Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science
Bachelor's Program in Computer Science

Bachelor's Thesis

Representations of Boolean Functions in Constructive Type Theory

submitted by
Gilles Nies

submitted on
March 20, 2012

Supervisor
Prof. Dr. Gert Smolka

Advisor
Dr. Chad E. Brown

Reviewers
Prof. Dr. Gert Smolka
Dr. Chad E. Brown

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken,

Datum/Date	Unterschrift/Signature

Abstract

Boolean functions are of great importance in computer science, be it in logic or for verification purposes in cryptography and hardware verification. Our goal is to formalize an informative canonical representation of boolean functions in Coq's constructive type theory: Prime trees. For that purpose we present several plausible characterizations of boolean functions and suggest various definitions of decision trees, the underlying datatype of prime trees: A dependent inductive type, a recursive type and a simple inductive type. In order to show that prime trees are canonical we explicitely define mappings between prime trees and boolean functions in a constructive way and prove that these mappings form an isomorphism, thereby showcasing the elusiveness of the concept of isomorphism in an intentional type theory. Some of these isomorphisms will require the assumption of axioms ranging from conventional assumptions to more controversial ones.

Acknowledgements

First of all, I would like to express my deep gratitude to my advisor Dr. Chad E. Brown for his masterful way of guiding me through the whole process of writing my thesis. His advice, more than once, gave me flashes of insight, that were invaluable to the completion of this document.

Furthermore, I would like to thank Prof. Dr. Gert Smolka for his lecture 'Introduction to Computational Logic' that lead to the opportunity of finishing my Bachelor studies under his chair's guidance.

Lastly, I want to express my appreciation for all the people that have supported me to whatever extend during the last months.

Contents

Chapter 1

Introduction

1.1 Motivation and Overview

Boolean functions are an important topic in hardware design for computers, especially for hardware verification purposes. In order to show that pieces of hardware, more precisely the circuits embedded in that hardware, function correctly one has to show that they implement their specification. Popular means of formalizing such specifications are boolean functions. Circuits that compute the specified boolean functions are considered correct. One of the major advances in not only verification, but in computer science and mathematics in general, are proof assistants or proof checkers. The catchphrase of "machine-checked proofs" comes to mind. It is, therefore, crucial to come up with suitable representations of boolean functions in such proof assistants.
In this thesis we give several ways of representing boolean functions in Coq's constructive type theory [2, 4]: n-ary cascaded functions which is a recursive function type, as well as a the continuous members of $(nat \to bool) \to bool$ or alternatively $Stream\ bool \to bool$. Since Coq's type theory, the Calculus of Inductive Constructions, is intentional, we often-times fail to prove equality of Coq's built-in functions and therefore we make it our goal to formalize an informative canonical representation for boolean functions in Coq.
This will happen by way of decision trees, a tree-like structure, to which we add constraints to finally arrive at prime trees [7], that share most of the convenient properties of Bryant's Binary Decision Diagrams (BDDs) [5]. We realize decision trees as a dependent inductive type $DT : nat \to Type$, a recursive type $DT_{rec} : nat \to Type$ equivalent to DT and as a simple inductive type $SDT : Type$. Thereafter, we declare prime trees to be only the reduced and ordered decision trees.
We give ways of mapping boolean functions to prime trees and back and show that prime trees are indeed unique for a boolean function by proving that these mappings form an isomorphism between boolean functions and prime trees. To formalize the notion of isomorphism in Coq we seize the idea of using partial and total setoids [3], which typically are utilized to represent sets in type theory. They consist of a carrier type, a relation on that type serving as an equality notion, as well as a proof that the equality is a (partial) equivalence relation.
For some of the isomorphisms presented in this thesis, the assumption of axioms is necessary to avoid pitfalls. To be more accurate, we will need to circumvent the elim restriction.
The axioms we will assume range from conventional assumptions like Functional Extensionality (FE) and Proof-irrelevance (PI) to more controversial ones like the Axiom of Continuity. We will, however, also be able to present isomorphisms between boolean functions and prime trees, that do not require any unprovable assumptions, but have their own set of draw-backs.

1.2 Structure of the thesis

The thesis consists of three main parts. In Chapter 2 we introduce the mathematical notions of boolean functions, decision trees and prime trees, as well as notation for those semantic objects.

Chapter 3 gives a short introduction to Coq's Calculus of Inductive Constructions. In addition we introduce some types which form the very base of our endeavour, like *bool* and a model that expresses an isomorphism between two types, as well as a classical axiom which is always of importance when talking about functions in Coq: Functional Extensionality.

The fourth chapter introduces one possible representation for boolean functions in constructive type theory; cascaded boolean functions. Furthermore, we give a suitable inductive type for decision trees as a so-called dependent inductive type. With this, we then derive a straightforward definition of prime trees and prove that our representation of prime trees is isomorphic to cascaded boolean functions. In addition, we briefly demonstrate how to define decision trees in a recursive way and show that both representations of decision trees coincide.

Chapter 5 outlines the construction of decision trees as a simple inductive type. Moreover, we give three alternative definitions for boolean functions and show which problems arise while proving that boolean functions and prime trees are isomorphic in a constructive setting. In order to work around these problems axioms are needed.

Chapter 2

Theory

In this Chapter we give a short formal introduction to boolean functions, decision trees and prime trees as given by Smolka and Brown in [7]. Prime trees are essentially a simplified version of Bryant's Binary Decision Diagrams [5].

2.1 Boolean functions

Boolean functions are an abstraction of functional circuits, which map a certain number of two-valued inputs $x_1, \ldots, x_n$ to a certain number of outputs $y_1, \ldots, y_m$. Each one of those outputs is computed using as many available inputs as needed. This means that every output combined with the inputs, characterizes a certain boolean function. Mathematically speaking, we have a finite set of variables V and *assignments* which are functions $V \to \mathbb{B}$ mapping variables to truth-values $\{1, 0\} =: \mathbb{B}$. Boolean functions compute a truth-value from the assigned values of the variables. This pinpoints the set of boolean functions to $(V \to \mathbb{B}) \to \mathbb{B}$.

There are several possible ways to represent a boolean function, the first one being truth tables. Truth tables list the result of a boolean function for every possible assignment. For $V := \{a, b\}$ an example of a truth table for a boolean function ϕ is given by:

$\sigma\, a$	$\sigma\, b$	$\phi\, \sigma$
0	0	0
0	1	0
1	0	0
1	1	1

One can easily observe that $\phi\ \sigma = 1$ if $\sigma\ a = 1$ and $\sigma\ b = 1$. Otherwise ϕ evaluates to 0. While truth tables give a very detailed characterization of boolean functions, they are not suited well for storing a boolean function in memory due to their high need of space.
The well-known *logical formulae* give a much more compact representation of a boolean function. It is no surprise that the above ϕ is described by the formula $a \wedge b$. We denote with $\mathbb{F}$ the set of logical formulas depending only variables from V. For $s \in \mathbb{F}$ we define $\mathcal{V}\ s$ to be the set of variables used in s. We also specify a function $\langle\cdot\rangle \in \mathbb{F} \to (V \to \mathbb{B}) \to \mathbb{B}$ mapping formulas to their boolean

functions:

$$\begin{aligned}
\langle \top \rangle &= \lambda\sigma.\, 1 \\
\langle \bot \rangle &= \lambda\sigma.\, 0 \\
\langle a \rangle &= \lambda\sigma.\, \sigma a \\
\langle \neg s \rangle &= \lambda\sigma.\, 1 - \langle s \rangle\sigma \\
\langle s \wedge t \rangle &= \lambda\sigma.\, min\{\langle s \rangle\sigma, \langle t \rangle\sigma\} \\
\langle s \vee t \rangle &= \lambda\sigma.\, max\{\langle s \rangle\sigma, \langle t \rangle\sigma\} \\
\langle s \to t \rangle &= \lambda\sigma.\, max\{1 - \langle s \rangle\sigma, \langle t \rangle\sigma\} \\
\langle s \equiv t \rangle &= \lambda\sigma.\, if\ \langle s \rangle\sigma = \langle t \rangle\sigma\ then\ 1\ else\ 0
\end{aligned}$$

While formulas give compact characterizations of boolean functions, there is not just one formula per boolean function. In fact there are infinitely many formulas representing the same function. For instance, $\top$ and $x \vee \neg x$ both represent the same boolean function, as $\top \equiv (x \vee \neg x)$ holds.

2.2 Decision Trees

Considering that formulas are not unique for a boolean function, we will treat a special class of formulas called *decision trees*, which will form the foundation of an informative canonical representation for boolean functions. We will start by introducing *conditionals*, which are well-known from programming. Conditionals are formulas of the form $t \wedge s_1 \vee \neg t \wedge s_2$ which we will write as (t, s_1, s_2). Using this construct, decision trees can elegantly be defined by induction.

- $\top$ and $\bot$ are decision trees
- (x, t_1, t_2) is a decision tree, if x is a variable and if t_1 as well as t_2 are decision trees.

As the name suggests, decision trees can be interpreted as tree diagrams. Using this diagram representation of a formula, it is easy to decide whether it is valid for a certain input. For $\sigma \in V \to \mathbb{B}$ and $s \in \mathbb{F}$, we can compute $\langle s \rangle\ \sigma$ by recursively following the path described by σ starting at the root x, where $\sigma\ x = 1$ means, that we can forget about the right subtree of x and continue by evaluating the left subtree. The result is then given by the leaf we inevitably reach, where $\top$ means 1 and $\bot$ means 0.

2.3 Prime Trees

Prime trees are decision trees which fulfill two properties. They are *reduced* and *ordered*. A decision tree is reduced if no internal node has equal subtrees, meaning that no subtree is of the form (x, t, t). Formally, we can capture this as a recursive predicate *reduced* on decision trees:

- $\bot$ and $\top$ are reduced
- (x, t_1, t_2) is reduced $\iff t_1 \neq t_2$ and t_1 as well as t_2 are reduced

Assume a linear order $<$ on V. For $x, y \in V$, we say that x is smaller than y if $x < y$. A decision tree is then ordered if, for every path from root to leaf, the variables become smaller. We also formalize this with a predicate on decision trees with an intermediate step. We first define when a decision tree is ordered with respect to a variable.
Let x, y be variables:

- $\bot$ and $\top$ are ordered w.r.t. x

- (y, t_1, t_2) is ordered w.r.t. $x \iff y < x$ and both t_1, t_2 are ordered w.r.t. y

With this intermediate step, we are able to define when a decision tree is ordered in an elegant way:

- $\bot$ and $\top$ are ordered
- (x, t_1, t_2) is ordered $\iff t_1, t_2$ are ordered w.r.t. x

In fact, these constraints remind of Binary Decision Diagrams introduced by Bryant in 1986 [5].
For illustration purposes, let $V = \{1, 2\}$. Then $(1, \top, \top)$ and $(1, (2, \bot, \top), \top)$ are not prime trees, the former not being reduced and the latter not being ordered since $2 \not< 1$. The decision tree $(2, (1, \bot, \top), \top)$, however, is reduced and ordered, which also makes it a prime tree.

Chapter 3

Preliminaries

In this chapter we introduce Coq's Calculus of Inductive Constructions as well as some well-known issues that come with it (Section 3.2). We discuss issues that arise while dealing with functions in Constructive Type Theory in Section 3.3 and give a few Coq definitions which form the very basis of the thesis' goal in Sections 3.1 and 3.3. The main focus lies on Section 3.4, where we try to bring across the complications one has to deal with, when trying to find a reasonable way of representing isomorphisms in Coq. In the end, we come up with a model of isomorphism based on the notions of setoid and partial setoid as given by Barthe et al. in [3].

3.1 A two-valued type: bool

The intuition behind boolean functions as an abstract object is more or less clear. However, before one can begin to define such functions in Coq, one needs to come up with a type to represent the two-valued inputs and outputs of boolean functions. Such a type is given by *bool*. It is predefined in Coq as an inductive type, hence, the keyword *Inductive* [2].

```
Inductive bool : Set :=
  | true : bool
  | false : bool.
```

Three constructors are involved in this definition. *bool* is a type constructor. It is of type *Set*, which is a universe of types. *true* and $false$ are the member constructors of the type *bool*. In general, the only members of an inductive type in Coq, are those terms constructed by its member constructors. This means that *bool* has exactly two members: *true* and $false$. *bool* will serve us as representative for our two-valued inputs and outputs.

3.2 Coq and the Elim Restriction

The central idea of Constructive Type Theory is to model propositions as types. This means that *programs* and *properties* are both written in the same language. For Coq this language is the *Calculus of Inductive Constructions*. The heart of Coq is a *type checking* algorithm. What this means for functional programming is clear: We design structures of a certain type T and we do computation on them by defining procedures which take elements of type T as arguments. The type checker then makes sure that the argument and the procedure are compatible before doing any computation. In Coq, such datastructures, say D, would be inhabitants of the *type universe Type* and procedures on them would have *functions types* from D to some other type T. Function types are of the form $\forall(t_1 : T_1) \cdots (t_n : T_n),\ T\ t_1 \cdots t_n$ and are themselves again members of *Type*. What is denoted by *Type*, is, however, not just one universe, but infinitely many, namely: $Type_n$ for any natural

number n.[1] We say that function types quantify over T, if their members take arguments of type T. Function types which quantify over $Type_n$, are in general inhabitants of a higher universe than their arguments: $Type_{n+1}$. For propositions and proofs the whole story is essentially the same with one big exception. Propositions are terms of type $Prop$. $Prop$ is the lowest type universe and it is *impredicative*, meaning that quantification over a type (including $Prop$) yields another proposition. Finding a proof of a proposition, say P, boils down to finding a term of type P. This means *proof checking* is in fact type checking. The impredicativity of $Prop$, however, comes at a price. We are not allowed to use information stored in a proof, to build something different then a proof. We refer to this restriction as the *elim restriction*. The issue is best made clear by a small example: Suppose we would define an inductive proposition $bool'$:

```
Inductive bool' : Prop :=
  | true' : bool'
  | false' : bool'.
```

The definition of the following function is allowed.

```
Definition bool2bool' (b : bool) : bool' :=
  if b then true' else false'.
```

$bool2bool'$ maps members of $bool$ to proofs of $bool'$ in an obvious way. The definition of the inverse of this function is however not permitted:

```
Definition bool'2bool (b : bool') : bool :=
  if b then true else false.
```

$bool'2bool$ maps proofs of $bool'$ to inhabitants of $bool$ which is *not* a proposition and is restricted because we need to know the structure of the proof in order to make a decision on which boolean to return.
A switching function on $bool'$ is, however, allowed, since we construct proofs from proofs.

```
Definition switch (b : bool') : bool' :=
  if b then false' else true'.
```

The elim restriction is necessary to ensure consistency of Coq.

3.3 Functional Extensionality

Obviously, functions are an important topic of this thesis. However, set theoretic functions and type theoretic functions are two very distinct notions. It is therefore very important to understand the difference, which is best shown by a small example. Consider the functions $bool \to bool$. Set theory tells us that there are exactly 4 different functions $bool \to bool$:

$$\begin{aligned} K_\top &:= \lambda\_.\, true \\ K_\bot &:= \lambda\_.\, false \\ negb &:= \lambda b.\, \neg b \\ id_{bool} &:= \lambda b.\, b \end{aligned}$$

Every other function $bool \to bool$ is considered equal to one of the four functions above. For instance $id_{bool} \circ id_{bool} = negb \circ negb = id_{bool}$ or $negb \circ K_\top = K_\bot$. This is because set theoretical functions abstract from algorithmic content. In type theory, however, functions are procedures with algorithmic content, which means that we have to give an implementation in order to define a function. We conclude, that $negb \circ negb$ and $id_{bool} \circ id_{bool}$ are different algorithms and therefore $negb \circ negb = id_{bool} \circ id_{bool}$ is not provable in Coq, implying that there are infinitely many functions

[1] The type universe *Set* mentioned in Section 3.1 can be seen as the second lowest type universe. Most of the basic types, like *bool*, are inhabitants of *Set*.

of type $bool \to bool$ in Coq. However, both procedures $negb \circ negb$ and $id_{bool} \circ id_{bool}$ compute the same mathematical function id_{bool}. We say that two procedures f and g, which compute the same function, are *equivalent*. In Coq, this idea is captured by the following definition:

`Definition` *equivalent* {A B : `Type`} (f g : $A \to B$) : `Prop` := $\forall$ a:A, f a = g a.

We consider procedures as equivalent (and write $f \equiv g$), if they have the same input-output behaviour. In the remainder of this thesis, we will call procedures functions, whenever it is clear that we are talking about procedures in Coq.
Sometimes one wants to be able to replace equivalent functions with each other in every possible context. This is not immediately possible in Coq, since it would involve proving something of the form $f \equiv g \Rightarrow f = g$ for every two functions of matching types. While it is easy to prove this set theoretically, since equivalent functions are also equal, it is not provable in Coq. The result is known as *Functional Extensionality* and it can only be assumed in Coq as an *axiom*.

`Axiom` $FE : \forall$ {A B : `Type`} (f g : $A \to B$), $f \equiv g \to f$=g.

Functional Extensionality is a conventional mathematical reasoning principle.

3.4 Representing an isomorphism in Coq

The main goal of this thesis is to show that prime trees give a canonical representation for boolean functions. We want to achieve this by showing that prime trees and boolean functions are isomorphic. Now the question arises on how to best represent an isomorphism in constructive type theory. In mathematics an isomorphism between two sets A and B is defined as an invertible function $f \in A \to B$.

$$f\ isomorphism_{A,B} \iff \exists g \in B \to A : f \circ g = id_B \wedge g \circ f = id_A$$

One can generalize this definition up to two relations $\equiv_A \subseteq A \times A$ and $\equiv_B \subseteq B \times B$, as follows:

$$f\ isomorphism_{A,B}^{\equiv_A,\equiv_B} \iff \exists g \in B \to A : \forall b \in B, (f \circ g)(b) \equiv_B b \quad \wedge \\ \forall a \in A, (g \circ f)(a) \equiv_A a$$

If one wants to show that two sets A and B are isomorphic, it suffices to find two functions $f \in A \to B$ and $g \in B \to A$, where g plays the role of f^{-1}, such that f is an $isomorphism_{A,B}^{\equiv_A,\equiv_B}$. We denote this using $(A, \equiv_A) \cong_f (B, \equiv_B)$.
Unfortunately, this definition does not characterize what we want. Consider the following relations on the natural numbers:

$$\equiv_{eq} = \{(x, y) \mid x = y\}$$
$$\equiv_{all} = \mathbb{N} \times \mathbb{N}$$

While $\equiv_{eq}$ tells us that two natural numbers are only related if they are equal, $\equiv_{all}$ relates every pair of natural numbers. Surprisingly, we can now prove the following, which we obviously don't want to have:

Lemma 3.1. $(\mathbb{N}, \equiv_{eq}) \cong_{id} (\mathbb{N}, \equiv_{all})$.

Proof. We have $(id \circ id)(x) = x$ and therefore $(id \circ id)(x) \equiv_{eq} x$. Also $(id \circ id)(x) \equiv_{all} x$ holds because both $(id \circ id)(x)$ and x are natural numbers. ∎

We are able to prove this, because we have not ensured that our mappings f and g respect their

respective relations. Formally, we have to enforce the additional property

$$\forall a_1, a_2 \in A:\ a_1 \equiv_A a_2 \Rightarrow g(a_1) \equiv_B g(a_2) \quad \wedge$$
$$\forall b_1, b_2 \in B:\ b_1 \equiv_B b_2 \Rightarrow f(b_1) \equiv_A f(b_2)$$

Clearly, this is violated for the above lemma, because $x \equiv_{all} y \not\Rightarrow id(x) \equiv_{eq} id(y)$, as x and y are in general not equal.

To capture this mathematical definition of an isomorphism in Coq, we first have to model relations. A relation between two types A and B will be a predicate R of type $A \to B \to Prop$. This way, $a \equiv b$ holds, if and only if $R\ a\ b$ is provable in Coq. We have already seen such a relation in the section about Functional Extensionality (Section 3.3), namely, equivalence of functions ($\equiv$) of type $(A \to B) \to (A \to B) \to Prop$ for some types A, B.

3.4.1 Setoids and partial setoids

To represent the ideas of *isomorphisms up to two relations* and *functions respecting relations* in Coq, we decided to revert to an elegant approach involving *setoids*, *partial setoids* and *morphisms*. Usually setoids are used to represent sets in intentional type theory by giving the "set" as a type called the carrier, an equality on the set and a proof component involving the equality [3]. A (total) setoid is defined as a dependent triple:

$$Setoid\ :=\ (T : Type\ ,\ R \subseteq T \times T\ ,\ R \text{ equivalence relation on T})$$

In Coq, this can be realized with Σ types.

```
Definition Reflexive {T : Type} (R : T → T → Prop) :=
  ∀ t : T, R t t.

Definition Symmetric {T : Type} (R : T → T → Prop) :=
  ∀ t1 t2 : T, R t1 t2 → R t2 t1.

Definition Transitive {T : Type} (R : T → T → Prop) :=
  ∀ t1 t2 t3 : T, R t1 t2 → R t2 t3 → R t1 t3.

Definition ER {T : Type} (R : T → T → Prop) :=
  Reflexive R ∧ Symmetric R ∧ Transitive R.

Definition Setoid := { T : Type & { R : T → T → Prop | ER R } }.
```

We use multiple versions of so-called Σ types in this definition.

