \mathit{Quotient}\ A\ \_{\sim}\_ \to \mathit{Quotient}\ A\ \_{\sim}\_ \to B$.

Let us fix A, $\_{\sim}\_$ and B, so that we do not have to pass them around explicitly. Our goal is to write a term of the following type:

$$\begin{aligned}\mathit{Quotient\text{-}rec}\text{-}2 : {}& (f : A \to A \to B)\\ & (\mathit{resp} : (x\ y\ x'\ y' : A) \to x \sim x' \to y \sim y' \to f\ x\ y \equiv f\ x'\ y')\\ & \mathit{Quotient}\ A\ \_{\sim}\_ \to \mathit{Quotient}\ A\ \_{\sim}\_ \to B\end{aligned}$$

We will first use *Quotient-rec* to lift the left argument, i.e. we want to produce a function of type $\mathit{Quotient}\ A\ \_{\sim}\_ \to A \to B$ and then use *Quotient-rec* on this function to achieve our goal. So let us try writing the function that lifts the left argument:

$$\begin{aligned}&\mathit{lift\text{-}left} : (f : A \to A \to B)\\ &\quad (\mathit{resp} : (x\ y\ x'\ y' : A) \to x \sim x' \to y \sim y' \to f\ x\ y \equiv f\ x'\ y')\\ &\quad \mathit{Quotient}\ A\ \_{\sim}\_ \to A \to B\\ &\mathit{lift\text{-}left}\ f\ \mathit{resp}\ q = \mathit{Quotient\text{-}rec}\ f\ \mathit{goal}_0\ q\end{aligned}$$

where $\mathit{goal}_0 : (x\ x' : A) \to x \sim x' \to f\ x \equiv f\ x'$. Since we have quotient types, we also have function extensionality[1], hence we can solve this by proving $(x\ x'\ y : A) \to x \sim x' \to f\ x\ y \equiv f\ x'\ y$. However, to be able to use *resp*, we also need a proof of $y \sim y$, so if we assume that $\_{\sim}\_$ is an equivalence relation, we can solve this goal.

We can now fill in $\mathit{lift\text{-}left}$ in the definition of $\mathit{Quotient\text{-}rec}\text{-}2$:

$$\mathit{Quotient\text{-}rec}\text{-}2\ f\ \mathit{resp}\ q\ q' = \mathit{Quotient\text{-}rec}\ (\mathit{lift\text{-}left}\ f\ \mathit{resp}\ q)\ \mathit{goal}_1\ q'$$

where $\mathit{goal}_1 : (y\ y' : A) \to y \sim y' \to \mathit{lift\text{-}left}\ f\ \mathit{resp}\ q\ y \equiv \mathit{lift\text{-}left}\ f\ \mathit{resp}\ q\ y'$, which can be proven using *Quotient-ind*. We then only have to consider the case where q is of the form $[\,a\,]$ for some $a : A$. In that case, $\mathit{lift\text{-}left}\ f\ \mathit{resp}\ q\ y$ reduces to $f\ a\ y$ and $\mathit{lift\text{-}left}\ f\ \mathit{resp}\ q\ y'$ to $f\ a\ y'$. Since we have $y \sim y'$, we again need $\_{\sim}\_$ to be reflexive to get $a \sim a$ so we can use *resp*. We now have the following:

$$\begin{aligned}&\mathit{goal}_1 : (y\ y' : A) \to y \sim y' \to \mathit{lift\text{-}left}\ f\ \mathit{resp}\ q\ y \equiv \mathit{lift\text{-}left}\ f\ \mathit{resp}\ q\ y'\\ &\mathit{goal}_1 = \lambda y\ y'\ r \to\\ &\quad \mathit{Quotient\text{-}ind}\ (\lambda w \to \mathit{lift\text{-}left}\ f\ \mathit{resp}\ w\ y \equiv \mathit{lift\text{-}left}\ f\ \mathit{resp}\ w\ y')\\ &\qquad\qquad\qquad\quad (\lambda a \to \mathit{resp}\ a\ y\ a\ y'\ ({\sim}\text{-}\mathit{refl}\ a)\ r)\\ &\qquad\qquad\qquad\quad \mathit{goal}_2\\ &\qquad\qquad\qquad\quad q\end{aligned}$$

[1] We can quotient *Bool* by the trivial relation. Using this, we can perform essentially the same proof of function extensionality as the one that uses the interval type.

Of course, we have still to prove that this respects the quotient structure on q:

$$\begin{array}{l} goal_2 : (p : x \sim x') \\ \quad transport\ (sound\ x\ x'\ p)\ (resp\ x\ y\ x\ y'\ (\sim\text{-}refl\ x)\ r) \equiv \\ \quad resp\ x'\ y\ x'\ y'\ (\sim\text{-}refl\ x')\ r \end{array}$$

Note that this equality is of type $Id\ (Id\ B\ (f\ x\ y)\ (f\ x\ y'))$, which means that if B happens to be an h-set, we can appeal to uniqueness of identity proofs and we are done.

It is interesting to see that even though we do not need $\_\sim\_$ to be an equivalence relation for the definition of quotient to work, we do find ourselves in need of properties such as reflexivity for $\_\sim\_$, in order to define operations on quotients.

3.2 Views on abstract types

Consider the dictionary example of the previous section. Most languages provide such a structure as an *abstract type*, e.g. in the Haskell Platform, a dictionary structure is provided by the `Data.Map` module. To the users importing this module, the type Map is opaque: its constructors are hidden. The user may only use the operations such as $insert$ and $lookup$. The advantage of this approach is that we can easily interchange an obvious but slow implementation (e.g. implementing a dictionary as a list of tuples) with a more efficient but more complex solution (e.g. using binary search trees instead of lists), without having to change a single line of code in the modules using the abstract type.

In dependently typed programming, such an approach often means that we have hidden too much: as soon as we try to prove properties about our program that uses some abstract type, we find ourselves having to add properties to the abstract type specification, or even worse: we end up exporting everything so we can use induction on the concrete type used in the actual implementation.

A solution to this problem is to supply the abstract type along with a concrete implementation of the abstract type, called a *view*. This approach was introduced by Wadler [1987] as a way to do pattern matching on abstract types.

3.2.1 Specifying views

An implementation of an abstract type is a type along with a collection of operations on that type. An abstract type can then be described in type theory as a nested Σ-type [Mitchell and Plotkin, 1988], e.g. a sequence abstract type can be described as follows:

$$\begin{array}{lllll} \mathit{Sequence} = & \Sigma\,(\mathit{seq} & : \mathit{Set} \to \mathit{Set}) & & . \\ & \Sigma\,(\mathit{empty} & : (A : \mathit{Set}) \to (\mathit{seq}\ A)) & & . \\ & \Sigma\,(\mathit{single} & : (A : \mathit{Set}) \to A \to \mathit{seq}\ A) & & . \\ & \Sigma\,(\mathit{append} & : (A : \mathit{Set}) \to \mathit{seq}\ A \to \mathit{seq}\ A \to \mathit{seq}\ A) & & . \\ & \quad(\mathit{map} & : (A\ B : \mathit{Set}) \to (A \to B) \to \mathit{seq}\ A \to \mathit{seq}\ B) & & \end{array}$$

An implementation of such an abstract type then is just an inhabitant of this nested Σ-type.

If we want to do more than just use the operations and prove properties about our programs that make use of abstract types, we often find that we do not have enough information in the abstract type specification available to prove the property at hand. One way to address this problem is to add properties to the specification, but it might not at all be clear a priori what properties are interesting and expressive enough to add to the specification.

Another solution, proposed by Licata [2012], is to use views: along with nested Σ-type, we also provide a concrete implementation, i.e. an inhabitant of said Σ-type, called a *view* on the abstract type. The idea is that the concrete view can be used to prove theorems about the abstract type. However, for this to work, we need to make sure that any implementation of the abstract type is also in some sense compatible with the view: the types of both implementations need to be isomorphic and the operations need to respect the isomorphism. To illustrate this, consider we have two sequence implementations:

$$\begin{array}{l} \mathit{ListImpl} : \mathit{Sequence} \\ \mathit{ListImpl} = (\mathit{List}, ([\,], (\lambda x \to [x], (\_{+\!\!+}\_, \mathit{map})))) \\ \mathit{OtherImpl} : \mathit{Sequence} \\ \mathit{OtherImpl} = (\mathit{Other}, (\mathit{otherEmpty}, (\mathit{otherSingle}, (\mathit{otherAppend}, \mathit{otherMap})))) \end{array}$$

We want List and Other to be "isomorphic"[2], i.e. we need to write the following terms:

- $\mathit{to} : (A : \mathit{Type}) \to \mathit{Other}\ A \to \mathit{List}\ A$
- $\mathit{from} : (A : \mathit{Type}) \to \mathit{List}\ A \to \mathit{Other}\ A$
- $\mathit{fromIsRightInverse} : (A : \mathit{Type})\ (\mathit{xs} : \mathit{List}\ A) \to \mathit{to}\ (\mathit{from}\ \mathit{xs}) \equiv \mathit{xs}$
- $\mathit{fromIsLeftInverse} : (A : \mathit{Type})\ (\mathit{xs} : \mathit{Other}\ A) \to \mathit{from}\ (\mathit{to}\ \mathit{xs}) \equiv \mathit{xs}$

[2] List and Other cannot be isomorphic, as they are not types but type *constructors*.

