Erasing propositions and homotopy type theory

Gabe Dijkstra

August 6, 2013

Abstract

Write abstract

Contents

Chapter 1

Introduction

Write introduction

Chapter 2

Homotopy type theory

Chapter 3

Applications of homotopy type theory

Introductory paragraph explaining what applications we will look at, apart from that we have already seen function extensionality.

Contributions:

- Investigate how higher inductive types can be used to construct quotients and how this compares to other approaches (setoids, definable quotients).
- Elaborate on Licata's use of univalence for views on abstract types and extend it to non-isomorphic views.

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, it often happens that we have defined a data type that has more structure than we want to expose. This situation typically occurs when 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 $insert : KeyValuePair \to BST \to BST$ 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, 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:

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

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:

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

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:

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

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 $\_\sim\_$.

3.1.3 Binary operations on quotients

We have seen how to lift a function $f : A \to B$ to $\widetilde{f} : Quotient\ A\ \_\sim\_\ B$ given a proof of $(x\ y : A) \to x \sim y \to f\ x \equiv f\ y$, using $Quotient\text{-}rec$. 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 $\widetilde{f} : Quotient\ A\ \_\sim\_ \to Quotient\ A\ \_\sim\_ \to B$.

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:

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

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

$$\begin{aligned}
&lift\text{-}left : (f : A \to A \to B) \\
&\quad (resp : (x\ y\ x'\ y' : A) \to x \sim x' \to y \sim y' \to f\ x\ y \equiv f\ x'\ y') \\
&\quad Quotient\ A\ _\sim_ \to A \to B \\
&lift\text{-}left\ f\ resp\ q = Quotient\text{-}rec\ f\ goal_0\ q
\end{aligned}$$

where $goal_0 : (x\ \ x' : A) \to x \sim x' \to f\ \ x \equiv f\ \ x'$. 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 $lift\text{-}left$ in the definition of $Quotient\text{-}rec$-2:

$$Quotient\text{-}rec\text{-}2\ f\ resp\ q\ q' = Quotient\text{-}rec\ (lift\text{-}left\ f\ resp\ q)\ goal_1\ q'$$

where $goal_1 : (y\ y' : A) \to y \sim y' \to lift\text{-}left\ f\ resp\ q\ y \equiv lift\text{-}left\ f\ resp\ q\ y'$, which can be proven using $Quotient\text{-}ind$. We then only have to consider the case where q is of the form $[a]$ for some $a : A$. In that case, $lift\text{-}left\ f\ resp\ q\ y$ reduces to $f\ \ a\ \ y$ and $lift\text{-}left\ f\ resp\ q\ y'$ 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:

$$\begin{array}{l} goal_1 : (y\ y' : A) \to y \sim y' \to lift\text{-}left\ f\ resp\ q\ y \equiv lift\text{-}left\ f\ resp\ q\ y' \\ goal_1 = \lambda y\ y'\ r \to \\ \quad Quotient\text{-}ind\ (\lambda w \to lift\text{-}left\ f\ resp\ w\ y \equiv lift\text{-}left\ f\ resp\ w\ y') \\ \qquad\qquad (\lambda a \to resp\ a\ y\ a\ y'\ (\sim\text{-}refl\ a)\ r) \\ \qquad\qquad goal_2 \\ \qquad\qquad q \end{array}$$

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 a 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 $\_\sim\_$. This stems from the fact that the relation we take the quotient by, is the smallest *equivalence* relation generated by $\_\sim\_$.

3.1.4 Coherence issues

In the HoTT book, they specifically mention the quotient should be the 0-truncation. Why do we need this? Can we find a h-set A with a relation $\_\sim\_ : A \to A \to Prop$ such that $Quotient\ A\ \_\sim\_$ is not a h-set.

[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.

If we use the 0-truncation, then we can also easily prove that $[\_]$ preserves all the equivalence relation structure: reflexivity of the one structure gets mapped to the other reflexivity, same for symmetry. Transitivity also gets mapped in the right way, i.e. things commute.

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, e.g. a sequence abstract type can be described as follows:

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

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 Dan Licata[2], 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:

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

We want List and Other to be "isomorphic"[3], i.e. we need to write the following terms:

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

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

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

These properties can be added to the original $\mathit{Sequence}$ 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 between these to guide us to the desired properties. We know that in order to prove that $a\ b : \Sigma\ A\ B$ are propositionally equal, we need to show its fields are propositionally equal as well:

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

[2] `http://homotopytypetheory.org/2012/11/12/abstract-types-with-isomorphic-types/`

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

If we want to prove that $ListImpl \equiv OtherImpl$, then using Σ- $\equiv$, 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 $A : Type$:

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

with $ListImpl_A$ and $OtherImpl_A$ defined straightforwardly from the previous definitions. To show that $ListImpl_A$ and $OtherImpl_A$, we need to show using univalence that $List\ A \equiv Other\ A$, so the beginning of the proof looks like this:

$$\begin{aligned} &spec : \\ &\quad (from : List\ A \to Other\ A) \\ &\quad (to : Other\ A \to List\ A) \\ &\quad \to Iso\ (List\ A)\ (Other\ A)\ from\ to \\ &\quad \to ListImpl_A \equiv OtherImpl_A \\ &spec\ from\ to\ iso = \Sigma\text{-} \equiv\ (univalence\ (List\ A)\ (JoinList\ A)\ iso) \\ &\qquad\qquad\qquad\qquad (\Sigma\text{-} \equiv\ goal_0 \\ &\qquad\qquad\qquad\qquad (\Sigma\text{-} \equiv\ goal_1 \\ &\qquad\qquad\qquad\qquad (\Sigma\text{-} \equiv\ goal_2 \\ &\qquad\qquad\qquad\qquad\qquad goal_3))) \end{aligned}$$

The first goal, $goal_0$, has type $fst\ (transport\ (univalence\ (List\ A)\ (JoinList\ A)\ iso)\ ([\,], (\lambda x \to [x], (\_{+\!+}\_, map)))) \equiv otherEmpty$. 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 *transport* applied to the dependent pair, is *transport* applied to the first field of the dependent pair:

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

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

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

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.

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 it is not unthinkable that a lot of this can be automated.

3.2.2 Reasoning with views

If we want to prove a property about our abstract type, we can 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:

$$
\begin{aligned}
&\mathit{left\text{-}unit\text{-}append} : (\mathit{xs} : \mathit{List}\ A) \to [\,] +\!\!+ \mathit{xs} \equiv \mathit{xs} \\
&\mathit{left\text{-}unit\text{-}append}\ \mathit{xs} = \mathit{refl}
\end{aligned}
$$

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:

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

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:

$$
\begin{aligned}
&\textbf{data}\ \mathit{JoinList}\ (A : \mathit{Type}) : \mathit{Type}\ \textbf{where}\\
&\quad \mathit{nil}\ \ : \mathit{JoinList}\ A\\
&\quad \mathit{unit} : A \to \mathit{JoinList}\ A\\
&\quad \mathit{join} : \mathit{JoinList}\ A \to \mathit{JoinList}\ A \to \mathit{JoinList}\ A
\end{aligned}
$$

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 $\mathit{JoinList}$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: to and from do not form an isomorphism: $\mathit{JoinList}$ has a finer structure than List. If only the first equality holds, 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 $\mathit{List}\ A$. 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 $\mathit{Quotient} - \mathit{rec}$ 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 $A/\sim$ to-$A/\sim$ and from-$A/\sim$. 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 (A is a set and $\_\sim\_$ is an equivalence relation) for the induction step on $A/\sim$. The fact that to-$A/\sim$ is a retraction with from-$A/\sim$ as its section can be proved using the same techniques.

To lift the operations on A to operations on $A/\sim$ we simply apply to-$A/\sim$ and from-$A/\sim$ 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 $JoinList$ example. Let us define $JoinList\ A/\sim\ A$ as $JLAquote$ with $x \sim y$ defined as $to\ x \equiv to\ y$. We have the following functions:

- $to : JoinList\ A \to List\ A$
- $from : List\ A \to JoinList\ A$
- $\widetilde{to} : JoinList\ A \to JoinList\ A/\sim\ A$
- $\widetilde{from} : JoinList\ A/\sim\ A \to JoinList\ A$

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

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

In general we have that $\widetilde{from}\ (\widetilde{to}\ x) \equiv from\ (to\ x)$ 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 $\widetilde{from}$ in other positions:

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

We end up with having to prove that $(xs\ ys : JoinList\ A/\sim\ A) \to to\ (join\ (\widetilde{from}\ xs)\ (\widetilde{from}\ ys)) \equiv to\ (\widetilde{from}$ which follows from $(xs\ ys : JoinList\ A) \to to\ (join\ xs\ ys) \equiv to\ xs +\!\!+ to\ ys$.

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. We can use the type $\Sigma\ (x : A)\ .s\ (r\ x) \equiv x$, i.e. restrict A to those inhabitants for which s and r are isomorphisms. The func-

tion $box : A \to \Sigma\ (x : A)\ .s\ (r\ x) \equiv x$ is then defined by $\lambda x \to (s\ (r\ x))$, $ap\ s\ (is\text{-}retract\ (r\ x))$, where $is\text{-}retract : (x : B) \to r\ (s\ x) \equiv x$.

3.3 Conclusion

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 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 $isSorted : List\ \mathbb{N} \to List\ \mathbb{N} \to Type$ 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:

$$sort : (xs : List\ \mathbb{N}) \to \Sigma\ (ys : List\ \mathbb{N})\ (isSorted\ xs\ ys)$$

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 runtime the type checking has been done, hence the proof of correctness has already been verified: we want to *erase* the logical parts.