- $\{a : A \mid P\ a\}$: This is notation for $sig\ A\ P$, where $A : Type$ and $P : A \to Prop$. It is a Curry-Howard version of existential quantification. While $\exists a : A,\ P\ a$ is a proposition, $\{a : A \mid P\ a\}$ has type $Type$. For both constructs, we have to give a witness $a : A$ for which the proposition $P\ a$ can be proven.

- $\{a : A\ \&\ T\ a\}$: Such a term is mostly referred to as a *dependent sum*. Without notation it takes the form $sigT\ A\ T$, where $A : Type$ and $T : A \to Type$. The difference to sig is that, the right-hand side $T\ a$ is a type instead of a predicate, which depends on the left hand side.

We will use overloaded notation for both sig and $sigT$ by writing $\Sigma_{a:A}(P\ a)$ whether P has type $A \to Prop$ or $A \to Type$.

We write functions for extracting the domain type and the relation over the carrier type.

```
Definition 𝒟 (S : Setoid) : Type := let (T,_) := S in T.
```

```
Definition 𝓡 (S : Setoid) : (𝓓 s → 𝓓 s → Prop) :=
  let (T, P) as z return (𝓓 z → 𝓓 z → Prop) := S in
  let (r, _) := P in r.
```

A function mapping from one setoid to another by preserving their individual equality is called a *morphism* and embodies the idea of a function respecting relations mentioned above. The definition is again by Σ type:

```
Definition morphism (S1 S2 : Setoid) : Type :=
  { f : 𝓓 S1 → 𝓓 S2 | ∀ x y : 𝓓 S1, (𝓡 S1) x y → (𝓡 S2) (f x) (f y) }.
```

Instead of *morphism* S_1 S_2 we will write $S_1 \twoheadrightarrow S_2$.
Two functions $\mathcal{D}\ S_1 \to \mathcal{D}\ S_2$ over setoid domains are equivalent if they their outputs are equal under $\mathcal{R}\ S_2$.

```
Definition morphism_eq {S1 S2 : Setoid} (f g : 𝓓 S1 → 𝓓 S2) : Prop :=
  ∀ x, (𝓡 S2) (f x) (g x).
```

We will write $f \equiv_{\mathcal{R}} g$ for *morphism_eq* f g.
Instead of saying that two types are isomorphic we now consider the idea of two setoids being isomorphic, by finding suitable relations and morphisms between them such that the composition of the morphisms is equivalent to the identity function. Formalized in Coq, this looks as follows:

```
Definition SetoidIsomorphism (S1 S2 : Setoid) (m1 : S1 ↠ S2) (m2 : S2 ↠ S1) :=
  let (f,_) := m1 in let (g,_) := m2 in (f ∘ g) ≡𝓡 id ∧ (g ∘ f) ≡𝓡 id.
```

By characterizing an isomorphism predicate like this, we make sure that the relations and mappings involved, fulfill some necessary conditions before we can even argue about an isomorphism at all.

Sometimes we have a suitable equality on a type but we fail to turn them into a setoid, because the equality is not provably reflexive. For us, this will be the case due to extensionality issues. We account for such cases by weakening the constraint on the equalities, such that they only have to be *partial* equivalence relations.

```
Definition PER {T : Type} (R : T → T → Prop) :=
  Symmetric R ∧ Transitive R.
```

We adapt the definition of setoid to define *partial setoids*.

```
Definition PartialSetoid := { T : Type & { R : T → T → Prop | PER R } }.
```

The definitions of $\mathcal{D}$, $\mathcal{R}$ and *morphism*, also work on partial setoids, by changing the argument type to *PartialSetoid*. Therefore we use $\mathcal{D}$, $\mathcal{R}$ and *morphism* for both flavours of setoids.

The definition of *morphism_eq* has to be adapted to account for the missing reflexivity of the relations.

```
Definition morphism_eq_p {S1 S2 : PartialSetoid} (f g : 𝓓 S1 → 𝓓 S2) : Prop :=
  ∀ x, (𝓡 S1) x x → (𝓡 S2) (f x) (g x).
```

We will write $f \equiv^{p}_{\mathcal{R}} g$ for *morphism_eq_p* f g. A definition for an isomorphism between two partial setoids can easily be derived from the above adaptation.

```
Definition PartialSetoidIsomorphism (S1 S2 : PartialSetoid) (m1 : S1 ↠ S2) (m2 : S2 ↠ S1) :=
  let (f,_) := m1 in let (g,_) := m2 in (f ∘ g) ≡^p_𝓡 id ∧ (g ∘ f) ≡^p_𝓡 id.
```

We will see a legitimate example of two partial setoids which are isomorphic in Section 5.6.

Chapter 4

Decision trees as a dependent inductive type

In this chapter we provide a first formalization of boolean functions, decision trees and prime trees in Coq. Section 4.1 introduces boolean functions as n-ary cascaded functions as defined in Coq's standard library [2]. In Section 4.2 we realize decision trees as a dependent inductive type and showcase the complications and lack of automation that comes with it. Section 4.2 also covers a small debate about recursive types and inductive types, inspired by Adams in [1] and gives an alternative recursive definition of decision trees and a proof that both representations are in fact equivalent. In the last section (Section 4.3), we add constraints to decision trees to obtain prime trees, which are a simplified version of Bryant's Binary Decision Diagrams (BDD) [5] , and show that prime trees are canonical representatives for boolean functions.

4.1 Cascaded boolean functions

In Chapter 3 we already saw how to model the two-valued inputs and outputs of boolean functions. It is however not immediately clear how a suitable type for boolean functions themselves looks like. In Chapter 2, we defined boolean functions using a finite set of variables V and assignments of the form $V \to \mathbb{B}$. To translate this into Coq, we have to define a function type taking arbitrarily, but finitely many two-valued inputs. More precisely, a type taking n booleans before returning a boolean: $bool^n \to bool$. In fact, such function types $A^n \to B$ for arbitrary types A and B are predefined in Coq's standard library by recursion on the number of inputs.

```
Fixpoint nfun A n B :=
  match n with
    | 0 ⇒ B
    | S n ⇒ A → (nfun A n B)
  end.
```

We call functions of type $A^n \to B$ *cascaded functions* from A to B. Note that $A^0 \to B$ reduces to B, which matches our intuition of a 'function taking no argument'. Obviously, a boolean function taking n arguments will be modelled in Coq as *nfun bool n bool*, which we will abbreviate as $bool^n \to bool$.

In order to prove any meaningful result, we need to be able to give all the arguments to a cascaded function $A^n \to B$ at once. Since we cannot quantify over n variables, we need to come up with a type, whose inhabitants contain exactly n elements of type A. The Coq standard library has such a type predefined. It is called *nprod* and like the name suggests, it realizes an n-ary product or n-tuple. It is also defined by recursion on the size of the tuple.

```
Fixpoint nprod A n : Type :=
  match n with
    | 0 ⇒ unit
    | S n ⇒ A × nprod A n
  end.
```

This definition is worthy of more explanation. It involves the inductive types *unit* and *prod*. *unit* is the inductive type with exactly one inhabitant, which is obtained by its sole constructor $tt : unit$. The type *prod* realizes pairs in Coq. It, too, has only one constructor *pair* taking two elements $a : A$ and $b : B$ for two arbitrary types A, B. We conclude that *nprod A n* stands for $A \times ... \times A \times unit$, with n occurrences of A in this type. We will use the conventional mathematical notation A^n for n-tuples over A and we will denote members of A^n as $\vec{a}$. In order to avoid ambiguities we will write $A^n \to B$ for *nfun A n B* and $(A^n) \to B$ for $nprod\ A\ n \to B$.

Of course, cascaded functions cannot immediately handle n-tuples as arguments. Coq's standard library provides a function called *nuncurry* for exactly that purpose. We name this procedure Ap and it will allow us to feed the n arguments to the function one by one.

```
Fixpoint Ap {A B : Type} {n : nat} : (A^n → B) → (A^n) → B :=
  match n with
    | 0 ⇒ fun b _ ⇒ b
    | S n ⇒ fun f a⃗ ⇒ let (a,a⃗') := a⃗ in Ap (f a) a⃗'
  end.
```

The first thing that stands out is the return type $\forall (f : A^n \to B)(\vec{a} : A^n), B$ of Ap. The structure of f and $\vec{a}$ depends on the implicit argument n, which is why we bind them after we did the case analysis on n, when their top level structure is certain. If we have no arguments, $A^0 \to B$ reduces to B, making f an element of type B, which we suggestively bind as b, then we return exactly b. In the other case, we use the fact that $\vec{a}$ has type A^{n+1}, meaning that we can extract the first element a, apply it to f, which is of type $A^{n+1} \to B$, and continue recursively by applying $f\ a : A^n \to B$ to $\vec{a}' : A^n$. For readability we will leave Ap implicit most of the time. We will, thus, write $f\ \vec{a}$ for $Ap\ f\ \vec{a}$.

We have seen in Section 3.3, that we consider two functions as equivalent, when they have the same input-output behaviour. With Ap we are able to capture this idea for cascaded functions.

```
Definition Feq {A B : Type} {n : nat} (f g : A^n → B) : Prop :=
  ∀ a⃗ : A^n, Ap f a⃗ = Ap g a⃗.
```

We write $f \equiv g$, when f is equivalent to g.

Lemma 4.1. *$\equiv$ is an equivalence relation.*

Proof. Let $f, g, h : A^n \to B$ be given.

- Reflexivity: For any given $\vec{a} : A^n$, $f\ \vec{a} = f\ \vec{a}$ holds by reflexivity of =.
- Symmetry: Let $f \equiv g$ and $\vec{a} : A^n$ be given. Because of $f \equiv g$ we also have $f\ \vec{a} = g\ \vec{a}$ which gives us the desired result by symmetry of =.
- Transitivity: Let $f \equiv g$ and $g \equiv h$ be given. For any given $\vec{a} : A^n$, we have $f\ \vec{a} = g\ \vec{a}$ and $g\ \vec{a} = h\ \vec{a}$. The claim follows through transitivity of =.

■

One can show that for $n = 0$, equality can be proven from equivalence.

Lemma 4.2. $\forall fg : A^0 \to B,\ f \equiv g \to f = g$.

Proof. Let f and g be 'functions' taking no argument. We have $A^0 = unit$, $f\ \vec{a} = f$ and $g\ \vec{a} = g$ for $\vec{a} : A^0$ by definition. Then,

$$\begin{aligned} f \equiv g &= \forall \vec{a} : A^0, f\ \vec{a} = g\ \vec{a} \\ &= unit \to f = g \end{aligned}$$

So, we obtain a proof of $f = g$ by feeding $tt : unit$ to the proof of $f \equiv g$. ∎

As for equivalence, we have to adapt the statement of Functional Extensionality to cascaded functions. In fact, we can prove that from Functional Extensonality one can obtain an extensionality principle for cascaded functions.

Lemma 4.3 (Cascaded Functional Extensionality). *Let A and B be two arbitrary types and $n : nat$. Then, $FE \to \forall fg : A^n \to B,\ f \equiv g \to f = g$.*

Proof. We assume FE. Let $f, g : A^n \to B$ with $f \equiv g$ be given. To show : $f = g$
Induction on n.

- $n = 0$: We know by Lemma 4.2 that equivalence and equality coincide for $n = 0$.
- $n \to n + 1$: We have $f, g : A^{n+1} \to B$ with $f \equiv g$ in addition to the inductive hypothesis

  $$\mathcal{IH} : \forall fg : A^n \to B,\ f \equiv g \to f = g$$

  According to FE, it suffices to show that $\forall a : A,\ f\ a = g\ a$ to obtain $f = g$.
  Let $a : A$ be given. Since $f\ a$ and $g\ a$ have type $A^n \to B$, $\mathcal{IH}$ applies and it suffices to show $f\ a \equiv g\ a$: $\forall \vec{a} : A^n,\ (f\ a)\ \vec{a} \triangleq f\ (a, \vec{a}) \overset{f \equiv g}{=} g\ (a, \vec{a}) \triangleq (g\ a)\ \vec{a}$.

∎

4.2 Decision trees

Syntax

Considering that we have defined decision trees inductively in Chapter 2, it is only natural to use an inductive type to represent decision trees in Coq. We will do so with a somewhat unconventional dependent inductive type DT of type $nat \to Type$. The idea is that every decision tree depends on a natural number, representing the number of variables the decision tree depends on. In the following we will refer to this natural number as the *level* of the decision tree.

```
Inductive DT : nat → Type :=
  | DT_0 : DT 0
  | DT_1 : DT 0
  | DT_I : ∀ {n}, DT n → DT n → DT (S n)
  | DT_L : ∀ {n}, DT n → DT (S n).
```

It may be surprising to the reader that DT has four constructors. As expected we have two constructors representing the base cases, DT_0 for $\bot$ and DT_1 for $\top$, which both are of level 0, because they do not depend on any variable. DT_I is the *branching constructor*. It takes two subtrees of equal level n and creates a decision tree of an elevated level $n + 1$. The fourth constructor DT_L is the *lifting constructor*, which increases the level of a decision tree from n to $n + 1$. This is necessary since a branching tree can only be constructed via two subtrees of the same level. Assume two trees $t_1 : DT\ n$ and $t_2 : DT\ m$ with $n < m$. If one wants to combine them using DT_I, then one can do so by iterating the lifting constructor $m - n$ times on t_1, equating their levels.

Semantics

With the syntax fixed, we are able to define semantics of decision trees. We will use the denotational approach by defining the meaning of decision trees by a function. Clearly, this has to happen recursively since DT is an inductive type.

```
Fixpoint DT_Den {n:nat} (t:DT n) : bool^n → bool :=
  match t with
    | DT_0 ⇒ false
    | DT_1 ⇒ true
    | DT_I n t_1 t_2 ⇒ fun b:bool ⇒ if b then (DT_Den t_1) else (DT_Den t_2)
    | DT_L n t ⇒ fun _ ⇒ (DT_Den t)
  end.
```

Unsurprisingly, we return $true$ and $false$ for DT_1 and DT_0 respectively. In the branching case DT_I, we do a case analysis on the first boolean argument to decide whether we have to consider the left or the right subtree. In the lifting case, we just take and ignore the first argument and continue recursively with the underlying tree.
As in the definition of Ap, the returned function depends on the parameter $n : nat$. In the first two cases DT_0 and DT_1, we have $n = 0$, because DT_0 and DT_1 are of type $DT\,0$, thus we must return an element of type $bool^0 \to bool$ which is equal to $bool$. In the last two cases, we have $n + 1$ arguments, therefore Coq expects us to return a function of type $bool^{n+1} \to bool = bool \to bool^n \to bool$. We do so by first taking an element of type $bool$ and recursively continuing with the subtrees of level n. In the following we will use $[\![t]\!]$ for $DT\_Den\ t$.
Note that the design of decision trees doesn't require any modelling of variables. We will therefore write $(\_, t_1, t_2)$ for $DT_I\ t_1\ t_2$, instead of (x, t_1, t_2) for any variable x, like we did in Chapter 2. Also we will use t^n for an n-times lifted tree $\underbrace{DT_L \ldots DT_L}_{n\ times}\ t$ in addition to $\bot$ and $\top$ for DT_0 and DT_1.

We will show that lifting a tree once, is equivalent to ignoring the first variable.

Lemma 4.4. *Let $t : DT\ n$, $b : bool$ and $\vec{b} : bool^n$ be given.*
Then, $[\![t]\!]\ \vec{b} = [\![t^1]\!]\ (b, \vec{b})$.

Proof. $[\![t^1]\!]\ (b, \vec{b}) = (fun\ \_ \Rightarrow [\![t]\!])\ (b, \vec{b}) = ((fun\ \_ \Rightarrow [\![t]\!])\ b)\ \vec{b} = [\![t]\!]\ \vec{b}$ ∎

Sometimes it is important to know if two decision trees are equal. In Coq, two terms are equal, if they are convertible, meaning, that they have the same normal form under all kinds of reductions. This implies that the terms must be of the same type. It turns out, that this prevents us from defining a decision procedure for equality on DT in a straightforward way. To be more precise, we want to define a certifying function of the form

$$\forall t_1\, t_2 : DT\ n,\ \{t_1 = t_2\} + \{t_1 \neq t_2\}.$$

which just us that we can either prove equality or inequality of two decision trees. However, the conventional approach using induction on t_1 and immediate case analysis on t_2 produces ill-typed cases like $\{\top = t^1\} + \{\top \neq t^1\}$. The problem is that, after induction on t_1, we end up having four cases: two cases where $t_2 : DT\ 0$ and two cases where $t_2 : DT\ (n+1)$. Subsequent case analysis on t_2 produces again 4 cases (per case), despite t_2 having type $DT\ 0$ (meaning that only $t_2 = \top$ or $t_2 = \top$ are possible) and $DT\ (n+1)$ (only $t_2 = (\_, t, t')$ or $t_2 = t^1$ are possible) respectively. Consequently, we have to come up with a dependent inversion function for DT, which deduces the appropriate cases from the level of a given decision tree. While the idea behind this is fairly easy, the translation into Coq is a quite tricky task.

```
Definition DT_Inv {n : nat} (dt : DT n) :
  match n as z return DT z → Type with
    | O ⇒ fun dt ⇒ {dt = ⊥} + {dt = ⊤}
    | S n' ⇒ fun dt ⇒ {p : (DT n') × (DT n') | dt = (_, fst p, snd p)}
                    + {t : DT n' | dt = t^1}
  end dt.
```

First of all, note, that we have not yet given a definition of DT_Inv. What we have written above is but the return type of DT_Inv.
We start by explaining the not previously introduced *sum* types. $sumbool : Prop \to Prop \to Set$ is a Curry-Howard version of the proposition $Or : Prop \to Prop \to Prop$. For both, an Or-proposition $P \vee Q$ and an instance of $sumbool\ \{P\} + \{Q\}$, it suffices to fulfill either P or Q. $sumbool$ is used in the $n = 0$ case of DT_Inv. $sum : Type \to Type \to Type$ is similar. Here, one has to give either a member of the left type or the right type. sum is used in the $n = S\ n'$ case of DT_Inv.
fst and snd are obviously the first and second projection on a non-dependent pair.
We achieve the inversion effect with a case analysis within the return type of DT_Inv, which states that the trees at level 0 are either $\bot$ or $\top$ and that trees of every other level are either lifted trees or branching trees. Note that the *return*-clause of the match states that we return a function taking a decision tree and returning a type. This is necessary, because, in order to make the return type of DT_Inv dependent, we have to bind the given decision tree dt anew in each individual case of the match. This is why we have to apply dt to the match at the very end.
Thankfully, it is not difficult at all to give a function which has this complicated type. Given the decision tree $dt : DT\ n$, we just have to consider the four different cases. Since dt depends on n, a case analysis on dt will also fix n to either 0 or $n' + 1$ for some $n' : nat$ and the match will reduce. Let dt be

- $\bot$: We have to give an element of $\{\bot = \bot\} + \{\bot = \top\}$. We trivially fulfill the left hand side.
- $\top$: We have to give an element of $\{\top = \bot\} + \{\top = \top\}$. We trivially fulfill the right hand side.
- $(\_, t_1, t_2)$: We fulfill the left hand side by giving the witness (t_1, t_2). With $fst\ (t_1, t_2) = t_1$ and $snd\ (t_1, t_2) = t_2$, we have $(\_, t_1, t_2) = (\_, fst\ (t_1, t_2), snd\ (t_1, t_2))$.
- t^1 : We fulfill the right hand side by giving t and prove the condition $t^1 = t^1$ trivially.

The necessity of such a complicated inversion function fuels a conflict well-known to Coq users: Which is better, inductive types or recursive types? The answer is best given by a small intuitive example given in [1]: Consider the following dependent inductive definition of *vectors of length n* over a type T:

```
Inductive V (T : Type) : nat → Type :=
  | nil : V T 0
  | cons n : T → V T n → V T (S n).
```

Now assume, that we want to prove the following obvious result about $\mathcal{V}\ T$:

$$\forall \vec{v} : \mathcal{V}\ T\ 0,\ \vec{v} = nil$$

In order to find a proof of this statement, we would have to write a similar inversion function to DT_Inv, which decides, by looking at the structure of the dependency, which case we are in. Now recall the definition of *nprod* from the previous section 4.1 , our *recursive* type for n-tuples. The corresponding result for *nprod* is a trivial:

$$\forall \vec{v} : T^0,\ \vec{v} = tt$$

The difference lies in the fact that we are easily able to deduce information about the structure of a recursive object by looking at the top-level structure of its dependency. This is due to the fact that recursive types are subject to reduction, which inductive types are not. In our example we know that T^0 reduces immediately to *unit*, which only holds *tt* as member. For $\mathcal{V}\ T\ 0$ this is not the case. We have to deduce by hand (by writing an inversion function) that a member of $\mathcal{V}\ T\ 0$ cannot be constructed with the constructor *cons* since it produces only non-empty vectors, hence it must be *nil*.