We also want the operations on $Other$ to behave in the same way as the operations on $List$s, i.e. the following properties should be satisfied:

- $to\ otherEmpty \equiv [\,]$
- $(x : A) \to to\ (otherSingle\ x) \equiv [x]$
- $(xs\ ys : Other\ A) \to to\ (otherAppend\ xs\ ys) \equiv to\ xs + \!\!+\ to\ ys$
- $(f : A \to B)\ (xs : Other\ A) \to to\ (otherMap\ f\ xs) \equiv map\ f\ (to\ xs)$

These properties can be added to the original $Sequence$ type. However, it is rather tedious having to formulate these properties for every operation of the abstract type. Since we have specified the abstract type as a Σ-type, we can use propositional equality and univalence between these to guide us to the desired properties. The full specification predicate now becomes the following:

$$\begin{array}{l} SequenceSpecification : Sequence \to Type \\ SequenceSpecification\ seqImpl = seqImpl \equiv ListImpl \end{array}$$

We know that in order to prove that two values a and b of type $\Sigma\ (x : A)\ .B\ x$ are propositionally equal, we need to show its fields are propositionally equal as well:

$$\begin{array}{lll} \Sigma\text{-}\equiv & : & \{A : Type\}\ \{B : A \to Type\} \\ & & \{s\ s' : \Sigma\ (x : A)\ .\ B\ x\} \\ & & (p : fst\ s \equiv fst\ s') \\ & & (q : transport\ B\ p\ (snd\ s) \equiv snd\ s') \\ & \to & s \equiv s' \end{array}$$

If we want to prove that $ListImpl \equiv OtherImpl$, using $\Sigma\text{-}\equiv$, we first need to show that $List \equiv Other$. This can be done by showing that for every $(A : Type)$, we have an isomorphism $to : Other\ A \to List\ A$. Using the univalence axiom and function extensionality, we can then prove our goal, $List \equiv Other$. For the second part of the outermost Σ-type, we need to transport the snd of $ListImpl$ along the proof of $List \equiv Other$ we just gave and prove it to be propositionally equal to the snd of $OtherImpl$. Rather than deal with the fully general $Sequence$ where will show how the transporting looks like for the case when we fix the type parameter. This is done so we do not have to deal with function extensionality and only have to use univalence directly once. We consider the following definitions where we fix the type parameter $A : Type$:

$$\begin{array}{lll} Sequence_A = \Sigma\ (seq_A & : Set) & . \\ \ \Sigma\ (empty_A & : seq_A) & . \\ \ \Sigma\ (single_A & : A \to seq_A) & . \\ \ \Sigma\ (append_A & : seq_A \to seq_A \to seq_A) & . \\ \ \ (map_A & : (A \to A) \to seq_A \to seq_A) & \end{array}$$

with $ListImpl_A$ and $OtherImpl_A$ defined from the previous definitions.

To show that $\mathit{ListImpl}_A$ and $\mathit{OtherImpl}_A$, we need to show using univalence that $\mathit{List}\ A \equiv \mathit{Other}\ A$, so the beginning of the proof looks like this:

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathit{spec} :} \\
\multicolumn{2}{l}{\quad (\mathit{from} : \mathit{List}\ A \to \mathit{Other}\ A)} \\
\multicolumn{2}{l}{\quad (\mathit{to} : \mathit{Other}\ A \to \mathit{List}\ A)} \\
\multicolumn{2}{l}{\quad \to \mathit{Iso}\ (\mathit{List}\ A)\ (\mathit{Other}\ A)\ \mathit{from}\ \mathit{to}} \\
\multicolumn{2}{l}{\quad \to \mathit{ListImpl}_A \equiv \mathit{OtherImpl}_A} \\
\mathit{spec}\ \mathit{from}\ \mathit{to}\ \mathit{iso} = & \Sigma\text{-}\equiv\ (\mathit{univalence}\ (\mathit{List}\ A)\ (\mathit{Other}\ A)\ \mathit{iso}) \\
 & (\Sigma\text{-}\equiv\ \mathit{goal}_0 \\
 & (\Sigma\text{-}\equiv\ \mathit{goal}_1 \\
 & (\Sigma\text{-}\equiv\ \mathit{goal}_2 \\
 & \qquad\ \ \mathit{goal}_3)))
\end{array}$$

The first goal, goal_0, has type $\mathit{fst}\ (\mathit{transport}\ (\mathit{univalence}\ (\mathit{List}\ A)\ (\mathit{JoinList}\ A)\ \mathit{iso})$ $([\,], (\lambda x \to [x], (\_{+\!\!+}\_, \mathit{map})))) \equiv \mathit{otherEmpty}$. The left hand side of the equation is stuck, as we made use of the univalence axiom. However, we can prove that the first field of $\mathit{transport}$ applied to the dependent pair, is $\mathit{transport}$ applied to the first field of the dependent pair:

$$\begin{array}{l}
\Sigma\text{-}\mathit{transport} : \\
\quad \{\mathit{Ctx} : \mathit{Type}\} \\
\quad \{A : \mathit{Ctx} \to \mathit{Type}\}\ \{B : (\mathit{ctx} : \mathit{Ctx}) \to A\ \mathit{ctx} \to \mathit{Type}\} \\
\quad \{\mathit{ctx}\ \mathit{ctx}' : \mathit{Ctx}\} \\
\quad \{x : A\ \mathit{ctx}\}\ \{y : B\ \mathit{ctx}\ x\} \\
\quad (\mathit{pf} : \mathit{ctx} \equiv \mathit{ctx}') \to \\
\quad \mathit{fst}\ (\mathit{transport}\ (\lambda c \to \Sigma\ (x : A\ c)\ .\ B\ c\ x))\ \mathit{pf}\ (x, y)) \equiv \mathit{transport}\ (\lambda c \to A\ c)\ \mathit{pf}\ x
\end{array}$$

If we apply this to goal_0, we now need to show that $\mathit{transport}\ (\lambda c \to c)\ (\mathit{univalence}\ (\mathit{List}\ A)\ (\mathit{JoinList}\ A)\ \mathit{iso})\ [\,] \equiv \mathit{otherEmpty}$, which we can further reduce using the "computation" rule for univalence:

$$\begin{array}{l}
\mathit{univalence}\text{-}\mathit{comp} : \\
\qquad \{A\ B : \mathit{Type}\} \\
\qquad \{\mathit{from} : A \to B\} \\
\qquad \{\mathit{to} : B \to A\} \\
\qquad \{\mathit{iso} : \mathit{Iso}\ A\ B\ \mathit{from}\ \mathit{to}\} \\
\qquad \{x : A\} \\
\quad \to \mathit{transport}\ (\lambda X \to X)\ (\mathit{univalence}\ A\ B\ \mathit{iso})\ x \equiv \mathit{from}\ x
\end{array}$$

We have reduced goal_0 to the proof obligation $\mathit{from}\ \ [\,] \equiv \mathit{otherEmpty}$. We can apply the same steps to the other goals and recover the properties we formulated earlier. As we have now seen, using this method, giving a specification of an abstract type amounts to giving a nested Σ-type specifying the interface and a concrete view specifying the behaviour. We now get to prove properties of the abstract type without having to add numerous properties to the interface.

With the current "implementation" of homotopy type theory done by adding things such as univalence as axioms, we have to do all this rewriting by hand, but if we have a version of univalence available that computes, we automatically arrive at the desired properties.

3.2.2 Reasoning with views

If we want to prove a property about our abstract type, we now only have to prove that it holds for the concrete view. The resulting proof can then be used to show that it also holds for any other implementation of the abstract type.

As an example of this, we will show that the *empty* operation of our sequence type is the (left) unit of *append*. The case for lists is easy, assuming that $+\!\!+$ only does induction on its left argument:

```
left-unit-append : (xs : List A) → [ ] ++ xs ≡ xs
left-unit-append xs = refl
```

The general case of this statement is:

$$(\mathit{xs} : \mathit{Other}\ A) \to \mathit{otherAppend}\ \mathit{otherEmpty}\ \mathit{xs} \equiv \mathit{xs}$$

which can be established by the following equational reasoning:

$$\begin{array}{l}
\quad \mathit{xs} \\
\equiv \{\text{isomorphism}\} \\
\quad \mathit{from}\ (\mathit{to}\ \mathit{xs}) \\
\equiv \{[\,]\ \text{is left unit of}\ +\!\!+\} \\
\quad \mathit{from}\ ([\,] +\!\!+ \mathit{to}\ \mathit{xs}) \\
\equiv \{\text{specification of}\ \mathit{otherImpl}\} \\
\quad \mathit{from}\ (\mathit{to}\ \mathit{otherEmpty} +\!\!+ \mathit{to}\ \mathit{xs}) \\
\equiv \{\text{specification of}\ \mathit{otherAppend}\} \\
\quad \mathit{from}\ (\mathit{to}\ (\mathit{otherAppend}\ \mathit{otherEmpty}\ \mathit{xs})) \\
\equiv \{\text{isomorphism}\} \\
\quad \mathit{otherAppend}\ \mathit{otherEmpty}\ \mathit{xs}
\end{array}$$

3.2.3 Non-isomorphic views

An implementation of an abstract type sometimes does not turn out to be isomorphic to the concrete view. An example of this is an implementation of sequences via join lists:

```
data JoinList (A : Type) : Type where
  nil  : JoinList A
  unit : A → JoinList A
  join : JoinList A → JoinList A → JoinList A
```

Note that in this section we will fix an $A : \mathit{Type}$ and use subscripts to emphasise this and avoid the confusion between $\mathit{JoinList}\ (A/\!\sim)$ and $(\mathit{JoinList}\ A)/\!\sim$.

We have a function $\mathit{to} : \mathit{JoinList}_A \to \mathit{List}_A$ that maps *nil* to *nil*, $\mathit{unit}\ a$ to $[\,a\,]$ and interprets *join* as concatenation of lists. The other way around, $\mathit{from} : \mathit{List}_A \to \mathit{JoinList}_A$ can be constructed by mapping every element a of the input list to $\mathit{unit}\ a$ and then using *join* to concatenate the resulting list of *JoinList*s.

While we do have that $(\mathit{ls} : \mathit{List}_A) \to \mathit{to}\ (\mathit{from}\ \mathit{ls}) \equiv \mathit{ls}$, it is not the case that $(\mathit{js} : \mathit{JoinList}_A) \to \mathit{from}\ (\mathit{to}\ \mathit{js}) \equiv \mathit{js}$, as to is not injective: $\mathit{JoinList}$ has a finer structure than List. This means that to and from do not form an isomorphism. If only the first equality holds ($\mathit{to}\ (\mathit{from}\ \mathit{ls}) \equiv \mathit{ls}$), but the second does not, to is called a *retraction* with from as its *section*. It still makes sense to use $\mathit{JoinList}$ as an implementation of sequences. The properties that the operations on $\mathit{JoinList}$s should respect, do not make use of the fact that from and to are isomorphisms; they can still be used for non-isomorphic views.