Types such as $isSorted\ xs\ ys$ 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*, and how they can occur in various places in certified programs. In sections 4.2 and 4.3 we review the methods Coq and Agda provide us to annotate parts of our program as being propositions. 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.

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*[1] :

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

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. In our quicksort case we get the following inductive family:

[1] 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.

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

with the following function definition[2]

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

Pattern matching on the $qsAcc\ xs$ 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 $i < length\ xs$ to deal with the (impossible) case where xs is empty. In the qs case, we need $qsAcc\ xs$ to guide the recursion. Even though we actually pattern match on $qsAcc\ xs$ 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 are base sorts of the hierarchy of sorts, $Type$, i.e. $Prop : Type\ (1)$, $Set : Type\ (1)$ and for every i, $Type\ (i) : Type\ (i + 1)$. 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. More importantly, the 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 $List\ \mathbb{N}$. For the Σ-type, Coq provides two options: sig and ex, defined as follows:

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

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.

[2]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 $qsAcc\ xs$ argument.

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 $Prop$ ($/\backslash$) 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 Michael Shulman[3], 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 $false$.

[3] `http://homotopytypetheory.org/2012/01/22/univalence-versus-extraction/`

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:

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

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 $R \notin R$. In type theory, an analogous paradox, Girard's paradox, arises if we allow for impredicativity via the $Type : Type$ rule. 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:

$$\begin{array}{l}\textbf{Definition}\ demorgan : Prop := \forall\ P\ Q : Prop, \\ \quad \sim (P\ /\backslash\ Q) \to\ \sim P\ \backslash/\ \sim Q\ .\end{array}$$

Coq allows for such definitions as the restrictions on $Prop$ prevent us from constructing paradoxes such as Girard's.

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:

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

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:

$$\begin{array}{l}\textbf{record}\ \Sigma_{\textbf{irr}}\ (A : Type)\ (B : A \to Type) : Type\ \textbf{where} \\ \quad \textbf{constructor}\ \_,\_ \\ \quad field \\ \quad\quad fst\ \ \ : A \\ \quad\quad .snd : B\ fst\end{array}$$

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 $(A : Type) \to\ .A \to A$.

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:

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

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:

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

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 $\mathit{irrelevantConstantDepFunction}$, 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 checking, 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:

$$\begin{aligned}&\textbf{record}\ \mathit{Squash}\ (A : \mathit{Type}) : \mathit{Type}\ \textbf{where}\\&\quad \textbf{constructor}\ \mathit{squash}\\&\quad \mathit{field}\\&\qquad .\mathit{proof} : A\end{aligned}$$

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

$$\begin{aligned}&\mathit{squashProofIrrelevance} : \{A : \mathit{Type}\}\ (x\ y : \mathit{Squash}\ A) \to x \equiv y\\&\mathit{squashProofIrrelevance}\ x\ y = \mathit{refl}\end{aligned}$$

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 and can also be achieved by defining the following higher inductive type:

$$
\begin{array}{l}
\textbf{data}\ (-1)\text{-}truncation\ (A : Type) : Type\ \textbf{where} \\
\quad inhab : A \\
\quad all\text{-}paths : (x\ y : A) \to x \equiv y
\end{array}
$$

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):

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

with elimination operator

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

and computation rules

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

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 $< \text{-}elim$, 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 $< \text{-}elim$.

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 $< \text{-}elim$ 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 $< \text{-}elim$ 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 $\vdash\ p \stackrel{\Delta}{=} q$. Families $D : I_0 \to \cdots \to I_n \to Type$ such as $< \text{-}elim$ 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 $\vdash\ p \stackrel{\Delta}{=} q$.

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. In this section we investigate if we can formulate an internal version of collapsibility, enabling the user to give a proof that a certain family is collapsible, if the compiler fails to notice so itself.

Recall the definition of a collapsible family[4] : given an inductive family D indexed by the type I, D is collapsible if for every index $i : I$ and terms x, y, the following holds:

$$\vdash x, y : D\ i \text{ implies } \vdash x \stackrel{\Delta}{=} y$$

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. 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 $x \stackrel{\Delta}{=} y$, hence in the empty context the equality reflection rule does hold. The converse is also true: definitional equality implies of x and y that $\vdash \mathit{refl} : x \equiv y$ 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

[4]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.

differ from collapsible families as can be seen by considering D to be the family Id. By canonicity we have that for any $A : Type$, $x, y : A$, a term p satisfying $\vdash p : Id\ A\ x\ y$ 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 typecheck. However, we can use Agda's irrelevance mechanism to instead generate a function in which the collapsible argument is marked as irrelevant, i.e. we want to write the following function (for the non-dependent case):

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

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:

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

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 $isInternallyCollapsible\ I\ D$, but this proof only tells us how the inhabitants of every $D\ i$ are related to eachother 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:

$$\begin{aligned}&\mathit{isInternallyCollapsibleDecidable} : (I : \mathit{Type})\ (A : I \to \mathit{Type}) \to \mathit{Type}\\&\mathit{isInternallyCollapsibleDecidable}\ I\ A = (i : I)\\&\quad \to (((x\ y : A\ i) \to x \equiv y)\ \otimes\ (A\ i\ \oplus\ A\ i \to \bot))\end{aligned}$$

If we then replace the occurrence of $\mathit{isInternallyCollapsible}$ in the type signature of $\mathit{optimiseFunction}$ with $\mathit{isInternallyCollapsibleDecidable}$

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. One approach, taken in Danielsson (2008) to prove time complexities of functions, is to write the functions with 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 by van Laarhoven[5] 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 too 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?

[5] http://twanvl.nl/blog/agda/sorting

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 $return\ 1\ decisionProcedure$ to make sure it has the right type. To prevent this, we can extend the list of primitives 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 **??** 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. In homotopy type theory, we are interested in postulating extra equalities needed to talk about univalence or higher inductive types. To stress the difference in what contexts we are considering, we will talk about internal collapsible for the empty context case and indexed h-propositions in the other 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.

$$
\begin{aligned}
&\mathbf{data}\ I : Set\ \mathbf{where}\\
&\quad zero : Interval\\
&\quad one\ : Interval\\
&\quad segment : zero \equiv one
\end{aligned}
$$

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

$$
\begin{aligned}
&I\text{-}elim : (B : I \to Type)\\
&\quad \to (b_0 : B\ zero)\\
&\quad \to (b_1 : B\ one)
\end{aligned}
$$

$$\to (p : (transport\ B\ segment\ b_0) \equiv b_1)$$
$$\to (i : I) \to B\ i$$

with computation rules[6]:

$$I\text{-}elim\ B\ b_0\ b_1\ p\ zero \triangleq b_0$$
$$I\text{-}elim\ B\ b_0\ b_1\ p\ one \triangleq b_1$$

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, $\top$:

$$\top\text{-}elim : (B : \top \to Type)$$
$$\to (b : B\ tt)$$
$$\to (t : \top) \to B\ t$$

with computation rule:

$$\top\text{-}elim\ B\ b\ tt \triangleq b$$

If we have canonicity, we can clearly assume every inhabitant of $\top$ to be tt at run-time and erase the t argument from $\top\text{-}elim$. 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:

$$I\text{-}elim\text{-}nondep : (B : Type)$$
$$\to (b_0 : B)$$
$$\to (b_1 : B)$$
$$\to (p : b_0 \equiv b_1)$$
$$\to I \to B$$

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:

$$I\text{-}id : I \to I$$
$$I\text{-}id = I\text{-}elim\text{-}nondep\ I\ zero\ one\ segment$$
$$I\text{-}const\text{-}zero : I \to I$$
$$I\text{-}const\text{-}zero = I\text{-}elim\text{-}nondep\ I\ zero\ zero\ refl$$

[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.

If we presuppose and erase the I argument to be $zero$ in the $I\text{-}id$ case, we would get definitionally different behaviour. In the case of $I\text{-}const\text{-}zero$, 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. Checking such a property can become difficult, as we can tell from this rather silly example:

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

with non-dependent eliminator:

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

If we were to check that all paths between 0 and n are mapped to a $refl$ between inhabitants of B, 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 $D : I \to Type$ being an indexed h-proposition is again not enough for us to be able to write the $optimiseFunction$ term. 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:

$$
\begin{aligned}
&isIndexedhPropDecidable : (I : Type)\ (A : I \to Type) \to Type\\
&isIndexedhPropDecidable\ I\ A = (i : I)\\
&\quad \to (isContractible\ (A\ i))\ \oplus\ (A\ i \to \bot)
\end{aligned}
$$

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. Proof irrelevance of the propositions in Coq is assumed when extracting a development, but not something that is enforced inside Coq, nor is it provable internally. 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.

As we have mentioned previously, 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 whether the latter is the case, is not tractable.

Chapter 5

Discussion

Write discussion

Acknowledgements

Write acknowledgements

Appendix A

Guide to source code

Write this.

Bibliography

Thorsten Altenkirch, Thomas Anberrée, and Nuo Li. Definable quotients in type theory.

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

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

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

RL Constable, SF Allen, HM Bromley, WR Cleaveland, JF Cremer, RW Harper, DJ Howe, TB Knoblock, NP Mendler, P Panangaden, et al. *Implementing mathematics with the Nuprl proof development system*. Prentice Hall, 1986.

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

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

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

Philip 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.

Index of symbols

$asdf$	propositional equality

Index