With this in mind we try to define a recursive type $DT_{rec} : nat \to Type$ isomorphic to DT. $DT\ 0$ has only two inhabitants, meaning that it is isomorphic to $bool =: DT_{rec}\ 0$. Alternatively, we could use $unit + unit : Set$, whose inhabitants are *inl unit tt* and *inr unit tt* . This idea is useful to define a type isomorphic to $DT\ (n+1)$ namely $(DT_{rec}\ n \times DT_{rec}\ n) + DT_{rec}\ n$. This way every $DT_{rec}\ (n+1)$ constructed with *inl* is considered to be a branching tree and all others are lifted trees. In Coq, recursive decision trees take the following form:

```
Fixpoint DT_rec (n : nat) : Type :=
  match n with
    | 0 ⇒ bool
    | S n ⇒ (DT_rec n × DT_rec n) + DT_rec n
  end.
```

What we hoped for, is indeed the case. By definition we have but two cases for a recursive decision tree of level 0, namely *true* or $false$, and two cases for a tree of level $n+1$, namely $inl\ (t_1, t_2)$ or $inr\ t$.

The denotation function $[\![\cdot]\!]_{rec} : \forall n,\ DT_{rec}\ n \to bool^n \to bool$ of DT_{rec} is similar to the one of DT:

```
Fixpoint DT_rec_Den {n : nat} : DT_rec n → bool^n → bool :=
  match n as z return DT_rec z → bool^z → bool with
    | 0 ⇒ fun t ⇒ if t then true else false
    | S n ⇒ fun t b ⇒
               match t with
                 | inl (t1, t2) ⇒ if b then DT_rec_Den t1 else DT_rec_Den t2
                 | inr t ⇒ DT_rec_Den t
               end
  end.
```

In fact one can easily prove that $DT\ n$ and $DT_{rec}\ n$ are isomorphic. Here are functions mapping back and forth. We will not use notation since the implicit argument $n : nat$ of DT_I and DT_L is important for the $DT2DT_{rec}$ mapping.

```
Fixpoint DT2DT_rec {n : nat} (t : DT n) : DT_rec n :=
  match t in DT z return DT_rec z with
    | DT_0 ⇒ false
    | DT_1 ⇒ true
    | DT_I n t1 t2 ⇒ inl (DT_rec n) ((DT2DT_rec t1), (DT2DT_rec t2))
    | DT_L n t' ⇒ inr (DT_rec n × DT_rec n) (DT2DT_rec t')
  end.

Fixpoint DT_rec2DT {n : nat} : (DT_rec n) → (DT n) :=
  match n as z return DT_rec z → DT z with
    | 0 ⇒ fun t : DT_rec 0 ⇒ if t then DT_1 else DT_0
    | S n ⇒ fun t : DT_rec (S n) ⇒ match t with
                                     | inl (t1,t2) ⇒ DT_I (DT_rec2DT t1) (DT_rec2DT t2)
                                     | inr t' ⇒ DT_L (DT_rec2DT t')
                                   end
  end.
```

We verify that the mappings fulfill the necessary conditions.

Lemma 4.5. *Let $n : nat$ be given.*

1. *$(DT\ n, =)$ and $(DT_{rec}\ n, =)$ are setoids.*
2. *$DT_{rec}2DT$ is a morphism $(DT_{rec}\ n, =) \twoheadrightarrow (DT\ n, =)$.*
3. *$DT2DT_{rec}$ is a morphism $(DT\ n, =) \twoheadrightarrow (DT_{rec}\ n, =)$.*

Proof.

1. = is an equivalence relation.
2. Let $t, t' : DT_{rec}\ n$ with $t = t'$ given. Then we immediately have $DT_{rec}2DT\ t = DT_{rec}2DT\ t'$.
3. Analogous to 2.

■

We prove that the two mappings indeed form an isomorphism between $(DT\ n, =)$ and $(DT_{rec}\ n, =)$.

Theorem 4.1. *Let $n : nat$ be given.*
We show: SetoidIsomorphism $(DT\ n, =)$ $(DT_{rec}\ n, =)$ $DT2DT_{rec}$ $DT_{rec}2DT$.

Proof. We have to show:

- $\forall t : DT\ n,\ t = DT_{rec}2DT\ (DT2DT_{rec}\ t)$: Let $t : DT\ n$ be given. Induction on t:
  - $t = \top$: By definition $DT2DT_{rec}\ \top = true$ and $DT_{rec}2DT\ true = \top$.
  - $t = \bot$: By definition $DT2DT_{rec}\ \bot = false$ and $DT_{rec}2DT\ false = \bot$.
  - $t = (\_, t_1, t_2)$: We have two inductive hypotheses:

    $$\mathcal{IH}_1 : t_1 = DT_{rec}2DT\ (DT2DT_{rec}\ t_1)$$
    $$\mathcal{IH}_2 : t_2 = DT_{rec}2DT\ (DT2DT_{rec}\ t_2)$$

    The claim follows immediately by the definitions of $DT_{rec}2DT$ and $DT2DT_{rec}$ as well as $\mathcal{IH}_1$ and $\mathcal{IH}_2$.
  - $t = t'^1$ We have the inductive hypothesis:

    $$\mathcal{IH} : t' = DT_{rec}2DT\ (DT2DT_{rec}\ t')$$

    The claim follows immediately by the definitions of $DT_{rec}2DT$ and $DT2DT_{rec}$ as well as $\mathcal{IH}$.
- $\forall t : DT_{rec}\ n,\ t = DT2DT_{rec}\ (DT_{rec}2DT\ t)$: We have $t : DT_{rec}\ n$. Induction on n:
  - $n = 0$: We either have $t = true$ or $t = false$. For both cases the claim follows since $DT_{rec}2DT\ true = \top$ and $DT2DT_{rec}\ \top = true$ as well as $DT_{rec}2DT\ false = \bot$ and $DT2DT_{rec}\ \bot = false$.
  - $n \to n + 1$: We have the inductive hypothesis:

    $$\mathcal{IH} : \forall t : DT_{rec}\ n,\ t = DT2DT_{rec}\ (DT_{rec}2DT\ t)$$

    Since $t : DT_{rec}\ (n + 1)$ we either have:

- $t = inl\ (t_1, t_2)$: We use $\mathcal{IH}$ for both t_1 and t_2 and obtain thereby $t_1 = DT2DT_{rec}\ (DT_{rec}2DT\ t_1)$ and $t_2 = DT2DT_{rec}\ (DT_{rec}2DT\ t_2)$. By this and the definitions of $DT2DT_{rec}$ and $DT_{rec}2DT$ we have the claim.
- $t = inr\ t'$: We use $\mathcal{IH}$ for t' and obtain $t' = DT2DT_{rec}\ (DT_{rec}2DT\ t')$. By this and the definitions of $DT2DT_{rec}$ and $DT_{rec}2DT$ we have the claim.

■

For the remainder of this chapter we will continue with the development for DT. All the results can be easily adjusted to DT_{rec}. If parts of the development are different for DT_{rec} we will write so explicitly. The main difference will be that we do not need an inversion function for DT_{rec} and that there is more automation support for the recursive version, which makes it more convenient to use.

We prove that the lifting and the branching constructor are injective. Coq provides a tactic *injection* for injectivity proofs. Consider the successor constructor $S : nat \to nat$ of the natural numbers. To show that S is injective, that is : $\forall n\, n' : nat,\ S\ n = S\ n' \to n = n'$, one applies *injection* to the proof of $S\ n = S\ n'$, which provides $n = n'$. Unfortunately, *injection* cannot prove injectivity claims for DT since it is a dependent inductive type, which is why we have to prove it by hand. We introduce a helping lemma:

Lemma 4.6 (f_equal). *Let A, B be arbitrary types and $f : A \to B$.*
Then, $\forall a_1 a_2 : A,\ a_1 = a_2 \to f\ a_1 = f\ a_2$.

Proof. trivial. ■

Note, that by instantiating f with the predecessor function *pred* on *nat*, one can show that S is injective using *f_equal*. *f_equal* is part of Coq's standard library.

Lemma 4.7 (Injectivity of $\mathbf{DT_I}$ and $\mathbf{DT_L}$). *We prove the following two claims* [1]*:*

1. *Let $dt_1, dt_2, dt_3, dt_4 : DT\ n$.*
   Then, $(\_, dt_1, dt_2) = (\_, dt_3, dt_4) \to dt_1 = dt_3 \wedge dt_2 = dt_4$.
2. *Let $dt_1, dt_2 : DT\ n$.*
   Then, $dt_1^1 = dt_2^1 \to dt_1 = dt_2$

Proof. Consider the functions $g, h : DT\ (S\ n) \to DT\ n$ defined as :

$$g\ t := \begin{cases} t_1, & t = (\_, t_1, t_2) \\ t', & t = t'^1 \end{cases} \qquad h\ t := \begin{cases} t_2, & t = (\_, t_1, t_2) \\ t', & t = t'^1 \end{cases}$$

1. By instantiating *f_equal* with $A := DT\ (S\ n)$, $B := DT\ n$, $f := g$ and the proof of $(\_, dt_1, dt_2) = (\_, dt_3, dt_4)$, we have $dt_1 = dt_3$, since $g\ (\_, dt_1, dt_2) = dt_1$ and $g\ (\_, dt_3, dt_4) = dt_3$.
   Analogously, we have $dt_2 = dt_4$, by instantiating $f := h$ instead of $f := g$.
2. By instantiating *f_equal* with $A := DT\ (S\ n)$, $B := DT\ n$, $f := g$ and the proof of $dt_1^1 = dt_2^1$ we have $dt_1 = dt_2$, since $g\ dt_1^1 = dt_1$ and $g\ dt_2^1 = dt_2$. Note, that we could also instantiate $f := h$. This is due to the fact that Coq only allows definitions of total functions. Since g and h take arguments of type $DT\ (S\ n)$ we also have to specify what happens with a lifted decision tree: Both functions return the underlying tree t'. ■

[1] The corresponding results for DT_{rec} can easily be proven using the *injection* tactic.

Now we are all set for the definition of a certifying equality deciding procedure on DT.

`Definition` *eq_dt_cert* $\{n : nat\}$ $(t_1\ t_2 : DT\ n) : \{t_1 = t_2\} + \{t_1 \neq t_2\}$.

We describe such a function by recursion on n.

- $n = 0$: We have $t_1, t_2 : DT\ 0$. With DT_Inv we end up with four cases, where either equality or inequality are trivially provable by constructor discrimination.

- $n = n' + 1$: We have $t_1, t_2 : DT\ (S\ n')$ and therefore t_1 and t_2 are either branching trees or lifted trees. Two of those cases can easily be treated with simple constructor discrimination to prove trivial inequalities. Two cases remain:

  - $t_1 = (\_, t_3, t_3')$ and $t_2 = (\_, t_4, t_4')$
    By recursion we can compare the left and right subtrees of t_1 and t_2. If $t_3 = t_4$ and $t_3' = t_4'$, then $(\_, t_3, t_3') = (\_, t_4, t_4')$ can easily be proven. If either $t_3 \neq t_4$ or $t_3' \neq t_4'$, then also $(\_, t_3, t_3') \neq (\_, t_4, t_4')$:
    Suppose $t_3 \neq t_4$ but $(\_, t_3, t_3') = (\_, t_4, t_4')$. We have $t_3 = t_4$ by Lemma 4.7. $\lightning$
    We use a symmetric argument for the $t_3' \neq t_4'$ case.

  - $t_1 = t_3^1$ and $t_2 = t_4^1$.
    By recursion we either have $t_3 = t_4$ and therefore also $t_3^1 = t_4^1$, or $t_3 \neq t_4$ which implies $t_3^1 \neq t_4^1$. For the last part, suppose $t_3 \neq t_4$ but $t_3^1 = t_4^1$. Through Lemma 4.7 we have $t_3 = t_4$. $\lightning$

For definition purposes we will define *bool* version of *eq_dt_cert*.

`Definition` *eq_dt* $\{n$:*nat*$\}$ $(t_1\ t_2 : DT\ n) : bool :=$ `if` *eq_dt_cert* $t_1\ t_2$ `then` *true* `else` *false*.

Whenever we refer to either version of decidable equality of decision trees we will write $t_1 =_{dt} t_2$.

4.2.1 Denotational Completeness for decision trees

In this section we will work on the first essential result of our development. It is a mapping $(bool^n \to bool) \to DT\ n$ which shows that every boolean function can be expressed as a decision tree. Such a result is called *denotational completeness*. To realize this in Coq, we use the concept of *certifying functions*, which means, that we define an algorithm carrying its correctness proof.

`Definition` *DT_DenCompl* $\{n : nat\} : \forall \phi : bool^n \to bool, \{t : DT\ n \mid [\![t]\!] \equiv \phi\}$.

We describe a function of this type via recursion on n:

- $n = 0$: ϕ takes no argument, implying that $\phi : bool$. Case analysis:

  - $\phi = true$: We choose $\top$, which clearly fulfills $[\![\top]\!] \equiv true$.

  - $\phi = false$: We choose $\bot$, which has the desired property $[\![\bot]\!] \equiv false$.

- $n = n' + 1$: We have $\phi : bool^{n'+1} \to bool$. By recursion, we obtain t_1 and t_0, the decision trees fulfilling $[\![t_1]\!] \equiv \phi\ true$ and $[\![t_0]\!] \equiv \phi\ false$, respectively. Our decision tree of choice is $(\_, t_1, t_0)$, and we show that $[\![(\_, t_1, t_0)]\!] \equiv \phi$.

Proof. Let $\vec{b} : bool^{n'+1}$ be given and let $\vec{b} = (b, \vec{b}')$:

$$\begin{aligned}
[\![(_, t_1, t_0)]\!]\ \vec{b} &= ([\![(_, t_1, t_0)]\!]\ b)\ \vec{b}' \\
&= (if\ b\ then\ [\![t_1]\!]\ else\ [\![t_0]\!])\ \vec{b}'
\end{aligned}$$

Case analysis on b:

$b = true$:

$$\begin{aligned}
[\![(_, t_1, t_0)]\!]\ \vec{b} &= [\![t_t]\!]\ \vec{b}' \\
&= (\phi\ true)\ \vec{b}' && ([\![t_1]\!] \equiv \phi\ true) \\
&= (\phi\ b)\ \vec{b}' \\
&\triangleq \phi\ \vec{b}
\end{aligned}$$

$b = false$:

$$\begin{aligned}
[\![(_, t_1, t_0)]\!]\ \vec{b} &= [\![t_0]\!]\ \vec{b}' \\
&= (\phi\ false)\ \vec{b}' && ([\![t_0]\!] \equiv \phi\ false) \\
&= (\phi\ b)\ \vec{b}' \\
&\triangleq \phi\ \vec{b}
\end{aligned}$$

■

By projection on the decision tree, we get a normal mapping from boolean functions to decision trees from the certifying version. One can define a general projection function π_1 returning the witness of a Σ type as follows:

```
Definition π1 {T : Type} {P : T → Prop} : { t : T | P t } → T :=
  fun σ ⇒ let (t,_) := σ in t.
```

We use π_1 to define a mapping from boolean functions to decision trees:

```
Definition BF2DT {n : nat} (φ : bool^n → bool) : DT := π1 (DT_DenCompl φ).
```

4.3 Prime trees

Given a decision tree it is not difficult to determine whether it is also a *prime tree*. According to Chapter 2, a prime tree is a reduced and ordered decision tree. The good news is, that we do not have to order our decision trees since the syntax and semantics of DT already enforce an order on variables. The reason is that we do not specify explicitly on which variable the branching constructor works and DT_Den is defined such, that it branches on the first remaining variable. Thus, we have no choice but to take the arguments from left to right, thereby enforcing orderedness of decision trees.

We can decide whether a decision tree is reduced in a recursive way as described in Chapter 2. For this purpose we use the boolean version of the fact that equality of decision trees is decidable.

```
Fixpoint reduced {n : nat} (t : DT n) : bool :=
  match t with
    | ⊥ ⇒ true
    | ⊤ ⇒ true
    | (_, t1, t2) ⇒ t1 ≠dt t2 ∧ reduced t1 ∧ reduced t2
    | t^1 ⇒ reduced t
  end.
```

We would like to define prime trees as the subset of decision trees which are reduced. In Coq, subsets are modelled with Σ types $\{a : A \mid P\ a\}$, where $P\ a$ is a proposition, meaning, it is of type *Prop*. *reduced dt*, however, has type *bool*. Luckily, we can automatically *coerce bool* into *Prop* every time we need a boolean b to be a proposition.

```
Coercion bool2Prop (b : bool) := if b then True else False.
```

Here, *True* is the trivial inductive proposition, which is directly provable by its constructor I : *True*. *False* is the empty proposition, only provable with inconsistent assumptions. By simple case analysis on b : *bool* one can show that $b = true$ iff *bool*2*Prop b* is provable. Since Coq puts coercions in automatically the definition of prime trees is now trivial:

```
Definition PT (n : nat) : Type := {dt : DT n | reduced dt}.
```

To reason about prime trees comes down to reasoning about the underlying decision tree. We can access this decision tree by projection.
By having *reduced* go to *bool*, we can show that all the proofs of *reduced dt* are equal, a result which is, for arbitrary propositions, only provable while assuming proof-irrelevance or equally strong assumptions. Therefore, it is always advisable to define predicates going to *bool* whenever possible.

Lemma 4.8. *Let dt be an arbitrary decision tree. Then,* $\forall\ r_1, r_2$ *: reduced dt ,* $r_1 = r_2$.

Proof. Case analysis. Let *reduced dt* be

- *true* : By coercion r_1 and r_2 are proofs of *True*, implying that $r_1 = I = r_2$.
- *false* : By coercion r_1 is a proof of *False*. $\lightning$

■

This fact leads to a very convenient property of prime trees considering that we are in a constructive environment: For two prime trees to be equal, it suffices that the underlying decision trees are equal.

Lemma 4.9. *Let* t_1 *and* t_2 *be two prime trees of level n. Then,* $\pi_1\ t_1 = \pi_1\ t_2 \rightarrow t_1 = t_2$.

Proof. We have prime trees t_1 and t_2 with $\pi_1\ t_1 = \pi_1\ t_2$. We know *reduced* $(\pi_1\ t_1)$ and *reduced* $(\pi_1\ t_2)$ by definition of prime trees. Equality of the proofs of *reduced* $(\pi_1\ t_1)$ and *reduced* $(\pi_1\ t_2)$ remains to be shown: With $\pi_1\ t_1 = \pi_1\ t_2$, we have two proofs of *reduced* $(\pi_1\ t_1)$. By Lemma 4.8 those two proofs must be equal. ■

For convenience and readability reasons we will use prime trees as if they were decision trees. Every time we use a prime tree *pt* in a context in which a decision tree is expected, we refer of course to $\pi_1\ pt$. Lemma 4.9 even allows us to do so for equality. In Coq this can be realized by another coercion:

```
Coercion PT_as_DT {n : nat} (pt : PT n) := π1 pt.
```

The recursive procedure for reducing decision trees is straightforward. The only interesting case to consider is the branching case, since only branching nodes may violate the reducedness condition. We will handle this case as follows:

$$reduce\ (\_, t_1, t_2) := \begin{cases} (reduce\ t_1)^1 & , reduce\ t_1 =_{dt} reduce\ t_2 \\ (\_, reduce\ t_1, reduce\ t_2) & , reduce\ t_1 \neq_{dt} reduce\ t_2 \end{cases}$$

Informally spoken, when we have two already reduced subtrees of a branching node, it is easy to merge them while preserving reducedness. We compare them using $=_{dt}$. Are the trees equal, then the variable of the branching node is irrelevant since, no matter how the variable gets assigned, we have the same subtree. Only when the reduced subtrees are different, the assignment of the variable leads to different behaviours.
In Coq, we will implement reducing of decision trees as a certifying function, which additionally carries proof that resulting prime tree has an equivalent denotation:

`Definition` *reduce_cert* $\{n : nat\}\ (dt : DT\ n) : \{\ pt : PT\ n \mid [\![dt]\!] \equiv [\![pt]\!]\ \}$.

We define *reduce_cert* by recursion on the decision tree dt:

- $dt = \bot$: Return $\bot$, which is a trivial prime tree and equivalent to itself.
- $dt = \top$: Return $\top$, which is a trivial prime tree and equivalent to itself.
- $dt = (\_, t_1, t_2)$: By recursion we have prime trees pt_1 and pt_2 with $[\![t_1]\!] \equiv [\![pt_1]\!]$ and $[\![t_2]\!] \equiv [\![pt_2]\!]$. Case analysis using $=_{dt}$:
  - $pt_1 = pt_2$: We return pt_1^1 and show $[\![pt_1^1]\!] \equiv [\![(\_, t_1, t_2)]\!]$.

    Proof. Let $\vec{b} : bool^{n+1}$ be given and $\vec{b} = (b, \vec{b}')$:

    $$[\![(\_, t_1, t_2)]\!]\ \vec{b} = (if\ b\ then\ [\![t_1]\!]\ else\ [\![t_2]\!])\ \vec{b}'$$

    Case analysis on b :

    $b = true$:

    $$\begin{aligned} [\![(\_, t_1, t_2)]\!]\ \vec{b} &= [\![t_1]\!]\ \vec{b}' \\ &= [\![pt_1]\!]\ \vec{b}' && ([\![pt_1]\!] \equiv [\![t_1]\!]) \\ &= [\![pt_1^1]\!]\ (b, \vec{b}') && (\text{Lemma 4.4}) \\ &= [\![pt_1^1]\!]\ \vec{b} \end{aligned}$$

    $b = false$:

    $$\begin{aligned} [\![(\_, t_1, t_2)]\!]\ \vec{b} &= [\![t_2]\!]\ \vec{b}' \\ &= [\![pt_2]\!]\ \vec{b}' && ([\![pt_2]\!] \equiv [\![t_2]\!]) \\ &= [\![pt_1]\!]\ \vec{b}' && (pt_1 = pt_2) \\ &= [\![pt_1^1]\!]\ (b, \vec{b}') && (\text{Lemma 4.4}) \\ &= [\![pt_1^1]\!]\ \vec{b} \end{aligned}$$

    pt_1^1 is reduced since pt_1 is reduced, which follows from pt_1 being a prime tree. ∎

  - $pt_1 \neq pt_2$: We return $(\_, pt_1, pt_2)$ and show $[\![(\_, pt_1, pt_2)]\!] \equiv [\![(\_, t_1, t_2)]\!]$.