Since we are only interested in using the $\mathit{JoinList}$ as a sequence and do not care how the inhabitants are balanced, we can take the quotient by the following relation:

$$\begin{aligned}&\_{\sim}\_ : \mathit{JoinList}_A \to \mathit{JoinList}_A \to \mathit{Type}\\ &x \sim y = \mathit{to}\ x \equiv \mathit{to}\ y\end{aligned}$$

The type $\mathit{Quotient}\ (\mathit{JoinList}_A)\ \_{\sim}\_$ is then isomorphic to List_A. This result can be generalised to arbitrary section-retraction pairs between h-sets A and B: given $r : A \to B$ and $s : B \to A$ such that $(a : A) \to s\ (r\ a) \equiv a$, then B is isomorphic to $A/{\sim}$ where $x \sim y$ is defined as $r\ x \equiv r\ y$. We have a function $A \to A/{\sim}$, namely the constructor box and can write a function $A/{\sim} \to A$. If we use $\mathit{Quotient\text{-}rec}$ for this, we need to supply a function $f : A \to A$ such that if $r\ x \equiv r\ y$, then also $f\ \ x \equiv f\ \ y$. Choosing f to be $\lambda x \to s\ (r\ x)$ works. The identity function need not work: if it did, r would be injective and would be an isomorphism. Let us name the functions between A and $A/{\sim}$ $\mathit{to\text{-}A/{\sim}}$ and $\mathit{from\text{-}A/{\sim}}$. Composing these functions with r or s, we get functions between $A/{\sim}$ and B that give us the desired isomorphism. Proving that this is an isomorphism mostly involves applying the proof that $r\ \ (s\ \ x) \equiv x$ in various ways. We also have to invoke the uniqueness of identity proofs property that $A/{\sim}$ admits (thanks to the 0-truncation) for the induction step on $A/{\sim}$. The fact that $\mathit{to\text{-}A/{\sim}}$ is a retraction with $\mathit{from\text{-}A/{\sim}}$ as its section can be proved using the same techniques.

To lift the operations on A to operations on $A/{\sim}$ we simply apply $\mathit{to\text{-}A/{\sim}}$ and $\mathit{from\text{-}A/{\sim}}$ in the right places. Showing that these lifted operations satisfy the conditions that follow from the specification then boils down to conditions that only refer to the operations on A in relation to those on B, as we will demonstrate with the $\mathit{JoinList}$ example. Let us define $\mathit{JoinList}_A/{\sim}$ as $\mathit{Quotient}\ A\ \_{\sim}\_$ with $x \sim y$ defined as $\mathit{to}\ x \equiv \mathit{to}\ y$. We have the following functions:

- $\mathit{to} : \mathit{JoinList}_A \to \mathit{List}_A$
- $\mathit{from} : \mathit{List}_A \to \mathit{JoinList}_A$
- $\overline{\mathit{to}} : \mathit{JoinList}_A \to \mathit{JoinList}_A/{\sim}$
- $\overline{\mathit{from}} : \mathit{JoinList}_A/{\sim} \to \mathit{JoinList}_A$

The isomorphism between $JoinList_A/\!\sim$ and $List_A$ is witnessed by $\mathit{to} \circ \overline{from} : JoinList_A/\sim \to List_A$ and $\overline{to} \circ \mathit{from} : List_A \to JoinList_A$. The $\overline{\mathit{empty}}$ of $JoinList_A/\sim$ is $\overline{to}\ \mathit{nil}$, which means that we need to establish $\mathit{to}\ (\overline{from}\ (\overline{to}\ \mathit{nil})) \equiv [\,]$. We can reduce this goal to $\mathit{to}\ \mathit{nil} \equiv [\,]$ via equational reasoning:

$$\begin{aligned}
&\quad \mathit{to}\ (\overline{from}\ (\overline{to}\ \mathit{nil}))\\
&\equiv \{\, \mathit{definition}\ \overline{to} \,\}\\
&\quad \mathit{to}\ (\overline{from}\ (\mathit{box}\ \mathit{nil}))\\
&\equiv \{\, \beta\ \mathit{reduction} \,\}\\
&\quad \mathit{to}\ (\mathit{from}\ (\mathit{to}\ \mathit{nil}))\\
&\equiv \{\, \mathit{to}\ /\ \mathit{from}\ \mathit{is}\ \mathit{a}\ \mathit{retraction}\ /\ \mathit{section} \,\}\\
&\quad \mathit{to}\ \mathit{nil}
\end{aligned}$$

In general we have that $\overline{from}\ (\overline{to}\ x) \equiv \mathit{from}\ (\mathit{to}\ x)$ holds for any $x : JoinList_A$. Deriving the property for single goes analogously to the derivation above. The rule for append is more interesting as we there also need $\overline{from}$ in other positions:

$$\begin{aligned}
&\quad \mathit{to}\ (\overline{from}\ (\overline{to}\ (\mathit{join}\ (\overline{from}\ \mathit{xs})\ (\overline{from}\ \mathit{ys}))))\\
&\equiv \{\, \beta\ \mathit{reduction} \,\}\\
&\quad \mathit{to}\ (\mathit{from}\ (\mathit{to}\ (\mathit{join}\ (\overline{from}\ \mathit{xs})\ (\overline{from}\ \mathit{ys}))))\\
&\equiv \{\, \mathit{to}\ /\ \mathit{from}\ \mathit{is}\ \mathit{a}\ \mathit{retraction}\ /\ \mathit{section} \,\}\\
&\quad \mathit{to}\ (\mathit{join}\ (\overline{from}\ \mathit{xs})\ (\overline{from}\ \mathit{ys}))
\end{aligned}$$

We end up with having to prove the following:

$$\begin{aligned}
&(\mathit{xs}\ \mathit{ys} : JoinList_A/\!\sim) \to\\
&\quad \mathit{to}\ (\mathit{join}\ (\overline{from}\ \mathit{xs})\ (\overline{from}\ \mathit{ys})) \equiv \mathit{to}\ (\overline{from}\ \mathit{xs}) +\!\!+\ \mathit{to}\ (\overline{from}\ \mathit{ys})
\end{aligned}$$

which follows from $(\mathit{xs}\ \mathit{ys} : JoinList_A) \to \mathit{to}\ (\mathit{join}\ \mathit{xs}\ \mathit{ys}) \equiv \mathit{to}\ \mathit{xs} +\!\!+\ \mathit{to}\ \mathit{ys}$.

The above derivation shows us that we might arrive at equations that are a bit less general than the equations we get from if we were to pretend our retraction-section pair is actually an isomorphism.

Non-isomorphic views via definable quotients

It so happens that the quotient $A/\sim$ is definable: it can be defined as the type $\Sigma\ (x : A)\ .\ s\ (r\ x) \equiv x$, i.e. restrict A to those inhabitants such that (the lifted versions of) s and r become isomorphisms. The function box is then defined by:

$$\begin{aligned}
&\mathit{box} : A \to \Sigma\ (x : A)\ .\ s\ (r\ x) \equiv x\\
&\mathit{box}\ x = (s\ (r\ x)), \mathit{ap}\ s\ (\mathit{is\text{-}retract}\ (r\ x))
\end{aligned}$$

where $\mathit{is\text{-}retract} : (x : B) \to r\ (s\ x) \equiv x$ witnesses the fact that r and s form a retraction-section pair.

Notice that for the quotient type we have the $\lambda x \to s\ (r\ x)$ in the "deconstructor" (i.e. in the function $\overline{from} : JoinList_A/{\sim} \to JoinList\ A$) and here we have it in the constructor (i.e. the function box). This stems from the fact that the soundness of quotient types is enforced by the way they are eliminated. It is only there that we have the obligation to show that we respect the relation on the type. With the Σ-type it is more correctness by construction.

From a computational perspective, the first approach with the quotient types is more desirable, as the values of the type do not carry around any correctness proof.

3.3 Conclusion

Higher inductive types allow us to straightforwardly define quotient types. This definition works better than the setoid method in that we no longer have to be careful whether we use the custom equivalence relation or propositional equality: we only have to consider propositional equality. However, as is common with higher inductive types, we have to take the 0-truncation in the definition of a quotient type. This makes the elimination principle more complex to work with, but since virtually any of the types we encounter in programming are h-sets, the extra conditions that the 0-truncation adds to the elimination principle are usually trivially satisfied.

Univalence gives us a very clean way to define specifications of abstract types using concrete views. Working with this specification, e.g. trying to prove that a given implementation satisfies the specification, involves a lot of manual fiddling with the computation rules of Σ-types and univalence. Having a computational interpretation of univalence would obviously be of great importance for this method to be useful.

Using quotient types, we can also define a view on an abstract type that is not isomorphic to the concrete type of the reference implementation, but only instead we have a retraction-section pair between the two types. Any retraction-section pair can be turned into an isomorphism, by quotienting out by the retraction. Such a quotient happens to be definable, which means that we do not need the quotient type construction using higher inductive types to do this. However, the higher inductive type construction does yield a definition that is more amenable to the optimisations that will be discussed in chapter 4, as the proofs that the quotient structure is respected only occur in the calls to the elimination principle, instead of occurring in all the terms of type, which is the case with the definable quotient implementation.

Chapter 4

Erasing propositions

When writing certified programs in a dependently typed setting, we can conceptually distinguish between the program parts and the proof (of correctness) parts. These are sometimes also referred to as the informative[1] and logical parts, respectively. In practice, these two seemingly separate concerns are often intertwined. Consider for example the sorting of lists of naturals: given some predicate $\mathit{isSorted} : \mathit{List}\ \mathbb{N} \rightarrow \mathit{List}\ \mathbb{N} \rightarrow \mathit{Type}$ that tells us whether the second list is a sorted permutation of the first one, we can to write a term of the following type:

$$\mathit{sort} : (\mathit{xs} : \mathit{List}\ \mathbb{N}) \rightarrow \Sigma\ (\mathit{ys} : \mathit{List}\ \mathbb{N})\ .\ (\mathit{isSorted}\ \mathit{xs}\ \mathit{ys})$$

To implement such a function, we need to provide for every list a sorted list along with a proof that this is indeed a sorted version of the input list. At run-time the type checking has been done, hence the proof of correctness has already been verified: we want to *erase* these logical parts.

Types such as $\mathit{isSorted}\ \mathit{xs}\ \mathit{ys}$ are purely logical: we care more about the presence of an inhabitant than what kind of inhabitant we exactly have at our disposal. In section 4.1 we will give more examples of such types, called *propositions* (compare this with the definition of h-propositions via proof irrelevance (section 2.4), and how they can occur in various places in certified programs. In section 4.2 and section 4.3 we review the methods Coq and Agda provide us to annotate parts of our program as being propositions in such a way that those parts can be erased after type checking and are absent at run-time. Section 4.4 reviews the concept of *collapsible families* and how we can automatically detect whether a type is a proposition, instead of annotating them ourselves. In section 4.5 we internalise the concept of collapsible families and try to do the same with the optimisation in section 4.6. The internalised version of collapsibility looks like an indexed version of the concept of h-propositions. In section 4.7 we investigate if we can use this to devise an optimisation akin to the optimisation based on collapsibility.

[1] Instead of "informative", it is sometimes also called "computation", but this is a bit of a misnomer as the proof parts can be computational as well, but then only at compile time (i.e. during type checking).

