Erasing propositions in homotopy type theory

Gabe Dijkstra

June 18, 2013

Contents

Chapter 1

Introduction

Chapter 2

Homotopy type theory

T

Chapter 3

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

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

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 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 $\mathit{isSorted}\ \mathit{xs}\ \mathit{ys}$ 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 3.1 we will give more examples of such types, called *propositions*, and how they can occur in various places in certified programs. In sections 3.2 and 3.3 we review the methods Coq and Agda provide us to annotate parts of our program as being propositions. Section 3.4 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 3.5 we internalise the concept of collapsible families and try to do the same with the optimisation in section 3.6. The internalised version of collapsibility looks like an indexed version of the concept of h-propositions. In section 3.7 we investigate if we can use this to devise an optimisation akin to the optimisation based on collapsibility.

3.1 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.

3.1.1 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] :

[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.

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

The recursive calls are done on $\mathit{filter}\ (gt\ x)\ xs$ and $\mathit{filter}\ (le\ x)\ xs$ 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:

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

with the following function definition[2]

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

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.

[2]This definition uses dependent pattern matching, 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.

3.2 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:

Inductive $sig\ (A : Type)\ (P : A \to Prop) : Type :=$
  $exist : \forall\ x : A, P\ x \to sig\ P$

Inductive $ex\ (A : Type)\ (P : A \to Prop) : Prop :=$
  $ex\_intro : \forall\ x : A, P\ x \to ex\ P$

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 $Prop$.

3.2.1 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:

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

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 $false$. 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 de-

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

sign decision to define the identity types to be in *Prop* is not compatible with homotopy type theory.

3.2.2 Quicksort example

Should the following be explained in so much detail? It is important that we can apply the Bove-Capretta method in such a way that the extracted version has non of the predicates around, but its details are rather technical. I also wonder if the idea comes across well enough if I explain it in prose like this.

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.

3.2.3 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:

$$\textbf{Definition}\ \mathit{demorgan} : \mathit{Prop} := \forall\ P\ Q : \mathit{Prop},$$
$$\sim(P\ /\backslash\ Q) \rightarrow \sim P\ \backslash/ \sim Q\ .$$

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

3.3 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:

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

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 $\Sigma_{\mathrm{irr}}\ (A : \mathit{Type})\ (B : A \rightarrow \mathit{Type}) : \mathit{Type}$ **where**
  constructor $\_,\_$
  field
    $\mathit{fst}\ \ : A$
    $.\mathit{snd} : B\ \mathit{fst}$

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 $.A \to A$.

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:

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

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:

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

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:

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

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

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

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:

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

3.3.1 Quicksort example

If we want to mark the $\mathit{qsAcc}\ \mathit{xs}$ 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 $\mathit{qsAcc}\ \mathit{xs}$ is indeed a decreasing argument and refuses the definition.

3.4 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):

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

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

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

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 $\mathit{ltS}\ 0\ 0\ p'$ 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 $D : I_0 \to \cdots \to I_n \to \mathit{Type}$ 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, so instead we limit ourselves to a subset that we can recognise, called *concretely* collapsible families. A family $D : I_0 \to \cdots \to I_n \to \mathit{Type}$ 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.

3.4.1 Erasing collapsible families

Elaborate on how concrete collapsibility can be used to optimise things: we can erase certain things because we can recover them from the indices.

3.4.2 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.

3.5 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: given an inductive family D indexed by the types $I_0, \cdots, I_n$, D is collapsible if for every sequence of indices $i_0, \cdots, i_n$ and terms x, y, the following holds:

$$\vdash x, y : D\ i_0\ \cdots\ i_n \text{ implies} \vdash x \stackrel{\Delta}{\equiv} y$$

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

$$\vdash x, y : D\ i_0\ \cdots\ i_n \text{ implies} \vdash p : x \equiv y$$

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 $x \stackrel{\Delta}{\equiv} y$, 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:

$$\vdash p : (i_0 : I_0) \to \cdots \to (i_n : I_n) \to (x\ y : D\ i_0\ \cdots\ i_n\ ) \to x \equiv y$$

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.

Is there a family that is internally but not concretely collapsible? e.g. can we prove that Compare is internally collapsible?

3.6 Internalising the collapsibility optimisation

In section 3.4.1 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 $f : (i_0 : I_0) \to \cdots \to (i_n : I_n) \to (x : D\ i_0\ \cdots\ i_n\ ) \to \tau$, we generally cannot produce a function $\widetilde{f} : (i_0 : IO) \to \cdots\ (i_n : I_n) \to \tau$, since we sometimes need the $x : D\ \ i_0\ \ \cdots\ \ i_n$ 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. Along with such a function, we should also give a proof that the generated function is equal to the original one.

Motivate why things are constant functions basically.

Can we recover the same optimisation as we did beforehand?

Can atleast be done for subset of internally collapsible families: the "contractible" ones. Making use of the irrelevance stuff of Agda, we can implement the optimisation internally and prove its correctness.

Argue why moving up from "contractible" to internally collapsible is hard if not impossible.

Timing issues: may be solved with counting monads?

Singleton elimination?

3.7 Indexed h-propositions and homotopy type theory

The definition of internal collapsibility looks a lot like an indexed version of h-propositions. In homotopy type theory, we no longer have an empty context at run-time: the context may contain non-canonical identity proofs, coming from the univalence axiom or higher inductive types. As such, we no longer have the canonicity property.

Make it a bit more clear what kind of contexts we are dealing with.

Introduce indexed h-propositions.

Show how things can go wrong with prop eq not implying def eq, with interval as HIT

Things can also go wrong if codomain is "normal" in the sense of a 0-HIT and 2-type.

If domain is "normal" then things *should?* pan out, sort of. However, lacking canonicity in the codomain we're still screwed.

If both domain and codomain are normal: still screwed, lacking canonicity.

How does this tie in with Voevodsky's canonicity conjecture?

3.8 (-1)-truncation and optimisations

Chapter 4

Applications of homotopy type theory

Remove this: a citation to keep stuff from complaining (Martin-Löf, 1985)

Bibliography

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.

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

Per 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.