Proof. Let $\vec{b} : bool^{n+1}$ be given and $\vec{b} = (b, \vec{b}')$:

$$[\![(\_, t_1, t_2)]\!]\ \vec{b} = (if\ b\ then\ [\![t_1]\!]\ else\ [\![t_2]\!])\ \vec{b}'$$

Case analysis on b :

$b = true$:

$$\begin{aligned}
[\![(_, t_1, t_2)]\!]\ \vec{b} &= [\![t_1]\!]\ \vec{b}' \\
&= [\![pt_1]\!]\ \vec{b}' && ([\![pt_1]\!] \equiv [\![t_1]\!]) \\
&\triangleq (if\ true\ then\ [\![pt_1]\!]\ else\ [\![pt_2]\!])\ \vec{b}' \\
&= (if\ b\ then\ [\![pt_1]\!]\ else\ [\![pt_2]\!])\ \vec{b}' && (b = true) \\
&= [\![(_, pt_1, pt_2)]\!]\ \vec{b}
\end{aligned}$$

$b' = false$:

$$\begin{aligned}
[\![(_, t_1, t_2)]\!]\ \vec{b} &= [\![t_2]\!]\ \vec{b}' \\
&= [\![pt_2]\!]\ \vec{b}' && ([\![pt_2]\!] \equiv [\![t_2]\!]) \\
&\triangleq (if\ false\ then\ [\![pt_1]\!]\ else\ [\![pt_2]\!])\ \vec{b}' \\
&= (if\ b\ then\ [\![pt_1]\!]\ else\ [\![pt_2]\!])\ \vec{b}' && (b = false) \\
&= [\![(_, pt_1, pt_2)]\!]\ \vec{b}
\end{aligned}$$

$(\_, pt_1, pt_2)$ is reduced, since pt_1 and pt_2 are reduced and $pt_1 \neq pt_2$. ∎

- $dt = t^1$ By recursion we have a prime tree pt with $[\![pt]\!] \equiv [\![t]\!]$. We return pt^1, which is reduced since pt is reduced, and prove $[\![t^1]\!] \equiv [\![pt^1]\!]$.

  Proof. Lemma 4.4 and $[\![pt]\!] \equiv [\![t]\!]$ ∎

Alternatively, we could have also defined *reduce_cert* via recursion on n instead of dt. A subsequent case analysis with DT_Inv gives us the exact same cases. The corresponding proof for DT_{rec} works via induction on n.
Similar to denotational completeness we can project on the prime tree to receive a mapping from decision trees to prime trees.

`Definition` *DT2PT* $\{n : nat\}$ $(dt : DT\ n) : PT\ n := \pi_1$ (*reduce_cert dt*)

4.3.1 Denotational Completeness for prime trees

Combining the mapping $BF2DT : (bool^n \to bool) \to DT\ n$ and $DT2PT : DT\ n \to PT\ n$, allows us to define a first version of a denotational completeness function for prime trees.

`Definition` *BF2PT_cert* $\{n : nat\} : \forall \phi : bool^n \to bool, \{pt : PT\ n \mid [\![pt]\!] \equiv \phi\}$.

For any given $\phi : bool^n \to bool$, $(DT2PT \circ BF2DT)\ \phi$ is the desired prime tree. $[\![(DT2PT \circ BF2DT)\ \phi]\!] \equiv \phi$ follows by the certificates of $DT2PT$ and $BF2DT$ as well as transitivity of $\equiv$ (Lemma 4.1).

For a direct denotational completeness mapping for prime trees, we recall the algorithm for reducing a decision tree t. It has only one interesting case, namely the case where $t = (\_, t_1, t_2)$. Instead of writing a certifying function only for reducing, one could one could also treat this case accordingly during the definition of the denotational completeness result for decision trees, giving us an equivalent and reduced decision tree for an argument function. In other words, we can define a denotational completeness function directly to prime trees. The equivalence proofs of the witness tree and the function are the same as in the denotational completeness for decision trees definition in section 4.2.1 except for the case that we are changing.

`Definition` $PT\_DenCompl\ \{n : nat\} : \forall \phi : bool^n \to bool, \{t : PT\ n \mid [\![t]\!] \equiv \phi\}$.

We describe a function of this type via recursion on n:

- $n = 0$: ϕ is a boolean. We choose $\top$ when $\phi = true$ and $\bot$ when $\phi = false$. Both $\top$ and $\bot$ are reduced by definition.

- $n = n' + 1$: We have $\phi : bool^{n'+1} \to bool$. By recursion, we obtain pt_t and pt_f, the prime trees fulfilling $[\![pt_t]\!] \equiv \phi\ true$ and $[\![pt_f]\!] \equiv \phi\ false$, respectively.
  Case analysis by $=_{dt}$:

  - $pt_t = pt_f$: In this case we return pt_t^1, which is reduced since pt_t is a prime tree. We show $\phi \equiv [\![pt_t^1]\!]$:

    Proof. Let $\vec{b} : bool^{n'+1}$ be given and let $\vec{b} = (b, \vec{b'})$:

    $$\phi\ \vec{b} = (\phi\ b)\ \vec{b'}$$

    Case analysis on b:

    $b = true$:

    $$\begin{aligned}
    (\phi\ b)\ \vec{b'} &= (\phi\ true)\ \vec{b'} \\
    &= [\![pt_t]\!]\ \vec{b'} && ([\![pt_t]\!] \equiv \phi\ true) \\
    &= [\![pt_t^1]\!]\ (b, \vec{b'}) && (\text{Lemma 4.4}) \\
    &= [\![pt_t^1]\!]\ \vec{b} && (\vec{b} = (b, \vec{b'}))
    \end{aligned}$$

    $b = false$:

    $$\begin{aligned}
    (\phi\ b)\ \vec{b'} &= (\phi\ false)\ \vec{b'} \\
    &= [\![pt_f]\!]\ \vec{b'} && ([\![pt_f]\!] \equiv \phi\ false) \\
    &= [\![pt_t]\!]\ \vec{b'} && (pt_t = pt_f) \\
    &= [\![pt_t^1]\!]\ (b, \vec{b'}) && (\text{Lemma 4.4}) \\
    &= [\![pt_t^1]\!]\ \vec{b} && (\vec{b} = (b, \vec{b'}))
    \end{aligned}$$

    ■

  - $pt_t \neq pt_f$: We chose $(\_, pt_t, pt_f)$, which is reduced since $pt_t \neq pt_f$, and both pt_t and pt_f are prime trees and hence also reduced. Only $\phi \equiv [\![(\_, pt_t, pt_f)]\!]$ remains to be shown.

Proof. Let $\vec{b} : bool^{n+1}$ be given and $\vec{b} = (b, \vec{b}')$:

$$\phi\ \vec{b} = (\phi\ b)\ \vec{b}'$$

Case analysis on b :

$b = true$:

$$\begin{aligned}
(\phi\ b)\ \vec{b}' &= (\phi\ true)\ \vec{b}' \\
&= [\![pt_t]\!]\ \vec{b}' && ([\![pt_t]\!] \equiv \phi\ true) \\
&\triangleq (if\ true\ then\ [\![pt_t]\!]\ else\ [\![pt_f]\!])\ \vec{b}' \\
&= (if\ b\ then\ [\![pt_t]\!]\ else\ [\![pt_f]\!])\ \vec{b}' && (b = true) \\
&= [\![(_, pt_t, pt_f)]\!]\ (b, \vec{b}') \\
&= [\![(_, pt_t, pt_f)]\!]\ \vec{b} && (\vec{b} = (b, \vec{b}'))
\end{aligned}$$

$b = false$:

$$\begin{aligned}
(\phi\ b)\ \vec{b}' &= (\phi\ false)\ \vec{b}' \\
&= [\![pt_f]\!]\ \vec{b}' && ([\![pt_f]\!] \equiv \phi\ false) \\
&\triangleq (if\ false\ then\ [\![pt_t]\!]\ else\ [\![pt_f]\!])\ \vec{b} \\
&= (if\ b\ then\ [\![pt_t]\!]\ else\ [\![pt_f]\!])\ \vec{b}' && (b = false) \\
&= [\![(_, pt_t, pt_f)]\!]\ (b, \vec{b}') \\
&= [\![(_, pt_t, pt_f)]\!]\ \vec{b} && (\vec{b} = (b, \vec{b}'))
\end{aligned}$$

■

An interesting question is, if the two denotational completeness versions return the same prime tree. We will answer this question when we have a few more important results available. What we can, however, easily prove is that the two prime trees have equivalent denotations. In fact, this result will be enough to prove equality of the resulting prime trees.

Lemma 4.10. $\forall \phi : bool^n \to bool,\ [\![\pi_1\ (BF2PT\_cert\ \phi)]\!] \equiv [\![\pi_1\ (PT\_DenCompl\ \phi)]\!]$.

Proof. Both $BF2PT\_cert$ and $PT\_DenCompl$ produce prime trees equivalent to their argument function ϕ. From transitivity of $\equiv$ (Lemma 4.1) follows the result. ■

4.3.2 Unique existence of prime trees

Our next aim is to prove that prime trees are indeed a canonical representation of boolean functions. We will start with the core result relating syntactic equality to semantic equality. We need a helping lemma stating three intuitive facts.

Lemma 4.11.

1. *Let* $dt_1, dt_2, dt_3, dt_4 : DT\ n$.
   Then, $(\_, dt_1, dt_2) \neq (\_, dt_3, dt_4) \to dt_1 \neq dt_3 \vee dt_2 \neq dt_4$.
2. *Let* $dt_1, dt_2 : DT\ n$.
   Then, $dt_1^1 \neq dt_2^1 \to dt_1 \neq dt_2$

3. *Let* $dt_1, dt_2, dt_3 : DT\ n$.
   Then, $[\![dt_1^1]\!] \equiv [\![(\_dt_2, dt_3)]\!] \to [\![dt_1]\!] \equiv [\![dt_2]\!] \wedge [\![dt_1]\!] \equiv [\![dt_3]\!]$.

Proof.

1. We have $(\_, dt_1, dt_2) \neq (\_, dt_3, dt_4)$. We compare dt_1 and dt_3 as well as dt_2 and dt_4 by $=_{dt}$. If either $dt_1 \neq dt_3$ or $dt_2 \neq dt_4$ holds then we are done. When $dt_1 = dt_3$ and $dt_2 = dt_4$, then we also have $(\_, dt_1, dt_2) = (\_, dt_3, dt_4)$. $\lightning$

2. We have $dt_1^1 \neq dt_2^1$. Proof by contradiction.
   Assume, $dt_1 = dt_2$. Then we would also have $dt_1^1 = dt_2^1$. $\lightning$

3. We have $[\![dt_1^1]\!] \equiv [\![(\_dt_2, dt_3)]\!]$. Let $\vec{b} : bool^n$ be given. $[\![dt_1]\!]\ \vec{b} = [\![dt_2]\!]\ \vec{b}$ holds since $[\![dt_1^1]\!]\ (true, \vec{b}) = [\![dt_1]\!]\ \vec{b}$ (Lemma 4.4) and $[\![(\_, dt_2, dt_3)]\!]\ (true, \vec{b}) = [\![dt_2]\!]\ \vec{b}$ together with transitivity of $\equiv$ (Lemma 4.1).
   $[\![dt_1]\!]\ \vec{b} = [\![dt_3]\!]\ \vec{b}$ holds analogously using $(false, \vec{b})$ instead of $(true, \vec{b})$.

■

Theorem 4.2. *If* $pt_1, pt_2 : PT\ n$ *are different prime trees, then they denote different boolean functions.*

Proof. We prove $\forall pt_1 pt_2 : PT\ n,\ pt_1 \neq pt_2 \to \exists \vec{b} : bool^n,\ [\![pt_1]\!]\ \vec{b} \neq [\![pt_2]\!]\ \vec{b}$ by induction on n and subsequent case analysis using DT_Inv:

- $n = 0$: We have $pt_1, pt_2 : PT\ 0$ with $pt_1 \neq pt_2$.
  - $pt_1 = \top$ and $pt_2 = \bot$: We suggest $tt : bool^0$ as witness. We have $[\![\top]\!]\ tt = true$ and $[\![\bot]\!]\ tt = false$ and thus the claim.
  - $pt_1 = \bot$ and $pt_2 = \top$: Analogous to previous case.
  - $pt_1 = \bot = pt_2$: $\lightning$ since $pt_1 \neq pt_2$.
  - $pt_1 = \top = pt_2$: $\lightning$ since $pt_1 \neq pt_2$.
- $n \to n + 1$: We have $pt_1, pt_2 : PT\ (n+1)$ with $pt_1 \neq pt_2$ and the inductive hypothesis:

$$\mathcal{IH} : \forall pt_1 pt_2 : PT\ n,\ pt_1 \neq pt_2 \to \exists \vec{b} : bool^n,\ [\![pt_1]\!]\ \vec{b} \neq [\![pt_2]\!]\ \vec{b}.$$

  - $pt_1 = (\_, t_1, t_1')$ and $pt_2 = (\_, t_2, t_2')$: Case analysis via Lemma 4.11 (1):
    - $t_1 \neq t_2$: Via induction we have $\vec{b} : bool^n$ s.t. $[\![t_1]\!]\ \vec{b} \neq [\![t_2]\!]\ \vec{b}$. Our witness is therefore $(true, \vec{b})$ since $[\![(\_, t_1, t_1')]\!]\ (true, \vec{b}) = [\![t_1]\!]\ \vec{b}$ and $[\![(\_, t_2, t_2')]\!]\ (true, \vec{b}) = [\![t_2]\!]\ \vec{b}$ implies also $[\![(\_, t_1, t_1')]\!]\ (true, \vec{b}) \neq [\![(\_, t_2, t_2')]\!]\ (true, \vec{b})$.
    - $t_1' \neq t_2'$: Via induction we have $\vec{b} : bool^n$ s.t. $[\![t_1']\!]\ \vec{b} \neq [\![t_2']\!]\ \vec{b}$. The witness is $(false, \vec{b})$ since $[\![(\_, t_1, t_1')]\!]\ (false, \vec{b}) = [\![t_1']\!]\ \vec{b}$ and $[\![(\_, t_2, t_2')]\!]\ (false, \vec{b}) = [\![t_2']\!]\ \vec{b}$ implies also $[\![(\_, t_1, t_1')]\!]\ (false, \vec{b}) \neq [\![(\_, t_2, t_2')]\!]\ (false, \vec{b})$.
  - $pt_1 = t_1^1$ and $pt_2 = (\_, t_2, t_2')$: $(\_, t_2, t_2')$ is reduced since it is a prime tree and thus we have $t_2 \neq t_2'$. Via induction we have $\vec{b} : bool^n$ s.t. $[\![t_2]\!]\ \vec{b} \neq [\![t_2']\!]\ \vec{b}$. Now assume the opposite of what we want to prove: $[\![(\_t_2, t_2')]\!] \equiv [\![t_1^1]\!]$. By Lemma 4.11 (3) we have $[\![t_2]\!] \equiv [\![t_1]\!] \equiv [\![t_2']\!]$. By Lemma 4.1 (transitivity of $\equiv$), this would, however, mean $[\![t_2]\!] \equiv [\![t_2']\!]$. $\lightning$
  - $pt_1 = (\_, t_1, t_1')$ and $pt_2 = t_2^1$: Analogous to previous case.

– $pt_1 = t_1^1$ and $pt_2 = t_2^1$: Lemma 4.11 (2) gives us $t_1 \neq t_2$. Via induction we have $\vec{b} : bool^n$ s.t. $[\![t_1]\!]\ \vec{b} \neq [\![t_2]\!]\ \vec{b}$. We choose $(true, \vec{b})$ as witness. By Lemma 4.4 we have $[\![t_1^1]\!]\ (true, \vec{b}) = [\![t_1]\!]\ \vec{b}$ and $[\![t_2^1]\!]\ (true, \vec{b}) = [\![t_2]\!]\ \vec{b}$ which gives us $[\![t_1^1]\!]\ (true, \vec{b}) \neq [\![t_2^1]\!]\ (true, \vec{b})$.

■

This gives us enough material to show that every boolean function has exactly one logically equivalent prime tree. We first define a uniqueness predicate as in the Coq library.

```
Definition unique {T : Type} (P : T → Prop) (t : T) :=
  P t ∧ ∀ t' , P t' → t = t'.
```

If *unique P t* is provable then t is the only inhabitant of T for which $P\ t$ is provable. We will use the conventional notation for uniqueness quantification $\exists! t : T, P\ t := \exists t : T, unique\ P\ t$. This notation is also predefined in Coq.

The previous theorem an denotational completeness are the key to prove unique existence of prime trees.

Corollary 4.1 (Unique existence of prime trees). $\forall \phi : bool^n \to bool,\ \exists! t : PT\ n,\ \phi \equiv [\![t]\!]$.

Proof. As witness we choose the tree which denotational completeness for prime trees returns for ϕ. Let $t := \pi_1\ (PT\_DenCompl\ \phi)$. $\phi \equiv [\![t]\!]$ holds because of the certificate that comes with $PT\_DenCompl$.
Uniqueness is still to show: $\forall t' : PT\ n, \phi \equiv [\![t']\!] \to t = t'$. For this part, assume that we have another prime tree t' equivalent to ϕ. Case analysis using $=_{dt}$:

- $t = t'$: Nothing to show.
- $t \neq t'$: By transitivity of $\equiv$ (Lemma 4.1) we know $[\![t]\!] \equiv [\![t']\!]$. Through the previous theorem (Theorem 4.2), however, we know that this is impossible since $t \neq t'$. $\lightning$

■

Corollary 4.2 (Injectivity of $[\![\cdot]\!]$). *Let t_1 and t_2 be two prime trees of the same level. Then,* $[\![t_1]\!] \equiv [\![t_2]\!] \to t_1 = t_2$.

Proof. We have $[\![t_1]\!] \equiv [\![t_2]\!]$ and by Lemma 4.1 ($\equiv$ is reflexive) we also have $[\![t_1]\!] \equiv [\![t_1]\!]$. By Corollary 4.1 with $\phi := [\![t_1]\!]$ we obtain the unique prime tree t equivalent to $[\![t_1]\!]$ and the fact that every other prime tree equivalent to $[\![t_1]\!]$ must be equal to t. We use this fact for t_2 and t_1 to get $t = t_1$ and $t = t_2$. ■

From the above Corollary we can now easily show a result we postponed before:

Corollary 4.3. *BF2PT_cert and PT_DenCompl map to the same prime tree.*

Proof. For $\phi : bool^n \to bool$, we have $[\![\pi_1\ (BF2PT\_cert\ \phi)]\!] \equiv [\![\pi_1\ (PT\_DenCompl\ \phi)]\!]$ through Lemma 4.10. By Corollary 4.2, we also have equality. ■

Next we would like to show that prime trees and cascaded boolean functions depending on n variables are isomorphic:

Lemma 4.12. *Let $n : nat$ be fixed.*

1. $(PT\ n, =)$ *is a total setoid*
2. $(bool^n \to bool, \equiv)$ *is a total setoid.*

Proof.

1. $=$ is an equivalence relation.
2. $\equiv$ is and equivalence relation by Lemma 4.1.

■

Lemma 4.13.

1. $[\![\cdot]\!]$ *is a morphism from* $(PT\ n, =)$ *to* $(bool^n \to bool, \equiv)$ *for any* $n : nat$.
2. $\lambda\phi.\ \pi_1\ (PT\_DenCompl\ \phi)$ *is a morphism from* $(bool^n \to bool, \equiv)$ *to* $(PT\ n, =)$ *for any* $n : nat$.

Proof.

1. We have $t_1, t_2 : PT\ n$ with $t_1 = t_2$. $[\![t_1]\!] \equiv [\![t_2]\!]$ then follows by reflexivity of $\equiv$ (Lemma 4.1).
2. We have $\phi, \psi : bool^n \to bool$ with $\phi \equiv \psi$. Let $t_\phi := \pi_1\ (PT\_DenCompl\ \phi)$ and $t_\psi := \pi_1\ (PT\_DenCompl\ \psi)$. Since $[\![t_\phi]\!] \equiv [\![t_\psi]\!]$ holds through the certificates coming with denotational completeness and transitivity of $\equiv$ (Lemma 4.1), we have $t_\phi = t_\psi$ by Corollary 4.2.

■

Theorem 4.3. $\forall n : nat$,
$SetoidIsomorphism\ (PT\ n, =)\ (bool^n \to bool, \equiv)\ [\![\cdot]\!]\ (\lambda\phi.\ \pi_1\ (PT\_DenCompl\ \phi))$.