4.1 Propositions

In the $sort$ example, the logical part $isSorted\ xs\ ys$ occurs in the result as part of a Σ-type. This means we can separate the proof of correctness from the sorting itself, i.e. we can write a function $sort' : List\ \mathbb{N} \to List\ \mathbb{N}$ and a proof of the following:

$$sortCorrect : (xs : List\ \mathbb{N}) \to isSorted\ xs\ (sort'\ xs)$$

The logical part here asserts properties of the *result* of the computation. If we instead have assertions on our *input*, we cannot decouple this from the rest of the function as easily as, if it is at all possible. For example, suppose we have a function, safely selecting the n-th element of a list:

$$elem : (A : Type)\ (xs : List\ A)\ (i : \mathbb{N}) \to i < length\ xs \to A$$

If we were to write $elem$ without the bounds check $i < length\ xs$, we would get a partial function. Since we can only define total functions in our type theory, we cannot write such a function. However, at run-time, carrying these proofs around makes no sense: type checking has already shown that all calls to $elem$ are safe and the proofs do not influence the outcome of $elem$. We want to erase terms of types such as $i < length\ xs$, if we have established that they do not influence the run-time computational behaviour of our functions.

4.1.1 Bove-Capretta method

The $elem$ example showed us how we can use propositions to write functions that would otherwise be partial, by asserting properties of the input. The Bove-Capretta method [Bove and Capretta, 2005] generalises this and more: it provides us with a way to transform any (possibly partial) function defined by general recursion into a total, structurally recursive one. The quintessential example of a definition that is not structurally recursive is *quicksort*[2] :

$$\begin{aligned}
&qs : List\ \mathbb{N} \to List\ \mathbb{N}\\
&qs\ []\qquad\quad = []\\
&qs\ (x :: xs) = qs\ (filter\ (gt\ x)\ xs) +\!\!+\ x :: qs\ (filter\ (le\ x)\ xs)
\end{aligned}$$

The recursive calls are done on $filter\ (gt\ x)\ xs$ and $filter\ (le\ x)\ xs$ instead of just xs, hence qs is not structurally recursive. To solve this problem, we create an inductive family describing the call graphs of the original function for every input. Since we can only construct finite values, being able to produce such a call graph essentially means that the function terminates for that input. We can then write a new function that structurally recurses on the call graph.

[2] In most implementations of functional languages, this definition will not have the same space complexity as the usual in-place version. We are more interested in this function as an example of non-structural recursion and are not too concerned with its complexity.

In our quicksort case we get the following inductive family:

$$\begin{array}{l}
\textbf{data}\ \mathit{qsAcc} : \mathit{List}\ \mathbb{N} \to \mathit{Type}\ \textbf{where} \\
\quad \mathit{qsAccNil} \quad : \mathit{qsAcc}\ [\,] \\
\quad \mathit{qsAccCons} : (x : \mathbb{N})\ (\mathit{xs} : \mathit{List}\ \mathbb{N}) \\
\qquad\qquad\qquad (h_1 : \mathit{qsAcc}\ (\mathit{filter}\ (\mathit{gt}\ x)\ \mathit{xs})) \\
\qquad\qquad\qquad (h_2 : \mathit{qsAcc}\ (\mathit{filter}\ (\mathit{le}\ x)\ \mathit{xs})) \\
\qquad\qquad\qquad \to \mathit{qsAcc}\ (x :: \mathit{xs})
\end{array}$$

with the following function definition[3]

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathit{qs} : (\mathit{xs} : \mathit{List}\ \mathbb{N}) \to \mathit{qsAcc}\ \mathit{xs} \to \mathit{List}\ \mathbb{N}} \\
\mathit{qs}\ .\mathit{nil} \quad \mathit{qsAccNil} & = [\,] \\
\mathit{qs}\ .\mathit{cons}\ (\mathit{qsAccCons}\ x\ \mathit{xs}\ h_1\ h_2) & = \mathit{qs}\ (\mathit{filter}\ (\mathit{gt}\ x)\ \mathit{xs})\ h_1 +\!\!+ \\
& \quad\ x :: \mathit{qs}\ (\mathit{filter}\ (\mathit{le}\ x)\ \mathit{xs})\ h_2
\end{array}$$

Pattern matching on the $\mathit{qsAcc}\ \mathit{xs}$ argument gives us a structurally recursive version of qs. Just as with the elem example, we need information from the proof to be able to write this definition in our type theory. In the case of elem, we need the proof of $i < \mathit{length}\ \mathit{xs}$ to deal with the (impossible) case where xs is empty. In the qs case, we need $\mathit{qsAcc}\ \mathit{xs}$ to guide the recursion. Even though we actually pattern match on $\mathit{qsAcc}\ \mathit{xs}$ and it therefore seemingly influences the computational behaviour of the function, erasing this argument yields the original qs definition.

4.2 The *Prop* universe in Coq

In Coq we have have the *Prop* universe, apart from the *Set* universe. Both universes act as base sorts of the hierarchy of sorts, *Type*, i.e. $\mathit{Prop} : \mathit{Type}\ (1)$, $\mathit{Set} : \mathit{Type}\ (1)$ and for every i, $\mathit{Type}\ (i) : \mathit{Type}\ (i+1)$. As the name suggests, by defining a type to be of sort *Prop*, we "annotate" it to be a logical type, a proposition. Explicitly marking the logical parts like this, makes the development easier to read and understand: we can more easily distinguish between the proof of correctness parts and the actual program parts. More importantly, Coq's extraction mechanism [Letouzey, 2003] now knows what parts are supposed to be logical, hence what parts are to be erased.

In the *sort* example, we would define *isSorted* to be a family of *Prop*s indexed by $\mathit{List}\ \mathbb{N}$. For the Σ-type, Coq provides two options: *sig* and *ex*, defined as follows:

$$\begin{array}{l}
\textbf{Inductive}\ \mathit{sig}\ (A : \mathit{Type})\ (P : A \to \mathit{Prop}) : \mathit{Type} := \\
\quad \mathit{exist} : \forall\ x : A, P\ x \to \mathit{sig}\ P \\
\textbf{Inductive}\ \mathit{ex}\ (A : \mathit{Type})\ (P : A \to \mathit{Prop}) : \mathit{Prop} := \\
\quad \mathit{ex\_intro} : \forall\ x : A, P\ x \to \mathit{ex}\ P
\end{array}$$

[3] This definition uses dependent pattern matching [Coquand, 1992], but can be rewritten directly using the elimination operators instead. The important thing here is to notice that we are eliminating the $\mathit{qsAcc}\ \mathit{xs}$ argument.

As can be seen above, sig differs from ex in that the latter is completely logical, whereas sig has one informative and one logical field and in its entirety is informative. Since we are interested in the $list$ $\mathbb{N}$ part of the Σ-type that is the result type of $sort$, but not the $isSorted$ part, we choose the sig version.

The extracted version of sig consists of a single constructor $exist$, with a single field of type A. Since this is isomorphic the type A itself, Coq optimises this away during extraction. This means $sort : (xs : List\ \mathbb{N}) \to \Sigma\ (ys : List\ \mathbb{N})\ .\ (isSorted\ xs\ ys)$ gets extracted to a function $sort' : List\ \mathbb{N} \to List\ \mathbb{N}$.

When erasing all the $Prop$ parts from our program, we do want to retain the computational behaviour of the remaining parts. Every function that takes an argument of sort $Prop$, but whose result type is not in $Prop$, needs to be invariant under choice of inhabitant for the $Prop$ argument. To force this property, Coq restricts the things we can eliminate a $Prop$ into. The general rule is that pattern matching on something of sort $Prop$ is allowed if the result type of the function happens to be in $Prop$.

4.2.1 Singleton elimination and homotopy type theory

There are exceptions to this rule: if the argument we are pattern matching on happens to be an *empty* or *singleton definition* of sort $Prop$, we may also eliminate into $Type$. An empty definition is an inductive definition without any constructors. A singleton definition is an inductive definition with precisely one constructor, whose fields are all in $Prop$. Examples of such singleton definitions are conjunction on $Prop$ (/\) and the accessibility predicate Acc used to define functions using well-founded recursion.

Another important example of singleton elimination is elimination on Coq's equality eq (where $a = b$ is special notation for $eq\ a\ b$), which is defined to be in $Prop$. The inductive family eq is defined in the same way as we have defined identity types, hence it is a singleton definition, amenable to singleton elimination. Consider for example the $transport$ function:

$$\begin{aligned}
&\textbf{Definition}\ transport : \forall\ A, \forall\ (P : A \to Type),\\
&\quad \forall\ (x\ y : A),\\
&\quad \forall\ (path : x = y),\\
&\quad P\ x \to P\ y\ .
\end{aligned}$$

Singleton elimination allows us to pattern match on $path$ and and eliminate into something of sort $Type$. In the extracted version, the $path$ argument gets erased and the $P\ x$ argument is returned. In homotopy type theory, we know that the identity types need not be singletons and can have other inhabitants than just the canonical $refl$, so throwing away the identity proof is not correct. As has been discovered by Schulman [2012], singleton elimination leads to some sort of inconsistency, if we assume the univalence axiom: we can construct a value $x : bool$ such that we can prove $x = false$, even though in the extracted version x normalises to $true$. Assuming univalence, we have two distinct proofs of $bool = bool$, namely $refl$ and the proof we get from applying univalence to the isomorphism $not : bool \to bool$. Transporting a value along a path we have obtained from using univalence, is the same as applying the isomorphism. Defining x to be $true$ transported along the path obtained from applying univalence to the isomorphism not, yields something that is propositionally equal to $false$. If we extract the development, we get a definition of x that ignores the proof of $bool = bool$ and just returns $true$.

In other words, Coq does not enforce or check proof irrelevance of the types we define to be of sort $Prop$, which internally is fine: it does not allow us to derive falsity using this fact. The extraction mechanism however, does assume that everything admits proof irrelevance. The combination of this along with singleton elimination means that we can prove properties about our programs that no longer hold in the extracted version. It also goes to show that the design decision to define the identity types to be in $Prop$ is not compatible with homotopy type theory.

4.2.2 Quicksort example

In the case of qs defined using the Bove-Capretta method, we actually want to pattern match on the logical part: $qsAcc\ xs$. Coq does not allow this if we define the family $qsAcc$ to be in $Prop$. However, we can do the pattern matching "manually", as described in Bertot and Castéran [2004]. We know that we have exactly one inhabitant of $qsAcc\ xs$ for each xs, as they represent the call graph of qs for the input xs, and the pattern matches of the original definition do not overlap, hence each xs has a unique call graph. We can therefore easily define and prove the following inversion theorems, that roughly look as follows:

$$qsAccInv_0 : (x : \mathbb{N})\ (xs : List\ \mathbb{N})\ (qsAcc\ (x :: xs)) \rightarrow qsAcc\ (filter\ (le\ x)\ xs)$$
$$qsAccInv_1 : (x : \mathbb{N})\ (xs : List\ \mathbb{N})\ (qsAcc\ (x :: xs)) \rightarrow qsAcc\ (filter\ (gt\ x)\ xs)$$

We define the function qs just as we originally intended to and add the $qsAcc\ xs$ argument to every pattern match. We then call the inversion theorems for the appropriate recursive calls. Coq still notices that there is a decreasing argument, namely $qsAcc\ xs$. If we follow this approach, we can define $qsAcc$ to be a family in $Prop$ and recover the original qs definition without the $qsAcc\ xs$ argument using extraction.

