Erasing propositions and homotopy type theory

Gabe Dijkstra

June 25, 2013

Contents

Chapter 1

Introduction

Chapter 2

Homotopy type theory

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

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

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.

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

with the following function definition[2]

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

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.

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:

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

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, 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 $Prop$.

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 $Prop$ ($/\backslash$) 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.

3.2.2 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.

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:

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

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

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:

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

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:

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

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 $.A \to A$.

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:

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

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:

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

3.3.1 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.

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):

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

with elimination operator

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

and computation rules

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

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

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

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

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.

3.4.1 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.

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[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 differ from collapsible families as can be seen by considering D to be the family Id. By canonicity we have that for any $A : \mathit{Type}$, $x, y : A$, a term p satisfying $\vdash\ p : \mathit{Id}\ A\ x\ y$ necessarily reduces to refl. This means that Id is a collapsible family. In contrast, Id does not satisfy the internalised condition given above, since this then boils down to the uniqueness of identity proofs principle, which does not hold, as we have discussed.

[4]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.

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 $f : (i : I) \to (x : D\ i) \to \tau$, 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):

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

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:

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

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

something about situation with concretely collapsible families

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

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

3.6.1 Time complexity issues

Using this definition we do get enough information to write $optimiseFunction$. However, the success of the optimistically named function $optimiseFunction$ relies on time complexity the proof given of $isInternallyCollapsibleDecidable\ D\ I$ 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. $Thunk : \mathbb{N} \to Type \to Type$ is implemented as an abstract type that comes with the following primitives:

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

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?

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.

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

3.7 Indexed h-propositions and homotopy type theory

In section **??** we have seen that h-propositions are exactly those types that obey proof irrelevance. We can generalise this internal notion to the indexed case. Previously we have called this 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 things.

3.7.1 Indexed h-propositions versus internally collapsibly families

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 canonical and unique and can be recovered from other parts of our program. In homotopy type theory we no longer have this canonicity and have a choice in what inhabitant we can 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:

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

with computation rules[6]:

[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.

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

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, $\top$:

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

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.

3.7.2 Externally optimising h-propositions

3.7.3 Internally optimising h-propositions

3.8 Conclusion and future work

Chapter 4

Applications of homotopy type theory

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.

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.