Proof. We need to show :

1. $\forall t : PT\ n,\ t = \pi_1\ (PT\_DenCompl\ [\![t]\!])$.
2. $\forall \phi : bool^n \to bool,\ \phi \equiv [\![\pi_1\ (PT\_DenCompl\ \phi)]\!]$.

(2) follows immediately via certificate of $PT\_DenCompl$. We show (1):
Let $t : PT\ n$ be given. We have $[\![\pi_1\ (PT\_DenCompl\ [\![t]\!])]\!] \equiv [\![t]\!]$ through the certificate of $PT\_DenCompl$.
Since $[\![\cdot]\!]$ is injective (Corollary 4.2) we have the claim. ■

By using Σ types we can gather prime trees and boolean functions of all levels. We first show that this yields total setoids. For this we extend the definition of $\equiv$ to Σ types. Let (n, ϕ) and (m, ψ) be of type $\Sigma_n(bool^n \to bool)$:

$$(n, \phi) \equiv_\Sigma (m, \psi) := \begin{cases} \phi \equiv \psi & , n = m \\ False & , n \neq m \end{cases}$$

Lemma 4.14.

1. $(\Sigma_n(bool^n \to bool), \equiv_\Sigma)$ *is a total setoid.*
2. $(\Sigma_n(PT\ n), =)$ *is a total setoid.*

Proof.

1. We show that $\equiv_\Sigma$ is an equivalence relation.
   - Reflexivity: For any (n, ϕ), $(n, \phi) \equiv_\Sigma (n, \phi)$ boils down to showing $\phi \equiv \phi$, which we have by Lemma 4.1.
   - Symmetry: Let (n, ϕ) and (m, ψ) with $(n, \phi) \equiv_\Sigma (m, \psi)$ be given.
     If $n = m$ holds, then we have to show $\psi \equiv \phi$ from $\phi \equiv \psi$ which we have already done (Lemma 4.1).
     If $n \neq m$, then $(n, \phi) \equiv_\Sigma (m, \psi)$ is a proof of *False*. $\lightning$
   - Transitivity: Let (n, ϕ), (m, ψ) and (k, ξ) with $(n, \phi) \equiv_\Sigma (m, \psi) \equiv_\Sigma (k, \xi)$ be given.
     If $n = m = k$, then we have to show $\phi \equiv \xi$ from $\phi \equiv \psi \equiv \xi$ which is the transitivity property of $\equiv$ already shown in Lemma 4.1.
     If $n \neq m$ or $m \neq k$, then we have a proof of *False*. $\lightning$
2. Equality is an equivalence relation.

■

Note, that the proof of Lemma 4.14 (1), while easy on paper, is trickier than one expects. It requires a lemma stating that the decidable equality procedure for nat, say $nat\_eq\_cert : \forall n\, m : nat,\ \{n = m\} + \{n \neq m\}$, always computes the same proof of $n = n$. To be more precise, we need the following: $nat\_eq\_cert\ n\ n = left\ (n \neq n)\ (eq\_refl\ n)$, where eq_refl is the sole constructor of Coq's inductive equality proposition eq and $left$ one of the constructors of *sumbool* indicating that we prove the left proposition. This lemma will be used several times in the Coq proofs of the next results (Lemma 4.15 and Theorem 4.4). Without such a lemma, we would run into dependency issues, only solvable using Coq's *Program* package, which assumes additional axioms that imply *Uniqueness of Identity Proofs (UIP)*.
We reuse the morphisms from the level dependent case to define morphisms for the Σ case.

`Definition` $PT2BF_\Sigma$ $(P : \{n{:}nat\ \&\ PT\ n\}) : \{n{:}nat\ \&\ bool^n \to bool\} :=$
  `let` $(n, t) :=$ P `in` $(existT$ (`fun` $n \Rightarrow bool^n \to bool)\ n\ [\![t]\!])$.

`Definition` $BF2PT_\Sigma$ $(F{:}\ \{n{:}nat\ \&\ bool^n \to bool\}) : \{n{:}nat\ \&\ PT\ n\} :=$
  `let` $(n, \phi) :=$ F `in` $(existT$ (`fun` $n \Rightarrow PT\ n)\ n\ (\pi_1\ (PT\_DenCompl\ \phi))$).

Lemma 4.15.

1. *$PT2BF_\Sigma$ is a morphism from $\Sigma_n(PT\ n)$ to $\Sigma_m(bool^m \to bool)$.*
2. *$BF2PT_\Sigma$ is a morphism from $\Sigma_n(bool^n \to bool)$ to $\Sigma_m(PT\ m)$*

Proof.

1. We have $(n, t_1), (m, t_2) : \Sigma_n(PT\ n)$ with $(n, t_1) = (m, t_2)$.
   By $(n, t_1) = (m, t_2)$ we also have $n = m$ and $t_1 = t_2$.[2]
   $[\![t_1]\!] \equiv [\![t_2]\!]$ then holds by reflexivity of $\equiv$ (Lemma 4.1).
2. We have $(n, \phi), (m, \psi) : \Sigma_n(bool^n \to bool)$ with $(n, \phi) \equiv_\Sigma (m, \psi)$.
   If $n = m$ then we have $\phi \equiv \psi$ and have to show $\pi_1\ (PT\_DenCompl\ \phi) = \pi_1\ (PT\_DenCompl\ \psi)$.
   This holds through Lemma 4.13.
   If $n \neq m$, then we have a proof of *False*. $\lightning$

■

[2] This is one of the rare cases where we actually obtain $n = m$ and $t_1 = t_2$ from $(n, t_1) = (m, t_2)$ through the *inversion* tactic. Usually Coq's *Program* package is necessary to show such results for members of dependent sums.

Theorem 4.4. *SetoidIsomorphism* $(\Sigma_n(PT\ n), =)\ (\Sigma_n(bool^n \to bool), \equiv_\Sigma)\ PT2BF_\Sigma\ BF2PT_\Sigma$.

Proof. We need to show :

1. $\forall(n,\phi) : \Sigma_n(bool^n \to bool),\ (n,\phi) \equiv_\Sigma PT2BF_\Sigma\ (BF2PT_\Sigma\ (n,\phi))$.
2. $\forall(n,t) : \Sigma_n(PT\ n),\ (n,t) = BF2PT_\Sigma\ (PT2BF_\Sigma\ (n,t))$.

1. Both $BF2PT_\Sigma$ and $PT2BF_\Sigma$ produce trees and functions of the same level as their argument, meaning that showing $(n,\phi) \equiv_\Sigma PT2BF_\Sigma\ (BF2PT_\Sigma\ (n,\phi))$ boils down to showing $\phi \equiv [\![\pi_1\ (PT\_DenCompl\ \phi)]\!]$, which holds because of Theorem 4.3.
2. Both $BF2PT_\Sigma$ and $PT2BF_\Sigma$ produce trees and functions of the same level as their argument, meaning that showing $(n,t) = BF2PT_\Sigma\ (PT2BF_\Sigma\ (n,t))$ boils down to showing $t = \pi_1\ (PT\_DenCompl\ [\![t]\!])$, which holds because of Theorem 4.3.

■

To make the isomorphism useful in any context, it is desirable to replace $\equiv_\Sigma$ with $=$. Unfortunately $SetoidIsomorphism\ (\Sigma_n(PT\ n), =)\ (\Sigma_n(bool^n \to bool), =)\ PT2BF_\Sigma\ BF2PT_\Sigma$ is unprovable. We fix this by assuming functional extensionality (FE). In a first step we prove that $(PT\ n, =)$ and $(bool^n \to bool, =)$ are setoid-isomorphic for a fixed n. $(bool^n \to bool, =)$ is a setoid for every $n : nat$ since $=$ is an equivalence relation. That $[\![\cdot]\!]$ and $(\lambda\phi.\ \pi_1\ (PT\_DenCompl\ \phi))$ are still morphisms follows by f_equal (Lemma 4.6).

Theorem 4.5. *We assume FE.* $\forall n : nat$,
$SetoidIsomorphism\ (PT\ n, =)\ (bool^n \to bool, =)\ [\![\cdot]\!]\ (\lambda\phi.\ \pi_1\ (PT\_DenCompl\ \phi))$.

Proof. We need to show :

1. $\forall t : PT\ n,\ t = \pi_1\ (PT\_DenCompl\ [\![t]\!])$.
2. $\forall\phi : bool^n \to bool,\ \phi = [\![\pi_1\ (PT\_DenCompl\ \phi)]\!]$.

(1) has already been shown in Theorem 4.3.
We show (2): Also by Theorem 4.3 we have $\forall\phi : bool^n \to bool,\ \phi \equiv [\![\pi_1\ (PT\_DenCompl\ \phi)]\!]$. By cascaded functional extensionality which follows from FE (Lemma 4.3) we have the claim. ■

The Σ version of the isomorphism is now easy to show. $(\Sigma_n(PT\ n), =)$ and $(\Sigma_n(bool^n \to bool), =)$ are setoids since $=$ is an equivalence relation and $PT2BF_\Sigma$, $BF2PT_\Sigma$ are morphisms between these setoids by f_equal (Lemma 4.6).

Theorem 4.6. *We assume FE.*
$SetoidIsomorphism\ (\Sigma_n(PT\ n), =)\ (\Sigma_n(bool^n \to bool), =)\ PT2BF_\Sigma\ BF2PT_\Sigma$.

Proof. We need to show :

1. $\forall(n,\phi) : \Sigma_n(bool^n \to bool),\ (n,\phi) = PT2BF_\Sigma\ (BF2PT_\Sigma\ (n,\phi))$.
2. $\forall(n,t) : \Sigma_n(PT\ n),\ (n,t) = BF2PT_\Sigma\ (PT2BF_\Sigma\ (n,t))$.

(2) follwos immediately through Theorem 4.3. We show (1):
Both $BF2PT_\Sigma$ and $PT2BF_\Sigma$ produce trees and functions of the same level as their argument, meaning that showing $(n,\phi) = PT2BF_\Sigma\ (BF2PT_\Sigma\ (n,\phi))$ boils down to showing $\phi = [\![\pi_1\ (PT\_DenCompl\ \phi)]\!]$, which holds because of Theorem 4.3 and cascaded functional extensionality (Lemma 4.3). ■

Chapter 5

Simply typed decision trees

In this chapter we present another formalization of decision trees as a simple inductive type (Section 5.1). We will introduce an alternative definition of boolean functions in Section 5.2 and show that the elim restriction (Section 3.2) is a major problem in this development. In Sections 5.3, 5.4 and 5.5, we suggest three different approaches to work around the elim restriction, which will lead to the use of axioms. In all three versions we will, in the end, be able to prove that prime trees are isomorphic to boolean functions. In the last Section 5.6 we will shortly present yet another representation of boolean functions, which we prove to be isomorphic to the previously introduced representation by using partial setoids.

5.1 Simple decision trees

In Chapter 4 we gained some first hand experience about the difficulties that arise while using dependent inductive types. Seemingly easy tasks like defining an equality deciding procedure require quite a bit of infrastructure before a definition is possible. Therefore we have hope, that we can come up with a simple inductive type for decision trees that does not require such complicated ideas.

Syntax

This new definition is now a straightforward translation of the mathematical definition given in Chapter 2, where we use $V := nat$.

```
Inductive SDT : Type :=
  | DT0 : SDT
  | DT1 : SDT
  | DTI : nat → SDT → SDT → SDT.
```

The first thing to notice is that the type SDT does not have a dependent return type. There is no natural number indicating the number of variables a decision tree depends on. Instead, we indicate explicitly, with a natural number, on which variable to branch. We also have but three instead of four constructors in contrast to $DT\ n$ of Chapter 4: DT_0, DT_1 for $\bot$, $\top$ respectively and the branching constructor DT_I taking a natural number and two arbitrary decision trees.

Semantics

Defining the denotational semantics $[\![\cdot]\!]$ for SDT is a straightforward task. We provide an assignment for variables as a function $\sigma : nat \to bool$, which leads to the following definition:

```
Fixpoint SDT_Den (t : SDT) (σ : nat → bool): bool :=
  match t with
    | DT_0 ⇒ false
    | DT_1 ⇒ true
    | DT_I n t_1 t_2 ⇒ if σ n then (SDT_Den t_1 σ) else (SDT_Den t_2 σ)
  end.
```

As expected we have $[\![\bot]\!]\,\sigma = false$ and $[\![\top]\!]\,\sigma = true$ for every assignment σ. For $[\![(n, t_1, t_2)]\!]\,\sigma$ we decide with $\sigma\ n$ which subtree to evaluate next.

Following the development of Chapter 4, we define a procedure that decides equality on SDT.

```
Definition eq_dt_cert (t_1 t_2 : SDT) : {t_1 = t_2} + {t_1 ≠ t_2}.
```

With SDT being a simple inductive type, we need no further infrastructure to compare two decision trees. This fact even allows us to derive such a function automatically in Coq using the tactic *decide equality*, which can compare two elements of a simple inductive type, whose constructors do not take proofs or functions as arguments.
We prove a few facts about simple decision trees.

Lemma 5.1. *Let n, m : nat and t_1, t_2, t_3, t_4 : SDT. Then,*

1. $(n, t_1, t_2) = (m, t_3, t_4) \to n = m \wedge t_1 = t_3 \wedge t_2 = t_4$
2. $(n, t_1, t_2) \neq (m, t_3, t_4) \to n \neq m \vee t_1 \neq t_3 \vee t_2 \neq t_4$

Proof.

1. The tactic *injection* applied to the proof of $(n, t_1, t_2) = (m, t_3, t_4)$ gives us all three conjuncts.
2. We have $(n, t_1, t_2) \neq (m, t_3, t_4)$.
   We compare t_1 with t_3 and t_2 with t_4 using $=_{dt}$ as well as n with m. If we obtain one inequality by these comparisons we are done. If $t_1 = t_3$, $t_2 = t_4$ and $n = m$, then we have $(n, t_1, t_2) = (m, t_3, t_4)$. $\lightning$

■

Since we explicitly specify the variable on which an internal node branches, we sometimes have to know if a given variable is present in a decision tree. We do this by recursion on the tree.

```
Fixpoint inTree (n : nat) (dt : SDT) : bool:=
  match dt with
    | (m, t_1, t_2) ⇒ if (n = m) then true else inTree n t_1 ∨ inTree n t_2
    | _ ⇒ false
  end.
```

Obviously, we return $false$ for leaf nodes, since they do not branch on variables. For a branching node, we return *true*, when it branches on the variable we are looking for. If it branches on a different variable, it suffices when n occurs in either subtree t_1 or t_2. We use intuitive notation $n \in \mathcal{V}\ t$ for *inTree n t*.

Lemma 5.2. *Let m : nat and (n, t_1, t_2) : SDT be given.*
Then, $m \notin \mathcal{V}\ (n, t_1, t_2) \to m \notin \mathcal{V}\ t_1 \wedge m \notin \mathcal{V}\ t_2$

Proof. Proof by contradiction.
Since *inTree* returns a boolean, we know from logic $m \notin \mathcal{V}\ t_1 \wedge m \notin \mathcal{V}\ t_2 \iff \neg(m \in \mathcal{V}\ t_1 \vee m \in \mathcal{V}\ t_2)$. We assume $m \in \mathcal{V}\ t_1 \vee m \in \mathcal{V}\ t_2$. In either case, we would have $m \in \mathcal{V}\ (n, t_1, t_2)$. $\lightning$ ■

Simple prime trees

Exactly analogous to the development for dependent decision trees we define a predicate *reduced* : $SDT \to bool$, which tests if a decision tree is reduced. Since we explicitly give the variable on which a node branches, we also have to come up with a procedure testing whether a decision tree is ordered with respect to the conventional 'less than' order < on natural numbers. This happens exactly like described in Chapter 2: We first specify when a decision tree is ordered w.r.t a variable with *ordered′* : $SDT \to nat \to bool$, then using *ordered′*, we define the *ordered* predicate.

```
Fixpoint ordered' (dt : SDT) (n : nat) : bool :=
  match dt with
    | (m, t1, t2) ⇒ if (n > m) then ordered' t1 m ∧ ordered' t2 m else false
    | _ ⇒ true
  end.

Definition ordered (dt : SDT) : bool :=
  match dt with
    | (n, t1, t2) ⇒ ordered' t1 n ∧ ordered' t2 n
    | _ ⇒ true
  end.
```

We prove a few properties about *ordered* and *ordered′*.

Lemma 5.3.

1. *Let* $n : nat$ *and* $t : SDT$*. Then,* $ordered' \ t \ n \to ordered \ t$
2. *Let* $n : nat$ *and* $t : SDT$*. Then,* $\forall m : nat, n > m \to ordered' \ t \ m \to ordered' \ t \ n$
3. *Let* $n : nat$ *and* $t_1, t_2 : SDT$*. Then,* $ordered \ (n, t_1, t_2) \to ordered \ t_1 \wedge ordered \ t_2$

Proof.

1. Case analysis on t: If $t = \top$ or $t = \bot$ then t is ordered by definition.
   So let $t = (m, t_1, t_2)$. We have $ordered' \ (m, t_1, t_2) \ n$ and have to show that $ordered \ (m, t_1, t_2)$, which means showing $ordered' \ t_1 \ m \wedge ordered' \ t_2 \ m$. Case analysis:
   - $n > m$: In this case $ordered' \ (m, t_1, t_2) \ n$ reduces to $ordered' \ t_1 \ m \wedge ordered' \ t_2 \ m$, which we had to show.
   - $n \not> m$: $ordered' \ (m, t_1, t_2) \ n$ is by coercion a proof of $False$. $\lightning$
2. Case analysis on t: If $t = \top$ or $t = \bot$ then t is ordered w.r.t every variable.
   Let $t = (k, t_1, t_2)$. We have $ordered' \ (k, t_1, t_2) \ m$ and have to show that $ordered' \ (k, t_1, t_2) \ n$. Case analysis:
   - $m > k$: In this case $ordered' \ (k, t_1, t_2) \ m$ reduces to $ordered' \ t_1 \ k \wedge ordered' \ t_2 \ k$. Since $n > m$ by assumption, we have $n > k$ by transitivity of >, which also gives us $ordered' \ (k, t_1, t_2) \ n$.
   - $m \not> k$: $ordered' \ (k, t_1, t_2) \ m$ is by coercion a proof of $False$. $\lightning$
3. $ordered \ (n, t_1, t_2)$ gives us by definition $ordered' \ t_1 \ n \wedge ordered' \ t_2 \ n$. By (1) we also have $ordered \ t_1 \wedge ordered \ t_2$

■

With *ordered* and *reduced* defined, the definition of SPT is just a formality:

`Definition` *SPT* : `Type` := {*dt* : *SDT* | *reduced dt* ∧ *ordered dt* }.

Note that ∧ denotes boolean conjunction.
As before in Section 4.2, we want to use simply typed prime trees as decision trees. The following lemmas allows us to to so:

Lemma 5.4. *Let t be a prime tree.* $\forall p_1\ p_2 : \mathit{reduced}\ t \wedge \mathit{ordered}\ t,\ p_1 = p_2.$

Proof. Case analysis:

- $\mathit{reduced}\ t = \mathit{true} = \mathit{ordered}\ t$: By coercion p_1 and p_2 are two proofs of *True*, since $\mathit{true} \wedge \mathit{true} = \mathit{true}$. *True* has only one proof, namley I, implying that $p_1 = I = p_2$.
- In all the other cases we have a proof of *False* by coercion. $\lightning$

■

Lemma 5.5. *Let t_1 and t_2 be two prime trees. Then,* $\pi_1\ t_1 = \pi_1\ t_2 \rightarrow t_1 = t_2.$

Proof. We have $\pi_1\ t_1 = \pi_1\ t_2$ already, so only equality of the proofs has to be shown.
Through $\pi_1\ t_1 = \pi_1\ t_2$, we have two proofs of $\mathit{reduced}\ (\pi_1\ t_1) \wedge \mathit{ordered}\ (\pi_1\ t_1)$, which have to be equal by Lemma 5.4. ■

`Coercion` *PT_as_DT* (*pt* : *SPT*) := $\pi_1\ pt$.

5.2 Alternative definition of boolean functions

After defining prime trees without any complication, we consider the question whether *SPT* and $\Sigma_n(\mathit{bool}^n \rightarrow \mathit{bool})$ are (partial) setoid-isomorphic. While there is a possibility that these two types are (partial) setoid-isomorphic, the mappings that form the isomorphism certainly do not preserve the meaning of the related objects.

Theorem 5.1. *Let $\asymp$ be some relation on $\Sigma_n(\mathit{bool}^n \rightarrow \mathit{bool})$. $(\Sigma_n(\mathit{bool}^n \rightarrow \mathit{bool}), \asymp)$ and $(SPT, =)$ are not (partial) setoid-isomorphic while preserving the semantics.*

Proof. Assume the semantics would be preserved by mapping back and forth. Consider $\lambda\_.\mathit{true} : \mathit{bool}^1 \rightarrow \mathit{bool}$ and $\lambda\_\_.\mathit{true} : \mathit{bool}^2 \rightarrow \mathit{bool}$. Both represent a *constant true* function, thus their corresponding prime trees shouldn't have to branch on any variable, implying that they both have the same prime tree $\top$. Thus, prime trees would not be canonical for boolean functions. ■

The difference of dependent prime trees to simple prime trees in this argument is that, $\lambda\_.\mathit{true}$ is mapped to $\top^1$ while $\lambda\_\_.\mathit{true}$ is mapped to $\top^2$ with $\top^1 \neq \top^2$, while for simple prime trees both are mapped to $\top$. Consequently, we have to come up with a new type for boolean functions. Chapter 2 defined boolean function theoretically as $(V \rightarrow \mathbb{B}) \rightarrow \mathbb{B}$. We will follow this approach by defining boolean functions as follows.