In the case of partial functions, we still have to add the missing pattern matches and define impossibility theorems: if we reach that pattern match and we have a proof of our Bove-Capretta predicate for that particular pattern match, we can prove falsity, hence we can use $False\_rect$ do deal with the missing pattern match.

4.2.3 Impredicativity

So far we have seen how *Prop* differs from *Set* with respect to its restricted elimination rules and its erasure during extraction, but *Prop* has another property that sets it apart from *Set*: *impredicativity*. Impredicativity means that we are able to quantify over something which contains the thing currently being defined. In set theory unrestricted use of this principle leads us to being able to construct Russell's paradox: the set $R = \{x | x \in x\}$ is an impredicative definition, we quantify over x, while we are also defining x. Using this definition we can prove that $R \in R$ if and only if $R \notin R$. Impredicativity is also a necessary ingredient for the Burali-Forti paradox: constructing the set of all ordinal numbers yields an inconsistency. It is this paradox that can be expressed in impredicative Martin-Löf's type theory (i.e. $Type : Type$ holds), where it is called Girard's paradox. However, impredicative definitions are sometimes very useful and benign, in particularly when dealing with propositions: we want to be able to write propositions that quantify over propositions, for example:

```
Definition demorgan : Prop := ∀ P Q : Prop,
  ~ (P /\ Q) → ~ P \/ ~ Q .
```

Coq allows for such definitions as the restrictions on *Prop* prevent us from constructing paradoxes such as Girard's. For details on these limitations, the reader is referred to the Coq FAQ[4].

4.3 Irrelevance in Agda

In Coq, we put the annotations of something being a proposition in the definition of our inductive type, by defining it to be of sort *Prop*. With Agda's irrelevance mechanism, we instead put the annotations at the places we *use* the proposition, by placing a dot in front of the corresponding type. For example, the type of the *elem* becomes:

```
elem : (A : Type) (xs : List A) (i : ℕ) → .(i < length xs) → A
```

We can also mark fields of a record to be irrelevant. In the case of *sort*, we want something similar to the *sig* type from Coq, where second field of the Σ-type is deemed irrelevant. In Agda this can be done as follows:

```
record Σ-irr (A : Type) (B : A → Type) : Type where
  constructor _,_
  field
    fst  : A
    .snd : B fst
```

[4] http://coq.inria.fr/V8.1/faq.html#htoc49

To ensure that irrelevant arguments are indeed irrelevant to the computation at hand, Agda has several criteria that it checks. First of all, no pattern matching may be performed on irrelevant arguments, just as is the case with Prop. (However, the absurd pattern may be used, if applicable.) Contrary to Coq, singleton elimination is not allowed. Secondly, we need to ascertain that the annotations are preserved: irrelevant arguments may only be passed on to irrelevant contexts. This prevents us from writing a function of type $(A : \mathit{Type}) \to\ .A \to A$.

Another, more important, difference with Prop is that irrelevant arguments are ignored by the type checker when checking equality of terms. This can be done safely, even though the terms at hand may in fact be definitionally different, as we never need to appeal to the structure of the value: we cannot pattern match on it. The only thing that we can do with irrelevant arguments is either ignore them or pass them around to other irrelevant contexts.

The reason why the type checker ignoring irrelevant arguments is important, is that it allows us to′ prove properties about irrelevant arguments in Agda, internally. For example: any function out of an irrelevant type is constant:

$$\begin{aligned}
&\mathit{irrelevantConstantFunction} &&: \{A : \mathit{Type}\}\ \{B : \mathit{Type}\} \\
& &&\to (f :\ .A \to B) \to (x\ y : A) \to f\ x \equiv f\ y \\
&\mathit{irrelevantConstantFunction}\ f\ x\ y = \mathit{refl}
\end{aligned}$$

There is no need to use the congruence rule for $\equiv$, since the x and y are ignored when the type checker compares $f\ x$ to $f\ y$, when type checking the refl. The result can be easily generalised to dependent functions:

$$\begin{aligned}
&\mathit{irrelevantConstantDepFunction} &&: \{A : \mathit{Type}\}\ \{B :\ .A \to \mathit{Type}\} \\
& &&\to (f :\ .(x : A) \to B\ x) \to (x\ y : A) \to f\ x \equiv f\ y \\
&\mathit{irrelevantConstantDepFunction}\ f\ x\ y = \mathit{refl}
\end{aligned}$$

Note that we do not only annotate $(x : A)$ with a dot, but also occurrence of A in the type $B : A \to \mathit{Type}$, otherwise we are not allowed to write $B\ x$ as we would use an irrelevant argument in a relevant context. When checking the term $\mathit{irrelevantConstantDepFunction}$, the term $f\ x \equiv f\ y$ type checks, without having to transport one value along some path, because the types $B\ x$ and $B\ y$ are regarded as definitionally equal by the type checker, ignoring the x and y. Just as before, there is no need to use the (dependent) congruence rule; a refl suffices.

We would also like to show that we have proof irrelevance for irrelevant arguments, i.e. we want to prove the following:

$$\mathit{irrelevantProofIrrelevance} : \{A : \mathit{Type}\}\ .(x\ y : A) \to x \equiv y$$

Agda does not accept this, because the term $x \equiv y$ uses irrelevant arguments in a relevant context: $x \equiv y$. If we instead package the irrelevant arguments in an inductive type, we can prove that the two values of the packaged type are propositionally equal.

Consider the following record type with only one irrelevant field:

```
record Squash (A : Type) : Type where
  constructor squash
  field
    .proof : A
```

Using this type, we can now formulate the proof irrelevance principle for irrelevant arguments and prove it:

```
squashProofIrrelevance : {A : Type} (x y : Squash A) → x ≡ y
squashProofIrrelevance x y = refl
```

The name "squash type" comes from Nuprl [Constable et al., 1986]: one takes a type and identifies (or "squashes") all its inhabitants into one unique (up to propositional equality) inhabitant. In homotopy type theory the process of squashing a type is called (-1)-truncation (section 2.4.1) and can also be achieved by defining the following higher inductive type:

```
data (−1)-truncation : (A : Type) : Type where
  inhabitant : A → (−1)-truncation A
  all-paths : (x y : (−1)-truncation A) → x ≡ y
```

4.3.1 Quicksort example

If we want to mark the *qsAcc xs* argument of the *qs* function as irrelevant, we run into the same problems as we did when we tried to define *qsAcc* as a family in *Prop*: we can no longer pattern match on it. In Coq, we did have a way around this, by using inversion and impossibility theorems to do the pattern matching "manually". However, if we try such an approach in Agda, its termination checker cannot see that *qsAcc xs* is indeed a decreasing argument and refuses the definition.

4.4 Collapsible families

The approaches we have seen so far let the user indicate what parts of the program are the logical parts and are amenable for erasure. Brady et al. [2004] show that we can let the compiler figure that out by itself instead. The authors propose a series of optimisations for the Epigram system, based on the observation that one often has a lot of redundancy in well-typed terms. If it is the case that one part of a term has to be definitionally equal to another part in order to be well-typed, we can leave out (presuppose) the latter part if we have already established that the term is well-typed.

The authors describe their optimisations in the context of Epigram. In this system, the user writes programs in a high-level language that gets elaborated to programs in a small type theory language. This has the advantage that if we can describe a translation for high-level features, such as dependent pattern matching, to a simple core type theory, the metatheory becomes a lot simpler. The smaller type theory also allows us to specify optimisations more easily, because we do not have to deal with the more intricate, high-level features.

As such, the only things we need to look at, if our goal is to optimise a certain inductive family, are its constructors and its elimination principle. Going back to the $elem$ example, we had the $i < length\ xs$ argument. The smaller-than relation can be defined as the following inductive family (in Agda syntax):

$$\begin{array}{l}
\textbf{data}\ \_ < \_ : \mathbb{N} \to \mathbb{N} \to Type\ \textbf{where} \\
\quad ltZ : (y : \mathbb{N}) \qquad\qquad\quad \to Z \quad < S\ y \\
\quad ltS : (x\ y : \mathbb{N}) \to x < y \to S\ x < S\ y
\end{array}$$

with elimination operator

$$\begin{array}{rl}
< \text{-}elim : & (P : (x\ y : \mathbb{N}) \to x < y \to Type) \\
& (m_Z : (y : \mathbb{N}) \to P\ 0\ (S\ y)\ (ltZ\ y)) \\
& (m_S : (x\ y : \mathbb{N}) \to (pf : x < y) \to P\ x\ y\ pf \to P\ (S\ x)\ (S\ y)\ (ltS\ x\ y\ pf)) \\
& (x\ y : \mathbb{N}) \\
& (pf : x < y) \\
& \to P\ x\ y\ pf
\end{array}$$

and computation rules

$$\begin{array}{lll}
< \text{-}elim\ P\ m_Z\ m_S\ 0 & (S\ y)\ (ltZ\ y) & \triangleq m_Z\ y \\
< \text{-}elim\ P\ m_Z\ m_S\ (S\ x) & (S\ y)\ (ltS\ x\ y\ pf) & \triangleq m_S\ x\ y\ pf\ (< \text{-}elim\ P\ m_Z\ m_S\ x\ y\ pf)
\end{array}$$

If we look at the computation rules, we see that we can presuppose several things. The first rule has a repeated occurrence of y, so we can presuppose the latter occurrence, the argument of the constructor. In the second rule, the same can be done for x and y. The pf argument can also be erased, as it is never inspected: the only way to inspect pf is via another call the $<$ -$elim$, so by induction it is never inspected. Another thing we observe is that the pattern matches on the indices are disjoint, so we can presuppose the entire target: everything can be recovered from the indices given to the call of $<$ -$elim$.

We have to be careful when making assumptions about values, given their indices. Suppose we have written a function that takes $p : 1 < 1$ as an argument and contains a call to $<$ -$elim$ on p. If we look at the pattern matches on the indices, we may be led to believe that p is of form $ltS\ 0\ 0\ p'$ for some $p' : 0 < 0$ and reduce accordingly. The presupposing only works for *canonical* values, hence we restrict our optimisations to the run-time (evaluation in the empty context), as we know we do not perform reductions under binders in that case and every value is canonical after reduction. The property that every term that is well-typed in the empty context, reduces to a canonical form is called *adequacy* and is a property that is satisfied by Martin-Löf's type theory.

The family $<$ -$elim$ has the property that for indices $x\ y : \mathbb{N}$, its inhabitants $p : x < y$ are uniquely determined by these indices. To be more precise, the following is satisfied: for all $x\ y : \mathbb{N}$, $\vdash\ p\ q : x < y$ implies $\vdash\ p \triangleq q$. Families $D : I_0 \to \cdots \to I_n \to Type$ such as $<$ -$elim$ are called *collapsible* if they satisfy that for every $i_0 : I_0, \cdots, i_n : I_n$, if $\vdash\ p\ q : D\ i_0\ \cdots\ i_n$, then $\vdash\ p \triangleq q$.

Checking collapsibility of an inductive family is undecidable in general. This can be seen by reducing it to the type inhabitation problem: consider the type $\top + A$. This type is collapsible if and only if A is uninhabited, hence determining with being able to decide collapsibility means we can decide type inhabitation as well. As such, we limit ourselves to a subset that we can recognise, called *concretely* collapsible families. A family $D : I_0 \to \cdots \to I_n \to Type$ is concretely collapsible if satisfies the following two properties:

- If we have $\vdash\ x : D\ i_0\ \cdots\ i_n$, for some $i_0 : I_0, \cdots, i_n : I_n$, then we can recover its constructor tag by pattern matching on the indices.
- All the non-recursive arguments to the constructors of D can be recovered by pattern matching on the indices.

Note that the first property makes sense because we only have to deal with canonical terms, due to the adequacy property. Checking whether this first property holds can be done by checking whether the indices of the constructors, viewed as patterns, are disjoint. The second property can be checked by pattern matching on the indices of every constructor and checking whether the non-recursive arguments occur as pattern variables.

4.4.1 Erasing concretely collapsible families

If D is a collapsible family, then its elimination operator D-$elim$ is constant in its target, if we fix the indices. This seems to indicate that there might be a possibility to erase the target altogether. Nevertheless, D might have constructors with non-recursive arguments giving us information. Concretely collapsible families satisfy the property that this kind of information can be recovered from the indices, so we can get away with erasing the entire target. Being concretely collapsible means that we have a function at the meta-level (or implementation level) from the indices to the non-recursive, relevant parts of the target. Since this is done by pattern matching on the fully evaluated indices, recovering these parts takes an amount of time that is constant in the size of the given indices. Even though this sounds promising, the complexity of patterns does influence this constant, e.g. the more deeply nested the patterns are, the higher the constant. We now also need the indices to be fully evaluated when eliminating a particular inductive family, whereas that previously might not have been needed. The optimisation is therefore one that gives our dependently typed programs a better space complexity, but not necessarily a better time complexity.

4.4.2 Quicksort example

The accessibility predicates $qsAcc$ form a collapsible family. The pattern matches on the indices in the computation rules for $qsAcc$ are the same pattern matches as those of the original qs definition. There are no overlapping patterns in the original definition, so we can indeed recover the constructor tags from the indices. Also, the non-recursive arguments of $qsAcc$ are precisely those given as indices, hence $qsAcc$ is indeed a (concretely) collapsible family. By the same reasoning, any Bove-Capretta predicate is concretely collapsible, given that the original definition we derived the predicate from, has disjoint pattern matches.

The most important aspect of the collapsibility optimisation is that we have established that everything we need from the value that is to be erased, can be (cheaply) *recovered* from its indices passed to the call to its elimination operator. This means that we have no restrictions on the elimination of collapsible families: we can just write our definition of qs by pattern matching on the $qsAcc\ xs$ argument. At run-time, the $qsAcc\ xs$ argument has been erased and the relevant parts are recovered from the indices.

4.5 Internalising collapsibility

Checking whether an inductive family is concretely collapsible is something that can be easily done automatically, as opposed to determining collapsibility in general, which is undecidable. Since collapsibility is also a meta-theoretical concept (it makes use of definitional equality and talks about provability), it is only the compiler that can find out whether an inductive family is collapsible or not. If we want to provide the user with the means to give a proof of collapsibility for a certain family itself, if the compiler fails to notice this, then we would need to specify a new language for such evidence. Instead of create such a language, we will create an internal version of the meta-theoretical notion of collapsibility, so that user can provide the evidence in the type theory itself.

Recall the definition of a collapsible family[5] : given an inductive family D indexed by the type I, we say that D is collapsible if for every index $i : I$ and terms x, y, the following holds:

$$\vdash x, y : D\ i \text{ implies} \vdash x \triangleq y$$

This definition makes use of definitional equality. Since we are working with an intensional type theory, we do not have the *equality reflection rule* at our disposal: there is no rule that tells us that propositional equality implies definitional equality. This might lead us to think that internalising the above definition will not work, as we seemingly cannot say anything about definitional equality from within Martin-Löf's type theory.

[5] The definition we originally gave allowed for an arbitrary number of indices. In the following sections we will limit ourselves to the case where we have only one index for presentation purposes. All the results given can be easily generalised to allow more indices.

Let us consider the following variation: for all terms x, y there exists a term p such that

$$\vdash x, y : D\ i \text{ implies} \vdash p : x \equiv y$$

Since Martin-Löf's type theory satisfies the canonicity property, any term p such that $\vdash\ p : x \equiv y$ reduces to *refl*. The only way for the term to type check, is if $x \overset{\Delta}{=} y$, hence in the empty context the equality reflection rule does hold. The converse is also true: definitional equality implies of x and y that $\vdash$ *refl* $: x \equiv y$ type checks, hence the latter definition is equal to the original definition of collapsibility.

The variation given above is still not a statement that we can directly prove internally: we need to internalise the implication and replace it by the function space. Doing so yields the following following definition: there exists a term p such that:

$$\vdash p : (i : I) \to (x\ y : D\ i) \to x \equiv y$$

Or, written as a function in Agda:

$$\begin{aligned}&\mathit{isInternallyCollapsible} : (I : \mathit{Type})\ (A : I \to \mathit{Type}) \to \mathit{Type}\\ &\mathit{isInternallyCollapsible}\ I\ A = (i : I) \to (x\ y : A\ i) \to x \equiv y\end{aligned}$$

We will refer to this definition as *internal collapsibility*. It is easy to see that every internally collapsible family is also collapsible, by canonicity and the fact that *refl* implies definitional equality. However, internally collapsible families do differ from collapsible families as can be seen by considering D to be the family Id. By canonicity we have that for any $A : \mathit{Type}$, $x, y : A$, a term p satisfying $\vdash p : \mathit{Id}\ A\ x\ y$ necessarily reduces to *refl*. This means that *Id* is a collapsible family. In contrast, *Id* does not satisfy the internalised condition given above, since this then boils down to the uniqueness of identity proofs principle, which does not hold, as we have discussed.

4.6 Internalising the collapsibility optimisation

In section 4.4.1 we saw how concretely collapsible families can be erased, since all we want to know about the inhabitants can be recovered from its indices. In this section we will try to uncover a similar optimisation for internally collapsible families.

We cannot simply erase the internally collapsible arguments from the function we want to optimise, e.g. given a function $f : (i : I) \to (x : D\ i) \to \tau$, we generally cannot produce a function $\widetilde{f} : (i : I) \to \tau$, since we sometimes need the $x : D\ i$ in order for the function to type check. However, we can use Agda's irrelevance mechanism to instead generate a function in which the collapsible argument is marked as irrelevant.

The goal is now to write the following function (for the non-dependent case):

$$\begin{array}{l}
\mathit{optimiseFunction} : \\
\quad (I : \mathit{Type})\ (A : I \to \mathit{Type})\ (B : \mathit{Type}) \\
\quad (\mathit{isInternallyCollapsible}\ I\ A) \\
\quad (f : (i : I) \to A\ i \to B) \\
\quad\ \to ((i : I) \to\ .(A\ i) \to B)
\end{array}$$

Along with such a function, we should also give a proof that the generated function is equal to the original one in the following sense:

$$\begin{array}{l}
\mathit{optimiseFunctionCorrect} : \\
\quad (I : \mathit{Type})\ (D : I \to \mathit{Type})\ (B : \mathit{Type}) \\
\quad (\mathit{pf} : \mathit{isInternallyCollapsible}\ I\ D) \\
\quad (f : (i : I) \to D\ i \to B) \\
\quad (i : I)\ (x : D\ i) \\
\quad\ \to \mathit{optimiseFunction}\ I\ D\ B\ \mathit{pf}\ f\ i\ x \equiv f\ i\ x
\end{array}$$

If we set out to write the function *optimiseFunction*, after having introduced all the variables, our goal is to produce something of type B. This can be done by using the function f, but then we need a $i : I$ and something of type $D\ i$. We have both, however the $D\ i$ we have is marked as irrelevant, so it may only be passed along to irrelevant contexts, which the function f does not provide, so we cannot use that one. We need to find another way to produce an $D\ i$. We might try to extract it from the proof of $\mathit{isInternallyCollapsible}\ I\ D$, but this proof only tells us how the inhabitants of every $D\ i$ are related to each other with propositional equality. From this proof we cannot tell whether some $D\ i$ is inhabited or empty.

The optimisation given for concretely collapsible families need not worry about this. In that case we have a lot more information to work with. We only have to worry about well-typed calls to the elimination operator, so we do not have to deal with deciding whether $D\ i$ is empty or not. Apart from this we only need to recover the non-recursive parts of the erased, canonical term.

If we extend the definition of internal collapsibility with something that decides whether $A\ i$ is empty or not, we get the following definition:

$$\begin{array}{l}
\mathit{isInternallyCollapsibleDecidable} : (I : \mathit{Type})\ (A : I \to \mathit{Type}) \to \mathit{Type} \\
\mathit{isInternallyCollapsibleDecidable}\ I\ A = (i : I) \\
\quad \to (((x\ y : A\ i) \to x \equiv y)\ \times\ (A\ i\ +\ A\ i \to \bot))
\end{array}$$

If we then replace the occurrence of *isInternallyCollapsible* in the type signature of *optimiseFunction* with *isInternallyCollapsibleDecidable*

4.6.1 Time complexity issues

Using this definition we do get enough information to write $\mathit{optimiseFunction}$. However, the success of the optimistically named function $\mathit{optimiseFunction}$ relies on time complexity the proof given of $\mathit{isInternallyCollapsibleDecidable}\ D\ I$ that is used to recover the erased $A\ i$ value from the index i. In the case of concrete collapsibility this was not that much of an issue, since the way we retrieve the erased values from the indices was constant in the size of the given indices.