`Definition` *BF* : `Type` := $(\mathit{nat} \rightarrow \mathit{bool}) \rightarrow \mathit{bool}$.

In fact, this invalidates the argument of Theorem 5.1, because there is only one *constant true* function $\lambda\_ : (\mathit{nat} \rightarrow \mathit{bool}).\ \mathit{true}$. We have, however, to take care of another problem. Since we redefined boolean functions to be of type $(\mathit{nat} \rightarrow \mathit{bool}) \rightarrow \mathit{bool}$, we have infinitely many variables. This

means that a corresponding decision tree for such a function would need to have infinitely many branching nodes. DT cannot contain such a tree since it is an inductive type, implying that all its inhabitants are constructed through finitely many member constructors. For $\sigma : nat \to bool$ the following function φ serves as an example [1]:

$$\varphi\ \sigma := \begin{cases} true & ,\forall x,\ \sigma\ x = true \\ false & ,\text{otherwise} \end{cases} \tag{5.1}$$

We therefore restrict ourselves to only those members of $(nat \to bool) \to bool$, where there is a last variable n, s.t. all subsequent variables $m, m \geq n$ are unimportant for the function. These functions are exactly those boolean functions we will refer to as *continuous* boolean functions. We express this in Coq as follows:

```
Definition cts' (n : nat) (φ : BF) : Prop :=
  ∀ σ1 σ2, (∀ m, m < n → σ1 m = σ2 m) → φ σ1 = φ σ2.
```

We will write $cts_n\ \phi$ for $cts'\ n\ \phi$ and we will refer to n as a *modulus of continuity*.

```
Definition cts (φ : BF) : Prop := ∃ n:nat , cts_n φ .
```

Before continuing the development we make a few definitions and prove a few lemmas which we will need frequently.

```
Definition update (n:nat) (b : bool) (σ : nat→bool) : (nat→bool) :=
  fun m ⇒ if (n = m) then b else (σ m).

Definition BFeq (ψ φ : BF) := ∀ σ : nat → bool, φ σ = ψ σ.
```

We will write σ_b^n for $update\ n\ b\ \sigma$ and we will use $\equiv$ as infix notation for $BFeq$.

Lemma 5.6. *BFeq is an equivalence relation.*

Proof. We use the fact that = is an equivalence relation. ∎

Lemma 5.7.

1. *Let $n : nat$ and $\sigma : nat \to bool$. Then, $\forall b : bool,\ \sigma_b^n\ n = b$*
2. *Let $n, m : nat$ with $n \neq m$ and $\sigma : nat \to bool$. Then, $\forall b : bool,\ \sigma_b^n\ m = \sigma\ m$*

Proof.

1. $\sigma_b^n\ n = (fun\ m \Rightarrow if\ (n = m)\ then\ b\ else\ (\sigma\ m))\ n = b$
2. $\sigma_b^n\ m = (fun\ m \Rightarrow if\ (n = m)\ then\ b\ else\ (\sigma\ m))\ m \overset{n \neq m}{=} \sigma\ m$

∎

We define a function, which decides from a proof of $cts_0\ \phi$, whether ϕ is constant false or constant true.

```
Definition constants (φ : BF)(e : cts_0 φ) : {∀ σ, φ σ = true} + {∀ σ, φ σ = false}.
```

Let $\sigma_f := \lambda\_.\ false$. Case analysis:

- $\phi\ \sigma_f = true$: We pick the left hand side.
  Let σ be given. We show $\phi\ \sigma = true$.

[1] If one would translate this definition into Coq, φ would have type $(nat \to bool) \to Prop$ because of the universal quantification. We don't know, however, if $(nat \to bool) \to bool$ does not contain a similar function to φ, which is why we have to account for such functions as well.

Proof. Since ϕ has modulus of continuity 0, we immediately have $\phi\,\sigma = \phi\,\sigma_f = \mathit{true}$, if we can show that $\forall m,\ m < 0 \to \sigma_f\ m = \sigma\ m$. Since there is no such m, we are done. ∎

- $\phi\ \sigma_f = \mathit{false}$: We pick the right hand side and show that for any given σ, we have $\phi\,\sigma_f = \phi\,\sigma = \mathit{false}$ as in the previous case.

We show that fixing the assignment of the last important variable reduces the modulus of continuity of a non-constant boolean function.

Lemma 5.8. *Let $\phi : BF$ such that $cts_{n+1}\ \phi$ holds. Then, $\forall b : bool,\ cts_n\ (\lambda\sigma.\ \phi\ \sigma_b^n)$.*

Proof. Let ϕ with $cts_{n+1}\ \phi$, $b : bool$, $\sigma, \sigma' : nat \to bool$ as well as $\mathcal{H} := \forall m : nat,\ m < n \to \sigma\ m = \sigma'\ m$ be given. We have to show $\phi\ \sigma_b^n = \phi\ {\sigma'}_b^n$. We use the fact that $cts_{n+1}\ \phi$ and thus it is enough to show $\forall m : nat,\ m < n+1 \to \sigma_b^n\ m = {\sigma'}_b^n\ m$. We have this immediately by $\mathcal{H}$ and Lemma 5.7. ∎

The following results concerning *ordered*, *ordered′*, *inTree* and *update* will be important later on.

Lemma 5.9.

1. *Let n : nat and t : SDT s.t. ordered′ t n holds. Then, $\forall m : nat,\ m \geq n \to m \notin \mathcal{V}\ t$*
2. *Let $n : nat, b : bool$ and t : SDT with $n \notin \mathcal{V}\ t$. Then, $\forall \sigma,\ [\![t]\!]\ \sigma_b^n = [\![t]\!]\ \sigma$*

Proof.

1. By induction on t.
   If $t = \top$ or $t = \bot$, then $m \notin \mathcal{V}\ t$ holds by definition.
   Let $t = (n', t_1, t_2)$. We have *ordered′* $(n', t_1, t_2)\ n, m$ with $m \geq n$ and two inductive hypotheses:

$$\mathcal{IH}_1 :\ \mathit{ordered'}\ t_1\ n \to m \notin \mathcal{V}\ t_1$$
$$\mathcal{IH}_2 :\ \mathit{ordered'}\ t_2\ n \to m \notin \mathcal{V}\ t_2$$

   We have to show that $m \notin \mathcal{V}\ (n', t_1, t_2)$. Case analysis:

   - $n > n'$: In this case *ordered′* $(n', t_1, t_2)\ n$ gives us *ordered′* $t_1\ n' \wedge$ *ordered′* $t_2\ n'$ and by Lemma 5.3 we have *ordered′* $t_1\ n \wedge$ *ordered′* $t_2\ n$. Since $n > n'$ and $m \geq n$ we have $m \neq n'$. So $m \notin \mathcal{V}\ (n', t_1, t_2)$ comes down to showing $m \notin \mathcal{V}\ t_1 \wedge m \notin \mathcal{V}\ t_2$. We have both conjuncts by $\mathcal{IH}_1$ and $\mathcal{IH}_2$.
   - $n \not> n'$: By definition *ordered′* $(n', t_1, t_2)\ n = \mathit{false}$. By coercion we then have a proof of *False*. ↯

2. Induction on t.
   If t is a leaf, then the claim holds by definition of $[\![\cdot]\!]$.
   Let $t = (m, t_1, t_2)$ and $\sigma : nat \to bool$, $b : bool$ be given. We have $n \notin \mathcal{V}\ (m, t_1, t_2)$ and two inductive hypotheses:

$$\mathcal{IH}_1 :\ n \notin \mathcal{V}\ t_1 \to [\![t_1]\!]\ \sigma_b^n = [\![t_1]\!]\ \sigma$$
$$\mathcal{IH}_2 :\ n \notin \mathcal{V}\ t_2 \to [\![t_2]\!]\ \sigma_b^n = [\![t_2]\!]\ \sigma$$

   We have to show that $[\![(m, t_1, t_2)]\!]\ \sigma_b^n = [\![(m, t_1 t_2)]\!]\ \sigma$. Case analysis:

   - $m = n$: Then we have $n \notin \mathcal{V}\ (n, t_1, t_2)$. ↯

- $m \neq n$: In this case $n \notin \mathcal{V}\ (m, t_1, t_2)$ gives us $n \notin \mathcal{V}\ t_1 \wedge n \notin \mathcal{V}\ t_2$ by Lemma 5.2. By $\mathcal{IH}_1$ and $\mathcal{IH}_2$ we thus have $[\![t_1]\!]\ \sigma_b^n = [\![t_1]\!]\ \sigma$ and $[\![t_2]\!]\ \sigma_b^n = [\![t_2]\!]\ \sigma$. By Lemma 5.7 we also have $\sigma_b^n\ m = \sigma\ m$ and the claim follows.

■

We define a type which contains only the continuous boolean functions as follows.

`Definition` $BF_{cts} := \{\phi : BF \mid cts\ \phi\}$.

We will denote members of BF_{cts} as if they were of type BF. With $\phi : BF_{cts}$ we thus mean that ϕ is a function for which $cts\ \phi$ holds. Using BF_{cts} as type for boolean functions we will, however, not be able to define a denotational completeness function, which will be one of the morphisms we use to show that boolean functions and simple prime trees are setoid-isomorphic.

$$\forall \phi : BF_{cts},\ \{pt : SPT \mid \phi \equiv [\![pt]\!]\}.$$

The reason is that we would need to extract the modulus of continuity from the $cts\ \phi$ proof to construct a member of $\{pt : SPT \mid \phi \equiv [\![pt]\!]\} : Type$. The elim restriction, however, does not permit such scenarios. Without modifications to our definition of BF_{cts}, we are stuck. When the modulus of continuity is available, however, we are able to define an auxiliary function from which we can easily derive the denotational completeness result.

`Definition` $SPT\_DenCompl\_aux\ \{n : nat\} : \forall \phi : BF, cts_n\ \phi \rightarrow$
$\{t : SDT \mid [\![t]\!] \equiv \phi \wedge reduced\ t \wedge ordered'\ t\ n\ \}$.

We do recursion on n:

- $n = 0$: We have ϕ with $cts_0\ \phi$. By *constants*, ϕ is either constant true or constant false. The correct prime trees are $\top$ and $\bot$ respectively. Both $\bot$ and $\top$ are ordered and reduced by definition and are equivalent to ϕ in their respective case.
- $n = n' + 1$: We have ϕ with $cts_{n'+1}\ \phi$. By Lemma 5.8, both $\phi_t := \lambda\sigma.\ \phi\ \sigma_{true}^n$ and $\phi_f := \lambda\sigma.\ \phi\ \sigma_{false}^n$ have modulus of continuity n. We can therefore use recursion to compute decision trees t_1 and t_0 with $[\![t_1]\!] \equiv \phi_t$, $[\![t_0]\!] \equiv \phi_f$, $reduced\ t_1$, $ordered'\ t_1\ n'$, $reduced\ t_0$ and $ordered'\ t_0\ n'$.

  Before we decide which tree to return, we first prove $\mathcal{H} := [\![(n', t_1, t_0)]\!] \equiv \phi$:

  Proof. Let $\sigma : nat \rightarrow bool$ be given.

  $$\phi\ \sigma = [\![(n', t_1, t_0)]\!]\ \sigma$$

  Case analysis on $\sigma\ n'$:

  $\sigma\ n' = true$:

  $$\begin{aligned} [\![(n', t_1, t_0)]\!]\ \sigma &= [\![t_1]\!]\ \sigma \\ &= \phi\ \sigma_{true}^{n'} && ([\![t_1]\!] \equiv \phi_t) \end{aligned}$$

  By using the fact that ϕ has modulus of continuity $n' + 1$, we have $\phi\ \sigma_{true}^{n'} = \phi\ \sigma$, if we can prove $\forall m, m < n' + 1 \rightarrow \sigma_{true}^{n'}\ m = \sigma\ m$. This is the case, because of Lemma 5.7 and the fact that $\sigma\ n' = true$.

  $\sigma\ n' = false$:

  $$\begin{aligned} [\![(n', t_1, t_0)]\!]\ \sigma &= [\![t_0]\!]\ \sigma \\ &= \phi\ \sigma_{false}^{n'} && ([\![t_0]\!] \equiv \phi_f) \end{aligned}$$

Analogous to the previous case we have $\phi\ \sigma^{n'}_{false} = \phi\ \sigma$. ∎

To decide which tree to return, a case analysis using $=_{dt}$ is necessary:

- $t_1 = t_0$: In this case we take t_1 as witness, which is already reduced and ordered w.r.t. n'. Through Lemma 5.3, t_1 is also ordered w.r.t. $n' + 1$. In addition we have $[\![t_1]\!] \equiv [\![(n', t_1, t_1)]\!] \equiv [\![(n', t_1, t_0)]\!]$ since $t_1 = t_0$ and $[\![(n', t_1, t_0)]\!] \equiv \phi$ through $\mathcal{H}$ and thus also $[\![t_1]\!] \equiv \phi$.
- $t_1 \neq t_0$: We use the witness (n', t_1, t_0), which is reduced since $t_1 \neq t_0$, *reduced* t_0 and *reduced* t_1 holds. We also have *ordered'* (n', t_1, t_0) $(n' + 1)$ since $n' + 1 > n'$, *ordered'* t_1 n' and *ordered'* t_0 n'. That $[\![(n', t_1, t_0)]\!] \equiv \phi$ holds, has been shown above $(\mathcal{H})$.

The reason for this auxiliary function is the following subtle difference between *ordered'* and *ordered*: Just because t_0 and t_1 are ordered, does not imply that (n, t_1, t_0) is ordered, since n might be smaller than the root of t_1 or t_0. For *ordered'* this is different. If *ordered'* t_1 n and *ordered'* t_0 n holds, then we at least know that *ordered'* (n, t_1, t_0) $(n + 1)$ holds, since there are no bigger variables than $n - 1$ in t_1 or t_0.

The definition of a denotational completeness mapping for prime trees will require additional assumptions in form of axioms or a modification of our type for boolean functions. We will therefore postpone it since there are plenty of results left to prove that do not require the assumption of axioms.

We will concentrate on proving that there is exactly one logically equivalent prime tree for a given boolean function. For this we need a bit of infrastructure. To motivate this, look at the proof of Lemma 4.2 of the previous chapter. The inductive hypothesis gives us the result for any two prime trees of the next smaller level. This means, that the inductive hypothesis applies for the subtrees of a branching tree, since they necessarily have a smaller level. This was possible because we did induction on the level of the prime trees. For simple prime trees we have no such thing. Our only hope is induction on the first prime tree, followed by a case analysis on the second. This way, however, it is not possible to use the inductive hypothesis for the two subtrees of a branching tree.

We prove a special version of an induction principle called *size induction*, which will present a solution to that problem.

Lemma 5.10 (double size induction). *Let T : Type, $P : T \to T \to$ Prop and $f : T \to$ nat be a size function on T. Then,*
$(\forall x_1\ x_2,\ (\forall y_1\ y_2,\ (f\ y_1 + f\ y_2) < (f\ x_1 + f\ x_2) \to P\ y_1\ y_2) \to P\ x_1\ x_2) \to \forall x_1\ x_2,\ P\ x_1\ x_2$.

Proof. We have $\mathcal{H} := \forall x_1\ x_2,\ (\forall y_1\ y_2,\ (f\ y_1 + f\ y_2) < (f\ x_1 + f\ x_2) \to P\ y_1\ y_2) \to P\ x_1\ x_2$ and $x_1, x_2 : T$. We have to show $P\ x_1\ x_2$. We immediately apply our assumption $\mathcal{H}$, and have to prove the premiss of $\mathcal{H}$, namely $\forall y_1\ y_2,\ (f\ y_1 + f\ y_2) < (f\ x_1 + f\ x_2) \to P\ y_1\ y_2$.
Induction on $f\ x_1 + f\ x_2$:

- $f\ x_1 + f\ x_2 = 0$: We have y_1, y_2 and $(f\ y_1 + f\ y_2) < 0$. ↯
- $f\ x_1 + f\ x_2 = n + 1$: We have y_1, y_2 and $(f\ y_1 + f\ y_2) < n + 1$ and the inductive hypothesis:

  $$\mathcal{IH} : \forall z_1\ z_2 : T,\ (f\ z_1 + f\ z_2) < n \to P\ z_1\ z_2.$$

  We have to show $P\ y_1\ y_2$.
  We again use assumption $\mathcal{H}$ and receive $z_1, z_2 : T$ with $(f\ z_1 + f\ z_2) < (f\ y_1 + f\ y_2)$ and have

to show $P\ z_1\ z_2$. With $\mathcal{IH}$ we have this, if we can show $(f\ z_1 + f\ z_2) < n$. This is the case since $(f\ z_1 + f\ z_2) < (f\ y_1 + f\ y_2) < n + 1$.

■

We define a suitable size function for decision trees.

```
Fixpoint size (t : SDT) : nat :=
  match t with
    | (n, t1, t2) ⇒ 1 + size t1 + size t2
    | _ ⇒ 0
  end.
```

We will denote *size t* using $|t|$. Note, that $|t|$ is exactly the number of internal nodes of t.
Based on the sum of the sizes of two trees we define a special kind of case analysis.

Lemma 5.11 (Size Case Analysis). *Let $t, t' : SPT$.*

1. *If $|t| + |t'| = 0$, then $(t = \top \vee t = \bot) \wedge (t' = \top \vee t' = \bot)$.*
2. *If $|t| + |t'| = n + 1$ and $t \neq t'$ then either:*
   - $t = (n, t_1, t_2) \wedge n \notin \mathcal{V}\ t'$,
   - $n \notin \mathcal{V}\ t \wedge t' = (n, t_1, t_2)$ *or*
   - $t = (n, t_1, t_2) \wedge t' = (n, t'_1, t'_2)$.

Proof.

1. We have $|t| + |t'| = 0$ and thus $|t| = 0$ and $|t'| = 0$. By definition of $|\cdot|$, the only trees of size 0 are leaves.
2. We have $|t| + |t'| = n + 1$ and $t \neq t'$. Case analysis on t and t':
   Since $|t| + |t'| = n + 1$, not both t and t' can be leaves. 5 cases remain:
   - $t = \top$ and $t' = (n, t_1, t_2)$: The second case applies. We have $n \notin \mathcal{V}\ \bot$ by definition of $\in$.
   - $t = \bot$ and $t' = (n, t_1, t_2)$: Analogous to previous case.
   - $t = (n, t_1, t_2)$ and $t' = \top$: The first case applies. We have $n \notin \mathcal{V}\ \top$ by definition of $\in$.
   - $t = (n, t_1, t_2)$ and $t' = \bot$: Analogous to previous case.
   - $t = (n, t_1, t_2)$ and $t' = (m, t'_1, t'_2)$: Case analysis:
     - $n > m$: The first case applies. We show $n \notin \mathcal{V}\ (m, t'_1, t'_2)$.
       Since $n > m$, we have $n \neq m$ and therefore it suffices to show that $n \notin \mathcal{V}\ t'_1 \wedge n \notin \mathcal{V}\ t'_2$. Since (m, t'_1, t'_2) is a prime tree and therefore both t'_1 and t'_2 are ordered w.r.t. m and by Lemma 5.3 also w.r.t. n. By Lemma 5.9 we have $n \notin \mathcal{V}\ t'_1$ and $n \notin \mathcal{V}\ t'_2$.
     - $n < m$: The second case applies. The proof that $m \notin \mathcal{V}\ (n, t_1, t_2)$ is analogous to the previous case.
     - $n = m$: The third case applies.

■

The next lemma motivates why we will be able to use the above mentioned inductive hypothesis for the subtrees of a branching tree.