Apart from requiring a decision procedure that gives us, for every index $i : I$, an inhabitant of $A\ i$ or a proof that $A\ i$ is empty, we need a bound on the time complexity of this procedure. If we want to analyse the complexity of the functions, we need an embedding of the language they are written in. Examples of this approach can be found in Swierstra [2011] and Danielsson [2008]. In Danielsson [2008] the functions are written using a monad that keeps track of how many "ticks" are needed to evaluate the function for the given input, called the Thunk monad. $\mathit{Thunk} : \mathbb{N} \to \mathit{Type} \to \mathit{Type}$ is implemented as an abstract type that comes with the following primitives:

- $\mathit{step} : (a : \mathit{Type}) \to (n : \mathbb{N}) \to \mathit{Thunk}\ n\ a \to \mathit{Thunk}\ (n+1)\ a$
- $\mathit{return} : (a : \mathit{Type}) \to (n : \mathbb{N}) \to a \to \mathit{Thunk}\ n\ a$
- $(\gg\!=) : (a\ \ b : \mathit{Type}) \to (n\ \ m : \mathbb{N}) \to \mathit{Thunk}\ m\ a \to (a \to \mathit{Thunk}\ n\ b) \to \mathit{Thunk}\ (m+n)\ b$
- $\mathit{force} : (a : \mathit{Type}) \to (n : \mathbb{N}) \to \mathit{Thunk}\ n\ a$

The user has to write its programs using these primitives. A similar approach has also been used in van Laarhoven [2013] to count the number of comparisons needed for various comparison-based sorting algorithms.

Using this to enforce a time bound on the decision procedure is not entirely trivial. We first need to establish what kind of time limit we want: do we want a constant time complexity, as we have with the concrete collapsibility optimisation? If we want it to be non-constant, on what variable do we want it to depend?

Apart from these questions, approaches such as the Thunk monad, are prone to "cheating": we can just write our decision procedure the normal way and then write $\mathit{return}\ 1\ \mathit{decisionProcedure}$ to make sure it has the right type. To prevent this, we can deepen our embedding of the programming language in such a way, that the users can write the program completely in this language. Such a language, if it is complete enough, will most likely make writing programs unnecessarily complex for the user.

Even though we can internalise certain conditions under which certain transformations are safe (preserve definitional equality), along with the transformations, guaranteeing that this transformation actually improves complexity proves to be a lot more difficult.

4.7 Indexed h-propositions and homotopy type theory

In section 2.4 we have seen that h-propositions are exactly those types that obey proof irrelevance. If we generalise this internal notion to the indexed case we arrive at something we previously have called internal collapsibility. We have also seen that if we restrict ourselves to the empty context, internal collapsibility implies collapsibility. The purpose of the collapsibility optimisations is to optimise the evaluation of terms in the empty context. In homotopy type theory however, we postulate extra equalities in order to implement univalence or higher inductive types. "Run-time" for these programs does therefore not mean evaluation in the empty context, but evaluation in a context that can possibly contain the aforementioned postulates. To stress this difference in what contexts we are considering to do the evaluation in, we will talk about internal collapsible for the empty context case and indexed h-propositions in for the homotopy type theory case. In this section we will investigate what these differences mean when trying to optimise our programs.

When postulating extra propositional equalities, we obviously lose the canonicity property, hence we can no longer say that propositional equality implies definitional equality at run-time. The essence of the concrete collapsibility optimisation is that we need not store certain parts of our programs, because we know that they are unique, canonical and can be recovered from other parts of our program. In homotopy type theory we no longer have this canonicity property and may have to make choice in what inhabitant we recover from the indices. As an example of this we will compare two non-indexed types: the unit type and the interval. Both types are h-propositions, so they admit proof irrelevance, but the interval does have two canonical inhabitants that can be distinguished by definitional equality.

$$\begin{array}{l} \textbf{data}\ I : \mathit{Set}\ \textbf{where} \\ \quad \mathit{zero} : \mathit{Interval} \\ \quad \mathit{one}\ \ : \mathit{Interval} \\ \quad \mathit{segment} : \mathit{zero} \equiv \mathit{one} \end{array}$$

The elimination operator for this type is defined in this way:

$$\begin{array}{l} \mathit{I\text{-}elim} : (B : I \to \mathit{Type}) \\ \quad \to (b_0 : B\ \mathit{zero}) \\ \quad \to (b_1 : B\ \mathit{one}) \\ \quad \to (p : (\mathit{transport}\ B\ \mathit{segment}\ b_0) \equiv b_1) \\ \quad \to (i : I) \to B\ i \end{array}$$

with computation rules[6]:

$$\begin{array}{l} \mathit{I\text{-}elim}\ B\ b_0\ b_1\ p\ \mathit{zero} \triangleq b_0 \\ \mathit{I\text{-}elim}\ B\ b_0\ b_1\ p\ \mathit{one}\ \ \triangleq b_1 \end{array}$$

[6] Apart from giving computation rules for the points, we also need to give a computation rule for the path constructor, *segment*, but as we do not need this rule for the discussion here, we have left it out.

In other words, in order to eliminate a value in the interval, we need to tell what has to be done with the endpoints interval and then have to show that this is done in such a way that the path between the endpoints is preserved.

Let us compare the above to the elimination operator for the unit type, $\top$:

$$
\begin{aligned}
&\top\text{-}\mathit{elim} : (B : \top \to \mathit{Type})\\
&\quad \to (b : B\ \mathit{tt})\\
&\quad \to (t : \top) \to B\ t
\end{aligned}
$$

with computation rule:

$$\top\text{-}\mathit{elim}\ B\ b\ \mathit{tt} \overset{\Delta}{\equiv} b$$

If we have canonicity, we can clearly assume every inhabitant of $\top$ to be tt at run-time and erase the t argument from $\top$-*elim*. In the case of I, we cannot do this: we have two canonical inhabitants that are propositionally equal, but not definitionally.

Not all is lost, if we consider the non-dependent elimination operator for the interval:

$$
\begin{aligned}
&I\text{-}\mathit{elim}\text{-}\mathit{nondep} : (B : \mathit{Type})\\
&\quad \to (b_0 : B)\\
&\quad \to (b_1 : B)\\
&\quad \to (p : b_0 \equiv b_1)\\
&\quad \to I \to B
\end{aligned}
$$

then it is easy to see that all such functions are constant functions, with respect to propositional equality. If we erase the I argument and presuppose it to be *zero*, we will get a new function that is propositionally equal to the original one. However, it is definitional equality that we are after. We can define the following two functions:

$$
\begin{aligned}
&I\text{-}\mathit{id} : I \to I\\
&I\text{-}\mathit{id} = I\text{-}\mathit{elim}\text{-}\mathit{nondep}\ I\ \mathit{zero}\ \mathit{one}\ \mathit{segment}\\
&I\text{-}\mathit{const}\text{-}\mathit{zero} : I \to I\\
&I\text{-}\mathit{const}\text{-}\mathit{zero} = I\text{-}\mathit{elim}\text{-}\mathit{nondep}\ I\ \mathit{zero}\ \mathit{zero}\ \mathit{refl}
\end{aligned}
$$

If we presuppose and erase the I argument to be *zero* in the I-*id* case, we would get definitionally different behaviour. In the case of I-*const*-*zero*, it does not matter if we presuppose the argument to be *zero* or *one*, since this function is also definitionally constant. This is because for the *refl* to type check, b_0 and b_1 have to definitionally equal. So if we want to optimise the elimination operators of higher inductive types that are h-propositions, such as the interval, we need to look at what paths the non-trivial paths are mapped to. If these are all mapped to *refl*, then the points all get mapped to definitionally equal points.

Suppose that f is the function that we are constructing using the elimination principle of some higher inductive type H, which happens to be a h-proposition. We want to verify that $ap\ f$ maps every path to *refl*. Checking this property can become difficult, as we can tell from this rather silly example:

$$\begin{array}{l}
\textbf{data}\ \mathbb{N}\text{-}\mathit{truncated} : \mathit{Type}\ \textbf{where} \\
\quad 0 : \mathbb{N}\text{-}\mathit{truncated} \\
\quad S : (n : \mathbb{N}\text{-}\mathit{truncated}) \to \mathbb{N}\text{-}\mathit{truncated} \\
\quad \mathit{equalTo0} : (n : \mathbb{N}\text{-}\mathit{truncated}) \to 0 \equiv n
\end{array}$$

with non-dependent eliminator:

$$\begin{array}{l}
\mathbb{N}\text{-}\mathit{truncated}\text{-}\mathit{elim}\text{-}\mathit{nondep} : (B : \mathit{Type}) \\
\quad \to (b_0 : B) \\
\quad \to (b_S : B \to B) \\
\quad \to (p : (b : B) \to b_0 \equiv b) \\
\quad \to \mathbb{N}\text{-}\mathit{truncated} \to B
\end{array}$$

If we were to check that all paths between 0 and n are mapped to *refl*, we have to check that p satisfies this property, which we cannot do.

4.7.1 Internally optimising h-propositions

The optimisation given in section 4.6 of course still is a valid transformation for the homotopy type theory case. The proof of a family $D : I \to \mathit{Type}$ being an indexed h-proposition is again not enough for us to be able to write the term $\mathit{optimiseFunction}$. What we called *isInternallyCollapsibleDecidable* is that we internally need a witness of the fact that every h-proposition in the family is either contractible or empty, so we could have written the property as follows:

$$\begin{array}{l}
\mathit{isIndexedhPropDecidable} : (I : \mathit{Type})\ (A : I \to \mathit{Type}) \to \mathit{Type} \\
\mathit{isIndexedhPropDecidable}\ I\ A = (i : I) \\
\quad \to (\mathit{isContractible}\ (A\ i))\ +\ (A\ i \to \bot)
\end{array}$$

4.8 Conclusions

In this chapter we have looked at various ways of dealing with types that are purely logical, called propositions. Coq and Agda both provide mechanisms to in a way "truncate" a type into a proposition. The first takes this approach by allowing the user to annotate a type as being a proposition when defining the type. Making sure it is a proposition and has no computational effect on non-propositions is handled by limiting the elimination of these propositions: we may only eliminate into other propositions. Singleton elimination is an exception to this rule, which does not play well with homotopy type theory and the univalence axiom, as it means that the equality used by Coq gets falsely recognised as a singleton type, even though it is provably not one. Using univalence we can construct a term that behaves differently in Coq as it does in the extracted version.