Lemma 5.12. *Let $t_1, t_2 : SDT$ and $n : nat$. Then, $\forall m : nat,\ |t_1| + |t_2| < m + |(n, t_1, t_2)|$.*

Proof. By definition of $|\cdot|$ we have that $|(n, t_1, t_2)| = 1 + |t_1| + |t_2| > |t_1| + |t_2|$. ∎

Theorem 5.2. *If $t_1, t_2 : SPT$ are different prime trees, then they denote different boolean functions.*

Proof. To show : $\forall t\, t' : SPT,\ t \neq t' \to \exists \sigma,\ [\![t]\!]\ \sigma \neq [\![t']\!]\ \sigma$.
Let $t, t' : SPT$ with $t \neq t'$ be given. We use Lemma 5.10 to do size induction, which gives us

$$\mathcal{IH} : \forall t_1\, t_2 : SPT,\ |t_1| + |t_2| < |t| + |t'| \to t_1 \neq t_2 \to \exists \sigma,\ [\![t_1]\!]\ \sigma \neq [\![t_2]\!]\ \sigma$$

Case analysis:

- $|t| + |t'| = 0$: By Lemma 5.11 (1) both t and t' are leaves. Since $t \neq t'$ every σ can be used as witness.
- $|t| + |t'| = n + 1$: We do a case analysis according to Lemma 5.11 (2):
  - $t = (n, t_1, t_2) \wedge n \notin \mathcal{V}\ t'$: Since (n, t_1, t_2) is a ordered and reduced, we have that t_1 and t_2 are also reduced, ordered w.r.t n and therefore ordered (Lemma 5.3), in addition to $t_1 \neq t_2$. Through Lemma 5.9 we also have $n \notin \mathcal{V}\ t_1$ and $n \notin \mathcal{V}\ t_2$ and by Lemma 5.12 $|t_1| + |t_2| < |t'| + |t|$ holds. Since all the premisses of $\mathcal{IH}$ are fulfilled, we have σ s.t. $[\![t_1]\!]\ \sigma \neq [\![t_2]\!]\ \sigma$. By Lemma 5.9, we therefore have $[\![t_1]\!]\ \sigma^n_{true} \neq [\![t_2]\!]\ \sigma^n_{false}$ and by this we have $[\![(n, t_1, t_2)]\!]\ \sigma^n_{true} \neq [\![(n, t_1, t_2)]\!]\ \sigma^n_{false}$ because $[\![(n, t_1, t_2)]\!]\ \sigma^n_{true} = [\![t_1]\!]\ \sigma^n_{true} \neq [\![t_2]\!]\ \sigma^n_{false} = [\![(n, t_1, t_2)]\!]\ \sigma^n_{false}$. By Lemma 5.9 we have, however, that $[\![t']\!]\ \sigma^n_{true} = [\![t']\!]\ \sigma^n_{false}$ since $n \notin \mathcal{V}\ t'$. Consequently, we have $[\![t]\!]\ \sigma^n_{true} \neq [\![t']\!]\ \sigma^n_{true}$ or $[\![t]\!]\ \sigma^n_{false} \neq [\![t']\!]\ \sigma^n_{false}$.
  - $n \notin \mathcal{V}\ t \wedge t' = (n, t_1, t_2)$: Analogous to previous case.
  - $t = (n, t_1, t_2) \wedge t' = (n, t'_1, t'_2)$: Since $t \neq t'$ we have $t_1 \neq t'_1$ or $t_2 \neq t'_2$ by Lemma 5.1. Reducedness and orderedness w.r.t n and thus orderedness of t_1, t'_1, t_2 and t'_2 are given by the fact that t and t' are prime trees and Lemma 5.3. We obviously have $|t_1| + |t'_1| < |(n, t_1, t_2)| + |(n, t'_1, t'_2)|$ and $|t_2| + |t'_2| < |(n, t_1, t_2)| + |(n, t'_1, t'_2)|$. By $\mathcal{IH}$ we thus have σ s.t. $[\![t_1]\!]\ \sigma \neq [\![t'_1]\!]\ \sigma$ or $[\![t_2]\!]\ \sigma \neq [\![t'_2]\!]\ \sigma$. Lemma 5.9 gives us that n doesn't occur in either t_1, t'_1, t_2 or t'_2. By Lemma 5.9, we therefore have $[\![t_1]\!]\ \sigma^n_{true} = [\![t_1]\!]\ \sigma$ and $[\![t'_1]\!]\ \sigma^n_{true} = [\![t'_1]\!]\ \sigma$ as well as $[\![t_2]\!]\ \sigma^n_{false} = [\![t_2]\!]\ \sigma$ and $[\![t'_2]\!]\ \sigma^n_{false} = [\![t'_2]\!]\ \sigma$. From this, $[\![t]\!]\ \sigma^n_{true} \neq [\![t']\!]\ \sigma^n_{true}$ or $[\![t]\!]\ \sigma^n_{false} \neq [\![t']\!]\ \sigma^n_{false}$ follows because $[\![(n, t_1, t_2)]\!]\ \sigma^n_{true} = [\![t_1]\!]\ \sigma^n_{true} = [\![t_1]\!]\ \sigma \neq [\![t'_1]\!]\ \sigma = [\![t'_1]\!]\ \sigma^n_{true} = [\![(n, t'_1, t'_2)]\!]\ \sigma^n_{true}$ and $[\![(n, t_1, t_2)]\!]\ \sigma^n_{false} = [\![t_2]\!]\ \sigma^n_{false} = [\![t_2]\!]\ \sigma \neq [\![t'_2]\!]\ \sigma = [\![t'_2]\!]\ \sigma^n_{false} = [\![(n, t'_1, t'_2)]\!]\ \sigma^n_{false}$

∎

Note, that by using size induction, we essentially have to prove the same goals as in the dependent inductive version (Theorem 4.2), where one can interpret $pt_1 = (\_, t_1, t_2)$ and $pt_2 = t'^1$ as the corresponding case to $pt_1 = (n, t_1, t_2)$ and $n \notin \mathcal{V}\ pt_2$. One can think of the size of a simple prime tree as an equivalent to the level of dependent prime trees for the above proof.

Corollary 5.1 (Injectivity of $[\![\cdot]\!]$). *Let t_1 and t_2 be two prime trees.*
Then, $[\![t_1]\!] \equiv [\![t_2]\!] \to t_1 = t_2$.

Proof. Assume $[\![t_1]\!] \equiv [\![t_2]\!]$. We want to know if t_1 and t_2 are already equal and thus do a case analysis using $=_{dt}$. If $t_1 = t_2$ holds, then we have nothing left to prove, so assume $t_1 \neq t_2$. This, however, also implies $[\![t_1]\!] \not\equiv [\![t_2]\!]$ (Theorem 5.2). $\lightning$ ∎

This is as far as we can get without further assumptions to circumvent the elim restriction. The following sections will introduce three different versions of keeping the modulus of continuity of a function available for definitions.

5.3 Version 1: Using the Axiom of Continuity

In this section we will use a rather drastic method to circumvent the elim restriction. We simply assume that every boolean function is continuous. We realize this such that we know the modulus of continuity of a boolean function at all times.
We define a version of the *cts* predicate of type $BF \to Type$.

`Definition` $ctsT\ (\phi : BF) :$ `Type` $:= \{n{:}nat \mid cts_n\ \phi\}$.

`Axiom` $CTS : \forall\ \phi : BF, ctsT\ \phi$.

CTS denotes the so-called *Axiom of Continuity*. When CTS is assumed, we can ask for the modulus of continuity of a function whenever we need it. In Section 5.3.1 at the end of the chapter we will show that CTS is inconsistent with informative classical logic.

Deriving denotational completeness is now an easy task.

`Definition` $SPT\_DenCompl : \forall \phi : BF, \{t : SPT \mid [\![t]\!] \equiv \phi\}$.

> Let $\phi : BF$ be given. Through CTS, we receive $n : nat$ s.t. $cts_n\ \phi$ holds. By the auxiliary function $SPT\_DenCompl\_aux$, we receive a reduced decision tree t, which is logically equivalent to ϕ and ordered w.r.t. n. By Lemma 5.3, t is also ordered, which makes it a prime tree.

Denotational completeness in addition to Theorem 5.2 make it possible to establish the unique existence result.

Corollary 5.2 (Uniqueness of prime trees). $\forall \phi : BF,\ \exists! t : SPT,\ \phi \equiv [\![t]\!]$.

Proof. We use $t := \pi_1\ (SPT\_DenCompl\ \phi)$ as witness. t is equivalent to ϕ by the certificate of $SPT\_DenCompl$. Uniqueness remains to be shown: $\forall t' : SPT,\ [\![t']\!] \equiv \phi \to t' = t$. Assume another prime tree t' logically equivalent to ϕ. Using $=_{dt}$ either $t = t'$ holds, then there is nothing to show, or $t \neq t'$ holds, but then $[\![t]\!] \not\equiv [\![t']\!]$ by Theorem 5.2, which is impossible because $[\![t]\!] \equiv \phi \equiv [\![t']\!]$. $\lightning$ ∎

We show that SPT and BF are setoid-isomorphic. Let

$$BF2SPT := \lambda \phi : BF.\ \pi_1\ (SPT\_DenCompl\ \phi)$$

Lemma 5.13.

1. $(SPT, =)$ *is a setoid.*
2. $(BF, \equiv)$ *is a setoid*
3. $BF2SPT : (BF, \equiv) \twoheadrightarrow (SPT, =)$
4. $[\![\cdot]\!] : (SPT, =) \twoheadrightarrow (BF, \equiv)$

Proof.

1. $=$ is an equivalence relation.
2. $\equiv$ is an equivalence relation by Lemma 5.6.
3. Let $\phi, \psi : BF$ with $\phi \equiv \psi$. Then, by definition of $BF2SPT$ we have $[\![BF2SPT\ \phi]\!] \equiv \phi \equiv \psi \equiv [\![BF2SPT\ \psi]\!]$, but with $[\![\cdot]\!]$ being injective by Corollary 5.1, we have $BF2SPT\ \phi = BF2SPT\ \psi$.
4. Let $t_1, t_2 : SPT$ with $t_1 = t_2$. $[\![t_1]\!] \equiv [\![t_2]\!]$ immediately follows.

■

Theorem 5.3. *SetoidIsomorphism* $(SPT, =)$ $(BF, \equiv)$ $[\![\cdot]\!]$ $BF2SPT$.

Proof.

- $\forall t : SPT,\ t = BF2SPT\ [\![t]\!]$: By definition we have $[\![BF2SPT\ [\![t]\!]]\!] \equiv [\![t]\!]$ and by Corollary 5.1 also $BF2SPT\ [\![t]\!] = t$
- $\forall \phi : BF,\ \phi \equiv [\![BF2SPT\ \phi]\!]$: Immediately by definition of $BF2SPT$.

■

As in the previous chapter we are able, by assuming FE, to prove *SetoidIsomorphism* $(SPT, =)$ $(BF, =)$ $[\![\cdot]\!]$ $BF2SPT$. $(SPT, =)$ and $(BF, =)$ are setoids because $=$ is an equivalence relation. $[\![\cdot]\!]$ and $BF2SPT$ are still morphisms by Lemma 4.6 (f_equal).

Theorem 5.4. *We assume FE.*
SetoidIsomorphism $(SPT, =)$ $(BF, =)$ $[\![\cdot]\!]$ $BF2SPT$.

Proof.

- $\forall t : SPT,\ t = BF2SPT\ [\![t]\!]$: Like in the previous theorem (Theorem 5.3).
- $\forall \phi : BF,\ \phi = [\![BF2SPT\ \phi]\!]$: We have $\forall \phi : BF,\ \phi \equiv [\![BF2SPT\ \phi]\!]$ by Theorem 5.3. FE gives us $\forall \phi : BF,\ \phi = [\![BF2SPT\ \phi]\!]$.

■

5.3.1 CTS vs. CDP

In this section we want to demonstrate the consequences of assuming CTS. Imagine we are in a strong classical environment where we want to assume *Computational Decidability of Propositions* (CDP).

`Axiom` CDP : `Type` := $\forall\ P$: `Prop`, $\{P\} + \{\neg\ P\}$.

We demonstrate that CTS and CDP leads to a proof of *False*.
Since CDP is of type *Type* we can ask if a proposition is provable at all times and in every context. We show, that it is possible, using CDP, to define a non-continuous boolean function. The trick is, to use CDP to define a $\forall$-quantification of type *bool* instead of *Prop*. In fact, with CDP, one can convert every proposition to a boolean as follows.

`Definition` *Prop2bool* (*P* : `Prop`) : *bool* := `if` (*CPD P*) `then` *true* `else` *false*.

Recall the definition of φ (5.1). In Coq, the definition of φ can now be realized as a function having type $(nat \to bool) \to bool$ instead of $(nat \to bool) \to Prop$.

`Definition` *phi* : *BF* := `fun` $\sigma \Rightarrow$ *Prop2bool* ($\forall\, n, \sigma\ n = true$) .

Now let $\sigma^n_\top, \sigma_\top : nat \to bool$ be defined as follows:

$$\sigma^n_\top\ m := \begin{cases} true & , m < n \\ false & , \text{otherwhise} \end{cases}$$
$$\sigma_\top\ m := true$$

It is now possible to show $\varphi\ \sigma^n_\top = \varphi\ \sigma_\top$, where n is the modulus of continuity of φ obtained by CTS.

Theorem 5.5. *Let n be the modulus of continuity of φ obtained via CTS. Then,* $\varphi\ \sigma^n_\top = \varphi\ \sigma_\top$.

Proof. Let n and $cts_n\ \varphi$ be given by CTS.
Then we have $\forall m : nat, m < n \to \sigma^n_\top\ m = true \wedge \sigma_\top\ m = true$, by definition of $\sigma^n_\top$ and $\sigma_\top$. Since φ has modulus of continuity n, we have $\varphi\ \sigma^n_\top = \varphi\ \sigma_\top$. ∎

Obviously, this is not possible since only the first n variables are assigned to *true* by $\sigma^n_\top$ and the rest to $false$. Thus, $\varphi\ \sigma^n_\top$ should be $false$ while $\varphi\ \sigma_\top$ should be *true*. Hence, we have $false = true$ by the above theorem, which is equivalent to *False*.

5.4 Version 2: Using the Axiom of Description

Simply assuming that all boolean functions are continuous is a radical solution. Instead of assuming CTS, we try to find a less controversial approach. We, again, restrict ourselves to only the continuous functions and therefore chose BF_{cts} as type for boolean functions. For this, we use the initial definition of *cts* as existential quantification. Remember, that we had trouble with the elim restriction when we wanted to define a denotational completeness function. In this version of our quest to find an isomorphism between boolean functions and prime trees we do not necessarily need denotational completeness as a function. We will show how to define a mapping from boolean functions to prime trees a little later in this section.
We still need denotational completeness, although not as a function, but as a proposition.

Theorem 5.6 (Denotational Completeness for prime trees). $\forall \phi : BF_{cts},\ \exists t : SPT,\ \phi \equiv [\![t]\!]$.

Proof. Let ϕ and $n : nat$ with $cts_n\ \phi$ be given. By the auxiliary function $SPT\_DenCompl\_aux$ we thus have a reduced and logically equivalent prime tree t with $ordered'\ t\ n$. By Lemma 5.3, t is also ordered an thus a prime tree. ∎

Note, that we did not violate the elim restriction in this proof, since we build *proofs* from proofs and functions.
It is also important to notice that, in a statement of the form $\forall \phi\ \psi : BF_{cts},\ \phi \equiv \psi$, we do not compare the modulus of continuity of ϕ and ψ. We mearly enforce that ϕ and ψ are continuous. For the rest, we simply compare the actual functions using *BFeq*.
From denotational completeness also follows that there exists only one prime tree for a given boolean function.

Corollary 5.3 (Unique existence of prime trees). $\forall \phi : BF_{cts},\ \exists! t : SPT,\ \phi \equiv [\![t]\!]$.

Proof. The proof is exactly analogous to the proof of Corollary 5.2. ∎

It is our aim to show that BF_{cts} and SPT are setoid-isomorphic. It is therefore crucial to show that decision trees, and thus also prime trees, only represent continuous functions. The following function computes the modulus of continuity of the denotation of a decision tree in a certifying manner.

`Definition` $SDT\_Den\_cts\ (t : SDT) : \{n{:}nat \mid cts_n\ [\![t]\!]\}$.
We describe such a function via structural recursion on t.

- $t = \top$: Obviously, a possible modulus of continuity is 0, since we don't have to consider any assignments. We vacuously have $cts_0\ [\![\top]\!]$.
- $t = \bot$: Analogous to previous case.
- $t = (n, t_1, t_2)$: By recursion we have n_1 and n_2 s.t. $cts_{n_1}\ [\![t_1]\!]$ and $cts_{n_2}\ [\![t_2]\!]$. A modulus of continuity of (n, t_1, t_2) is $\eta := max\{n, n_1, n_2\} + 1$.
  By definition we have $\eta > n, \eta > n_1$ and $\eta > n_2$. We have to show $cts_\eta\ [\![(n, t_1, t_2)]\!]$.

  Proof. Let $\sigma_1, \sigma_2 : nat \to bool$ and $\mathcal{H} := \forall m : nat,\ m < \eta \to \sigma_1\ m = \sigma_2\ m$ be given. We show $[\![(n, t_1, t_2)]\!]\ \sigma_1 = [\![(n, t_1, t_2)]\!]\ \sigma_2$.
  We have $\sigma_1\ n = \sigma_2\ n$ by $\mathcal{H}$ since $n < \eta$. Case analysis:

  - $\sigma_1\ n = true = \sigma_2\ n$: $[\![t_1]\!]\ \sigma_1 = [\![t_1]\!]\ \sigma_2$ needs to be shown. Since $cts_{n_1}\ [\![t_1]\!]$ holds, it suffices to show $\forall m : nat,\ m < n_1 \to \sigma_1\ m = \sigma_2\ m$. For every such m, we have $\sigma_1\ m = \sigma_2\ m$ by $\mathcal{H}$ since $m < n_1 < \eta$.
  - $\sigma_1\ n = false = \sigma_2\ n$: $[\![t_2]\!]\ \sigma_1 = [\![t_2]\!]\ \sigma_2$ can be shown as in the previous case.

  ∎

At this point we would like to define an actual mapping from boolean functions. Since there is no way around the need to know the modulus of continuity of the argument function to build a corresponding prime tree, we have to introduce a new assumption, namely the *Axiom of Description*. With *description* it is possible to ask for the witness of a uniqueness quantification proof whenever we need it. In other words, we can transform a unique existence proof into a mapping and can thus circumvent the elim restriction. We will use the *constructive_definite_description* axiom from Coq's *Description* library. We will, however, refer to it simply as *description*. [2]

`Axiom` $description : \forall\ (A : \texttt{Type})\ (P : A \to \texttt{Prop}), (\exists!\ x : A, P\ x) \to \{x : A \mid P\ x\}$.

We will use *description* to define a mapping $BF_{cts} \to SPT$ from the proof of Corollary 5.3. Let PT_Unique denote the proof of Corollary 5.3.

`Definition` $BF_{cts}2SPT\_cert\ (\phi : BF_{cts}) : \{t : SPT \mid \phi \equiv [\![t]\!]\} :=$
$description\ SPT\ (\texttt{fun}\ t \Rightarrow \phi \equiv [\![t]\!])\ (PT\_Unique\ \phi)$.

`Definition` $BF_{cts}2SPT\ (\phi : BF_{cts}) : SPT := \pi_1\ (BF_{cts}2SPT\_cert\ \phi)$

[2] Another possibility would be to assume a *choice* principle to turn the proof of Lemma 5.6 (Denotational Completeness for prime trees) into a mapping. Since $choice \to description$ holds (uniqueness quantification is also an existential quantification), but $description \to choice$ doesn't hold in general, we show a stronger result by assuming the weaker *description* instead of *choice*.

Lemma 5.14.

1. $(BF_{cts}, \equiv)$ *is a setoid*
2. $BF_{cts}2SPT$ *is a morphism* $(BF_{cts}, \equiv) \twoheadrightarrow (SPT, =)$
3. $[\![\cdot]\!]$ *is a morphism* $(SPT, =) \twoheadrightarrow (BF_{cts}, \equiv)$

Proof.

1. Since we just forget the *cts* proof carried by members of BF_{cts}, $\equiv$ denotes normal equivalence on the actual functions, which is an equivalence relation by Lemma 5.6.
2. Let $\phi, \psi : BF_{cts}$ with $\phi \equiv \psi$. Since $BF_{cts}2SPT$ is defined through $BF_{cts}2SPT\_cert$ we have by certificate $[\![BF_{cts}2SPT\ \phi]\!] \equiv \phi \equiv \psi \equiv [\![BF_{cts}2SPT\ \psi]\!]$. Through injectivity of $[\![\cdot]\!]$ (Corollary 5.1), we have $BF_{cts}2SPT\ \phi = BF_{cts}2SPT\ \psi$.
3. Let $t_1, t_2 : SPT$ with $t_1 = t_2$. $[\![t_1]\!] \equiv [\![t_2]\!]$ immediately follows. Through $SDT\_Den\_cts\ t_1$ and $SDT\_Den\_cts\ t_2$ we obtain the modulus of continuity n and n' of $[\![t_1]\!]$ and $[\![t_2]\!]$ respectively in addition to proofs of $cts_n\ [\![t_1]\!]$ and $cts_{n'}\ [\![t_2]\!]$. Hence, $[\![t_1]\!]$ and $[\![t_2]\!]$ are continuous.

■

Theorem 5.7. *SetoidIsomorphism* $(SPT, =)\ (BF_{cts}, \equiv)\ [\![\cdot]\!]\ BF_{cts}2SPT$.

Proof.

- $\forall t : SPT,\ t = BF_{cts}2SPT\ [\![t]\!]$: By certificate of $BF_{cts}2SPT\_cert$ we have $[\![BF_{cts}2SPT\ [\![t]\!]]\!] \equiv [\![t]\!]$ and by Corollary 5.1 also $BF_{cts}2SPT\ [\![t]\!] = t$
- $\forall \phi : BF,\ \phi \equiv [\![BF_{cts}2SPT\ \phi]\!]$: Immediately by certificate of $BF_{cts}2SPT\_cert$.