Agda allows the user to indicate that a type is a proposition when referring to that type, instead of having to annotate it when defining it. Agda enforces the proof irrelevance by ensuring that inhabitants of an annotated type are never scrutinised in a pattern match and may only be passed onto other irrelevant contexts. It contrast to Coq's mechanism, it does not allow for singleton elimination, but unlike Coq, it does enable the user to prove properties of the annotated types in Agda itself. As such, we can construct a squash type that is isomorphic to the (-1)-truncation from homotopy type theory, defined as a higher inductive type.

Instead of truncating a type such that it becomes a proposition, we can also let the compiler recognise whether a type is a proposition or not. This is the approach that the collapsible families optimisation takes in Epigram. The definition of collapsibility is reminiscent of the definition of h-proposition, albeit it an indexed version that uses definitional equality instead of propositional equality. The optimisation specifically focuses on families of propositions.

Recognising whether an inductive family is a collapsible family is undecidable, so the actual optimisation restricts itself to a subset called concretely collapsible families. To improve on this, we internalise the notion of collapsibility, allowing the user to provide a proof if the compiler fails to notice this property. We show that this notion of internal collapsibility is a subset of collapsibility. We also try to internalise the optimisation, but since the time complexity of the optimised function heavily depends on the user-provided proof, we cannot be sure whether it the "optimised" version actually improves on the complexity. We have looked at ways to enforce time complexities in the user-provided proofs. Our conclusion is that this is not viable.

Collapsible families look a lot like families of h-propositions. When internalising the collapsibility concept and the optimisation, we only considered the non-homotopy type theory case, i.e. no univalence and no higher inductive types. We have looked at extending the optimisations to the homotopy type theory case, but as we lose canonicity the optimised versions may no longer yield the same results as the original function, with respect to definitional equality. We have identified cases in which this is the case and cases in which definitional equality actually is preserved. We also argue that detecting such cases is undecidable.

Chapter 5

Discussion

One of the main goals of this project was to establish whether homotopy type theory is an interesting language to do dependently typed programming in. As it is incompatible with dependent pattern matching in general, it seems like we are taking a step backwards. However, univalence and higher inductive types can become the two steps forward. Univalence means that we can transport definitions along isomorphisms, which saves us a great deal of writing boring code applying the *to* and *from* parts of the isomorphisms in the right places. It also implies function extensionality, which is indispensable when proving properties about programs.

We have also seen the usefulness of higher inductive types. They allow us to define quotient types. It is all too easy to come up with a higher inductive type that has more structure than is desired: one quickly runs into *coherence issues*: the resulting type has too many different equalities at higher levels than is needed. The original definition of quotient types also suffered from this issue: we want it to be an h-set, but as could be seen from a simple example, one could easily define the circle: the simplest type that is not an h-set. Therefore one usually needs to truncate the higher inductive type to a certain level, e.g. take the 0-truncation in the case of quotients. Truncating a type does mean that we have extra conditions that we need to satisfy when eliminating something of that particular type. In a programming setting, one typically only encounters h-sets, except for univalent universes of h-sets. Eliminating into an h-set means that the extra conditions stemming from 0-truncation are automatically satisfied, so in programming this need not be too much of a problem.

For these two steps to be actual steps forward, there is still a lot of work that needs to be done. The most obvious and possibly most difficult problem is determining the computational content of the univalence axiom. Seeing as most types in programming applications are h-sets, it is already a big improvement if we get this to work for a type theory in which everything is 1-truncated and the only 1-type which is not an h-set is a univalent universe of all h-sets.

Giving up pattern matching altogether is quite drastic. There are still a lot of cases in which (dependent) pattern matching is still valid and can be transformed to an expression using only elimination principles. An interesting future research direction is to take the elaboration process described in Goguen et al. [2006], which critically depends on K, and see how one can uncover conditions in which K is not necessary for the elaboration to work.

There is also a lot of work to be done on higher inductive types. As of yet, a well-defined syntax for higher inductive types and a generic way to derive the induction principles is lacking. It has also been noted [Lumsdaine, 2012] that every higher inductive type that has higher path constructors in its definition, can be rewritten to an equivalent form that only has path constructors that construct paths between points (a so called 1-HIT). Having a mechanism that automatically translates the definition of a higher inductive type to a 1-HIT, also means that we only have to care about these cases when devising induction principles. Having a form of pattern matching for higher inductive types is also a research direction that can help make higher inductive types significantly more easy to work with.

In chapter 4, we have seen that in traditional Martin-Löf's type theory, propositional equality coincides with definitional equality at "run-time" (i.e. in the empty context). This property makes it possible to internalise optimisations: one could create a system in which we provide rules to the compiler akin to the GHC rewrite rules [Jones et al., 2001], but along with a proof of correctness. In homotopy type theory, we also want to have non-canonical proofs of propositional equality at run-time, so we lose this property. A further investigation of when propositional equality still does imply definitional equality might be an interesting research direction. Another interesting thing to look at is the question whether we really need definitional equality, i.e. identify cases in which we can safely replace something by something else that is propositionally but not necessarily definitionally equal.

Coming back to the main research question:

> What is homotopy type theory and why is it interesting to do programming in it?

In this thesis, we have given evidence that homotopy type theory is an interesting language to program in, but as of yet we have to sacrifice too much (i.e. dependent pattern matching and canonicity in its entirety) for it to be useful for programming right now, but the future looks promising, even if we only get to implement restricted versions of homotopy type theory.

Acknowledgements

Front and foremost, I would like to thank my supervisor Wouter for the guidance writing this thesis. Secondly, Paul deserves a mention for hosting me in Bordeaux, renewing my motivation to work on this project and letting me experience French culture, including but not limited to air traffic control and train strikes. The people from BBL682 and BBL681 provided much needed distraction. Last but not least, I would like to thank my parents for their unconditional support.

Bibliography

T. Altenkirch, T. Anberrée, and N. Li. Definable quotients in type theory.

S. Awodey. Type theory and homotopy. In *Epistemology versus Ontology*, pages 183–201. Springer, 2012.

Y. Bertot and P. Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer-Verlag New York Incorporated, 2004.

A. Bove and V. Capretta. Modelling general recursion in type theory. *Mathematical Structures in Computer Science*, 15(4):671–708, 2005.

E. Brady, C. McBride, and J. McKinna. Inductive families need not store their indices. In *Types for Proofs and Programs*, pages 115–129. Springer, 2004.

R. Constable, S. Allen, H. Bromley, W. Cleaveland, J. Cremer, R. Harper, D. Howe, T. Knoblock, N. Mendler, P. Panangaden, et al. *Implementing mathematics with the Nuprl proof development system*. Prentice Hall, 1986.

T. Coquand. Pattern matching with dependent types. In *Informal proceedings of Logical Frameworks*, volume 92, pages 66–79, 1992.

N. A. Danielsson. Lightweight semiformal time complexity analysis for purely functional data structures. In *ACM SIGPLAN Notices*, volume 43, pages 133–144. ACM, 2008.

N. A. Danielsson. Postulated computing quotients are unsound. online, https://lists.chalmers.se/pipermail/agda/2012/004052.html, 2012. [Agda mailing list post].

H. Goguen, C. McBride, and J. McKinna. Eliminating dependent pattern matching. In *Algebra, Meaning, and Computation*, pages 521–540. Springer, 2006.

A. Grothendieck. Pursuing Stacks, letter to D. Quillen. *Documents Mathématiques, Soc. Math. France, Paris, France*, 1983.

M. Hofmann and T. Streicher. The groupoid interpretation of type theory. In *In Venice Festschrift*, 1996.

S. P. Jones, A. Tolmach, and T. Hoare. Playing by the rules: rewriting as a practical optimisation technique in ghc. In *Haskell Workshop*, volume 1, pages 203–233, 2001.

C. Kapulkin, P. L. Lumsdaine, and V. Voevodsky. Univalence in simplicial sets. *arXiv preprint arXiv:1203.2553*, 2012.

N. Kraus and C. Sattler. Universe n is not an n-type. online, http://homotopytypetheory.org/2013/05/15/universe-n-is-not-an-n-type/, 2013. [blog post].

N. Kraus, M. Escardó, T. Coquand, and T. Altenkirch. Generalizations of hedberg's theorem. In *Typed Lambda Calculi and Applications*, pages 173–188. Springer, 2013.

P. Letouzey. A new extraction for Coq. In *Types for proofs and programs*, pages 200–219. Springer, 2003.

D. R. Licata. Running Circles Around (In) Your Proof Assistant; or, Quotients that Compute. online, http://homotopytypetheory.org/2011/04/23/running-circles-around-in-your-proof-assistant/, 2011. [blog post].

D. R. Licata. Abstract Types with Isomorphic Types. online, http://homotopytypetheory.org/2012/11/12/abstract-types-with-isomorphic-types/, 2012. [blog post].

D. R. Licata and R. Harper. Canonicity for 2-dimensional type theory. In *ACM SIGPLAN Notices*, volume 47, pages 337–348. ACM, 2012.

D. R. Licata and M. Shulman. Calculating the fundamental group of the circle in homotopy type theory. *arXiv preprint arXiv:1301.3443*, 2013.

P. L. Lumsdaine. Reducing all HIT's to 1-HIT's. online, http://homotopytypetheory.org/2012/05/07/reducing-all-hits-to-1-hits/, 2012. [blog post].

P. Martin-Löf. Constructive mathematics and computer programming. In *Proc. of a discussion meeting of the Royal Society of London on Mathematical logic and programming languages*, pages 167–184. Prentice-Hall, Inc., 1985.

J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 10(3):470–502, 1988.

C. Paulin-Mohring. Inductive Definitions in the System Coq - Rules and Properties. In M. Bezem and J.-F. Groote, editors, *Proceedings of the conference Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, 1993. LIP research report 92-49.

Á. Pelayo and M. A. Warren. Homotopy type theory and Voevodsky's univalent foundations. *arXiv preprint arXiv:1210.5658*, 2012.

E. M. Rijke. Homotopy type theory. Master's thesis, Universiteit Utrecht, 2012.

M. Schulman. Univalence versus Extraction. online, http://homotopytypetheory.org/2012/01/22/univalence-versus-extraction/, 2012. [blog post].

M. Sozeau, N. Tabareau, et al. Univalence for free. 2013.

W. Swierstra. Sorted. *Journal of Functional Programming*, 21(06):573–583, 2011.

The Univalent Foundations Program. Homotopy type theory: Univalent foundations of mathematics. Technical report, 2013.

T. van Laarhoven. The complete correctnes of sorting. online, http://twanvl.nl/blog/agda/sorting, 2013. [blog post].

V. Voevodsky. Univalent foundations. online, http://www.math.ias.edu/~vladimir/Site3/Univalent_Foundations_files/2011_UPenn.pdf, 2011. [presentation at University of Pennsylvania].

P. Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 307–313. ACM, 1987.