■

Unfortunately assuming FE will not suffice to show that $(SPT, =)$ and $(BF_{cts}, =)$ are (partial) setoid-isomorphic. The reason is, that showing $\phi = \psi$ for $\phi, \psi : BF_{cts}$ also involves showing that the proofs of $cts\ \phi$ and $cts\ \psi$ are equal. In order to show that two proofs are equal, we, in general, need the axiom of *proof-irrelevance* (PI). Proof-irrelevance (in $Prop$) can be assumed without contradiction in Coq. It expresses that only provability matters, whatever the exact form of the proof is.

$$PI\ :=\ \forall P : Prop,\ \forall p_1\ p_2 : P \to p_1 = p_2$$

With FE and PI in our arsenal, we will be able to prove $(SPT, =)$ and $(BF_{cts}, =)$ to be setoid-isomorphic. $(SPT, =)$ and $(BF_{cts}, =)$ are obviously both setoids ($=$ is an equivalence relation). That $[\![\cdot]\!]$ and $BF_{cts}2SPT$ are still morphisms, follows by f_equal (Lemma 4.6).

Theorem 5.8. *We assume FE and PI.*
SetoidIsomorphism $(SPT, =)\ (BF_{cts}, =)\ [\![\cdot]\!]\ BF_{cts}2SPT$.

Proof.

- $\forall t : SPT,\ t = BF_{cts}2SPT\ [\![t]\!]$: Already proven in Theorem 5.7.
- $\forall \phi : BF,\ \phi = [\![BF_{cts}2SPT\ \phi]\!]$: By Theorem 5.7 we have $\forall \phi : BF,\ \phi \equiv [\![BF_{cts}2SPT\ \phi]\!]$. Through FE and PI we know that $\forall \phi : BF,\ \phi = [\![BF_{cts}2SPT\ \phi]\!]$ since the underlying functions are equal (by FE) which makes $cts\ [\![BF_{cts}2SPT\ \phi]\!]$ and $cts\ \phi$ proofs of the same proposition. PI then gives us equality of those proofs.

■

5.4.1 Consequences of assuming Description

Assume we are in a classical environment where *excluded middle* (XM) is assumed. Excluded middle is the corresponding *proposition* to the CDP function.

$$XM\ :=\ \forall P : Prop,\ P \vee \neg P$$

Note that that we also have XM by assuming PI like we did for the proof of Theorem 5.8, since $PI \to XM$ holds. If *description* is assumed in addition, it is possible to define CDP which allows us to define φ (5.1) as a function of type $(nat \to bool) \to bool$.

5.5 Version 3: Boolean functions as dependent pairs

To reach our goal in version 1, we had to assume a controversial axiom in order to find a corresponding prime tree for a boolean function. This axiom made the modulus of continuity of boolean functions accessible in every context. In version 2 the modulus of continuity was out of reach whenever we needed it for definitions instead of proofs because of the elim restriction. The question arises whether there is a type for boolean functions, which keeps the modulus of continuity available in every context without using axioms. By pulling the modulus of continuity out of the *cts* proof and making it part of the identity of the boolean function we can do so. Version 3 combines ideas from version 1 and version 2. We use *ctsT* as in version 1 and pair every boolean function with their modulus of continuity and the corresponding proof in order to restrict ourselves to only the continuous boolean functions similar to version 2. Formally, we build the dependent sum of boolean functions and their modulus of continuity.

`Definition` BF_{ctsT} : `Type` := $\{\phi : BF\ \&\ ctsT\ \phi\}$.

To make clear that the modulus of continuity is now part of the identity of boolean functions we will write (ϕ, n) for members of BF_{ctsT}.

The definition of a denotational completeness mapping is no challenge.

`Definition` $SPT\_DenCompl : \forall (\phi, n) : BF_{ctsT}, \{t : SPT \mid [\![t]\!] \equiv \phi\}$.

> ϕ is a boolean function with $cts_n\ \phi$. By the auxiliary function $SPT\_DenCompl\_aux$, we obtain a reduced decision tree t, which is logically equivalent to ϕ and ordered w.r.t. n. By Lemma 5.3, t is also ordered, which makes it a prime tree.

Note that the elim restriction is not violated since the modulus of continuity must not be extracted out of a proof.
Uniqueness of prime trees can be proven as in the previous two sections.

Corollary 5.4 (Uniqueness of prime trees). $\forall (\phi, n) : BF_{ctsT},\ \exists! t : SPT,\ \phi \equiv [\![t]\!]$.

Proof. Analogous to the proof of Corollary 5.2. ∎

As in Section 5.4 we will only compare the actual functions using equivalence:

$$(\phi, n) \equiv (\psi, m) \;:=\; \phi \equiv \psi.$$

We define

$$BF_{ctsT}2SPT \;:=\; \lambda(\phi, n).\, \pi_1\, (SPT\_DenCompl\, (\phi, n))$$

Verifying that $(BF_{ctsT}, \equiv)$ is a setoid as well as showing that $BF_{ctsT}2SPT$ and $[\![\cdot]\!]$ are morphisms between $(BF_{ctsT}, \equiv)$ and $(SPT, =)$ is just as before.

Lemma 5.15.

1. $(BF_{ctsT}, \equiv)$ *is a setoid*
2. $BF_{ctsT}2SPT$ *is a morphism* $(BF_{ctsT}, \equiv) \twoheadrightarrow (SPT, =)$
3. $[\![\cdot]\!]$ *is a morphism* $(SPT, =) \twoheadrightarrow (BF_{ctsT}, \equiv)$

Proof. All three parts are analogous to the corresponding parts of Lemma 5.14. ∎

Theorem 5.9. *SetoidIsomorphism* $(SPT, =)\ (BF_{ctsT}, \equiv)\ [\![\cdot]\!]\ BF_{ctsT}2SPT.$

Proof. Analogous to the corresponding proof (Theorem 5.7) in Section 5.4. ∎

Unfortunately, it is not possible to prove that $(SPT, =)$ and $(BF_{ctsT}, =)$ are setoid-isomorphic, even when FE and PI are assumed. The reason is, that proving $(\phi, n) = (\psi, m)$ means proving $\phi = \psi$, $n = m$ and $cts_n\ \phi = cts_m\ \psi$. While FE and PI would take care of the first and last equality respectively, $n = m$ is in general not provable. In fact, when a function is continuous, we have infinitely many moduli of continuity for that function.

Theorem 5.10. *For any* $(\phi, n) : BF_{ctsT}$*, we have* $\forall m : nat,\ m > n \to cts_m\ \phi$.

Proof. Assume $\sigma, \sigma' : nat \to bool$ as well as $\mathcal{H} := \forall k : nat, k < m \to \sigma\ k = \sigma'\ k$. We have to show $\phi\ \sigma = \phi\ \sigma'$. We use the fact that $cts_n\ \phi$ holds and thus it suffices to show $\forall l : nat, l < n \to \sigma\ l = \sigma'\ l$. For any such k we have $\sigma\ l = \sigma'\ l$ by $\mathcal{H}$ since $k < n < m$ holds. ∎

5.6 Boolean functions as *Stream bool → bool*

In this section, we want to briefly show, that there is an alternative definition of boolean functions involving *Streams*. We will mostly follow the introduction of streams as done by Chlipala [6]. Streams can be seen as unbounded lists over a type T. With this analogy in mind we want to define streams as an infinite chain of constructors, that add one element to an already present stream (*cons* operations). Coq's mechanism to realize such infinite chains of constructors is *Coinduction*. In Coq's *Stream* library one finds the following definition [2].

```
CoInductive Stream (T : Type) : Type :=
  | Cons : T → Stream T → Stream T.
```

We will use the infix notation $\gg$ for *Cons*.
Along with the *Stream* type come two definitions for extracting the first element and the remainder of a stream.

```
Definition hd {T : Type} (s : Stream T) :=
  let (t, s') := s in t.

Definition tl {T : Type} (s : Stream T) :=
  let (t, s') := s in s'.
```

Suppose we would define boolean functions using streams:

```
Definition BF' := Stream bool → bool.
```

We would like to show that BF and BF' are isomorphic. Essentially this means to prove that *Stream bool* and $nat \to bool$ are isomorphic. Before taking this challenge on, we would like to make an important observation about coinductive types: Infinite data and infinite proofs go hand in hand. This is best made clear by a small example. Consider the following corecursive procedure which applies a function f to all the members of a stream.

```
CoFixpoint map {T T' : Type} (f: T → T') (s : Stream T) : Stream T' :=
  f (hd s) ≫ map f (tl s).
```

Now consider the following two definitions of a constant *true* stream.

```
CoFixpoint falses := false ≫ falses.

CoFixpoint trues := true ≫ trues.

CoFixpoint trues' := map negb falses.
```

Proving that $trues = trues'$ holds is impossible without assumptions. The reason is that Coq's equality predicate *eq* is limited to equalities that can be proven by "finite syntactic arguments" (see Chapter 'Infinite Data and Proofs' in [6]), which is not the case when arguing about streams. We have to define an extensional equality *stream_eq* on streams which we will write as $s \stackrel{\circ}{=} t$.

```
Fixpoint get {T : Type} (s : Stream T) (n : nat) : T :=
  match n with
    | 0 ⇒ hd s
    | S n ⇒ get (tl s) n
  end.
```

get s n will from now on be written as $s[n]$.

```
Definition stream_eq {T : Type} (s t : Stream T) := ∀ n, s[n] = t[n].
```

We will use conventional equivalence $\equiv$ as an equality notion for $nat \to bool$.

The mapping $Stream\ bool \to (nat \to bool)$ is given by the *get* function.

```
Definition str2sig (s : Stream bool) : nat → bool := get s.
```

For the inverse mapping we have to put in a bit more work. We first define a stream over *nat*, which, starting at some $n : nat$, increases n with every *Cons* operation.

```
CoFixpoint ascending (n : nat) := n ≫ (ascending (S n)).
```

Using the *map* procedure and *ascending*, a mapping $(nat \to bool) \to Stream\ bool$ can look like this:

```
Definition sig2str (σ : nat→bool) : Stream bool := map σ (ascending 0).
```

We show an important property about *ascending*, *get* and *map*:

Lemma 5.16. *For any $f : nat \to bool$ and $n, m : nat$, we have: $(map\ f\ (ascending\ m))[n] = f\ (n + m)$.*

Proof. We prove this by induction on n.

- $n = 0$: We have $(map\ f\ (ascending\ m))[0] = hd\ (map\ f\ (ascending\ m)) = f\ m = f\ (0 + m)$.
- $n \to n + 1$: We have the inductive hypothesis

$$\mathcal{IH} : \forall m : nat,\ (map\ f\ (ascending\ m))[n] = f\ (n + m)$$

and wish to prove $(map\ f\ (ascending\ m))[n + 1] = f\ ((n + 1) + m)$. By commutativity and associativity of + we have $f\ ((n + 1) + m) = f\ (n + (m + 1))$ and by rewriting using $\mathcal{IH}$ from right to left we end up with $f\ (n + (m + 1)) = (map\ f\ (ascending\ (m + 1)))[n]$, which equals $(map\ f\ (ascending\ m))[n + 1]$.

■

We now have everything we need to prove that *Stream bool* and $nat \to bool$ are setoid-isomorphic.

Lemma 5.17.

1. $(Stream\ bool, \hat{=})$ *is a setoid.*
2. $(nat \to bool, \equiv)$ *is a setoid.*
3. *str2sig is a morphism* $(Stream\ bool, \hat{=}) \twoheadrightarrow (nat \to bool, \equiv)$
4. *sig2str is a morphism* $(nat \to bool, \equiv) \twoheadrightarrow (Stream\ bool, \hat{=})$

Proof.

1. We prove that $\hat{=}$ is an equivalence relation.
   - Reflexivity: $\hat{=}$ is reflexive since for any stream s and $n : nat$ we have that $s[n] = s[n]$.
   - Symmetry: $\hat{=}$ is symmetric since for any two streams s, t with $s \hat{=} t$, we have $t[n] = s[n]$ for any n through symmetry of =.
   - Transitivity: $\hat{=}$ is transitive since for any streams s, t, r with $s \hat{=} t$ and $t \hat{=} r$, we have $s[n] = r[n]$ for any n through transitivity of =.
2. Like in (1), we use that fact that = in an equivalence relation.
3. Let $s, t : Stream\ bool$ with $s \hat{=} t$. For any given $n : nat$ we then have to show $str2sig\ s\ n = str2sig\ t\ n$. This holds since $str2sig\ s\ n = s[n]$, $str2sig\ t\ n = t[n]$ and by $s \hat{=} t$.
4. Let $\sigma, \sigma' : nat \to bool$ with $\sigma \equiv \sigma'$. We need to show $sig2str\ \sigma \hat{=} sig2str\ \sigma'$. Let $n : nat$ be given.

$$\begin{aligned}
(sig2str\ \sigma)[n] &= (map\ \sigma\ (ascending\ 0))[n] \\
&= \sigma\ (n + 0) && \text{(Lemma 5.16)} \\
&= \sigma\ n \\
&= \sigma'\ n && (\sigma \equiv \sigma') \\
&= \sigma'\ (n + 0) \\
&= (map\ \sigma'\ (ascending\ 0))[n] && \text{(Lemma 5.16)} \\
&= (sig2str\ \sigma')[n]
\end{aligned}$$

■

Theorem 5.11. *SetoidIsomorphism* $(Stream\ bool, \hat{=})\ (nat \to bool, \equiv)\ str2sig\ sig2str$.

Proof.

- $\forall s,\ sig2str\ (str2sig\ s) \stackrel{\circ}{=} s$: Let $s : Stream\ bool$ and $n : nat$ be given.
  We have $(map\ (str2sig\ s)\ (ascending\ 0))[n] = str2sig\ s\ (n+0)$ by Lemma 5.16. Unfolding the definition of $str2sig$ to get yields the claim.

- $\forall \sigma,\ str2sig\ (sig2str\ \sigma) \equiv \sigma$: Let $\sigma : nat \to bool$ and $n : nat$ be given.
  We have $(map\ \sigma\ (ascending\ 0))[n] = \sigma\ (n+0)$ by Lemma 5.16. The claim follows immediately.

■

The two mappings $str2sig$ and $sig2str$ also play an important role in the mappings $BF \to BF'$ and $BF' \to BF$. Here are the definitions:

`Definition` $BF2BF'\ (\phi : BF) : BF' :=$ `fun` $s \Rightarrow \phi\ (str2sig\ s)$.

`Definition` $BF'2BF\ (\phi : BF') : BF :=$ `fun` $\sigma \Rightarrow \phi\ (sig2str\ \sigma)$.

We would like to use $\equiv$ as equality on BF and a corresponding version of $\equiv$ on BF'. We will, however, not be able to prove that BF and BF' are setoid-isomorphic using these relations. The reason is, that we are not able to prove results of the form $\phi\ (str2sig\ (sig2str\ s)) = \phi\ s$ and $\phi\ (sig2str\ (str2sig\ \sigma)) = \phi\ \sigma$. The extensional equalities used to compare boolean streams and $nat \to bool$ assignments, do not allow us to rewrite $str2sig\ (sig2str\ s)$ to s nor $sig2str\ (str2sig\ \sigma)$ to σ in this context. We, therefore, define a modified version of $\equiv$, which doesn't work on equal inputs, but on equivalent inputs.

`Definition` $BFeq\_ext\ (\phi\ \psi{:}\ BF) := \forall\ \sigma\ \sigma', \sigma \equiv \sigma' \to \phi\ \sigma = \psi\ \sigma'$.

`Definition` $BFeq\_ext'\ (\phi\ \psi{:}\ BF') := \forall\ s\ s', s \stackrel{\circ}{=} s' \to \phi\ s = \psi\ s'$.

We will use $\equiv_{ext}$ as infix notation for $BFeq\_ext$ and $\equiv'_{ext}$ for $BFeq\_ext'$.
Unfortunately $\equiv_{ext}$ and $\equiv'_{ext}$ are not provably reflexive, since neither $\sigma \equiv \sigma'$ implies $\phi\ \sigma = \phi\ \sigma'$ nor $s \stackrel{\circ}{=} s'$ implies $\phi\ s = \phi\ s'$. We have, however, accounted for exactly such circumstances in Section 3.4 when we introduced partial setoids.

Lemma 5.18.

1. $(BF, \equiv_{ext})$ *is a partial setoid.*
2. $(BF', \equiv'_{ext})$ *is a partial setoid.*
3. $BF2BF'$ *is a morphism* $(BF, \equiv_{ext}) \twoheadrightarrow (BF', \equiv'_{ext})$.
4. $BF'2BF$ *is a morphism* $(BF', \equiv'_{ext}) \twoheadrightarrow (BF, \equiv_{ext})$.

Proof.

1. We show that $\equiv_{ext}$ is a partial equivalence relation.

   - Symmetry: Let $\phi, \psi : BF$ with $\phi \equiv_{ext} \psi$ as well as $\sigma, \sigma' : nat \to bool$ with $\sigma \equiv \sigma'$ be given. We show $\psi\ \sigma = \phi\ \sigma'$. We have $\sigma' \equiv \sigma$ since $\equiv$ is symmetric (Lemma 5.17). Using this, we have $\phi\ \sigma' = \psi\ \sigma$ through $\phi \equiv_{ext} \psi$.
   - Transitivity: Let $\phi, \psi, \xi : BF$ with $\phi \equiv_{ext} \psi$, $\psi \equiv_{ext} \xi$ as well as $\sigma, \sigma' : nat \to bool$ with $\sigma \equiv \sigma'$ be given. We show $\phi\ \sigma = \xi\ \sigma'$. We have $\sigma' \equiv \sigma'$ since $\equiv$ is reflexive (Lemma 5.17). By $\phi \equiv_{ext} \psi$ and $\psi \equiv_{ext} \xi$ we thus also have $\phi\ \sigma = \psi\ \sigma'$ and $\psi\ \sigma' = \xi\ \sigma'$. By transitivity of = we conclude $\phi\ \sigma = \xi\ \sigma'$.

2. Analogous to (1) replacing $\equiv$ by $\stackrel{\circ}{=}$ and $\equiv_{ext}$ by $\equiv'_{ext}$.

3. Assume $\phi, \psi : BF$ with $\phi \equiv_{ext} \psi$ as well as $s, s' : Stream\ bool$ with $s \stackrel{\circ}{=} s'$. We need to show $BF2BF'\ \phi\ s = BF2BF'\ \psi\ s'$, which, by definition, corresponds to $\phi\ (str2sig\ s) = \psi\ (str2sig\ s')$. By using $\phi \equiv_{ext} \psi$, we have this, if we can show $(str2sig\ s) \equiv (str2sig\ s')$. By definition this is the same as showing $\forall n : nat,\ s[n] = s'[n]$ which we already have with $s \stackrel{\circ}{=} s'$.

4. Assume $\phi, \psi : BF'$ with $\phi \equiv'_{ext} \psi$ as well as $\sigma, \sigma' : nat \to bool$ with $\sigma \equiv \sigma'$. We need to show $BF'2BF\ \phi\ \sigma = BF'2BF\ \psi\ \sigma'$, which, by definition, corresponds to $\phi\ (sig2str\ \sigma) = \psi\ (sig2str\ \sigma')$. By using $\phi \equiv'_{ext} \psi$, we have this, if we can show $(sig2str\ \sigma) \stackrel{\circ}{=} (sig2str\ \sigma')$ which is the same as $\forall n,\ (map\ \sigma\ (ascending\ 0))[n] = (map\ \sigma'\ (ascending\ 0))[n]$. Through Lemma 5.16 we can rewrite this to $\forall n,\ \sigma\ (n + 0) = \sigma'\ (n + 0)$, which we can show through $\sigma \equiv \sigma'$.

■

Theorem 5.12. *PartialSetoidIsomorphism* $(BF', \equiv'_{ext})\ (BF, \equiv_{ext})$ *BF'2BF BF2BF.*

Proof.

- $\forall \phi,\ \phi \equiv'_{ext} \phi \to BF2BF'\ (BF'2BF\ \phi) \equiv'_{ext} \phi$: We assume some $\phi : BF'$ with $\phi \equiv'_{ext} \phi$ as well as $s, s' : Stream\ bool$ with $s \stackrel{\circ}{=} s'$ and wish to show $BF2BF'\ (BF'2BF\ \phi)\ s = \phi\ s'$. By definition this is the same as $\phi\ (sig2str\ (str2sig\ s)) = \phi\ s'$. Since $str2sig$ and $sig2str$ form a setoid-isomorphism between $(Stream\ bool, \stackrel{\circ}{=})$ and $(nat \to bool, \equiv)$ we have $sig2str\ (str2sig\ s) \stackrel{\circ}{=} s$. Using transitivity of $\stackrel{\circ}{=}$ (Lemma 5.17), we also have $sig2str\ (str2sig\ s) \stackrel{\circ}{=} s'$ because of $s \stackrel{\circ}{=} s'$. The claim then follows via $\phi \equiv'_{ext} \phi$.
- $\forall \phi,\ \phi \equiv_{ext} \phi \to BF'2BF\ (BF2BF'\ \phi) \equiv_{ext} \phi$: Analogous to the previous case.

■

Next, we would like to build setoids $(Stream\ bool, =)$ and $(nat \to bool, =)$ as well as $(BF, =)$ and $(BF', =)$ and prove the corresponding isomorphisms. Note that all the candidates are in fact setoids since $=$ is an equivalence relation. Also $str2sig$, $sig2str$, $BF2BF'$ and $BF'2BF$ are still morphisms between their respective setoids, which follows from f_equal (Lemma 4.6). For the first setoid-isomorphism between $(Stream\ bool, =)$ and $(nat \to bool, =)$ we have to assume *stream extensionality* and *sigma extensionality*, the latter being a special case of FE.

$$\begin{aligned} Str\_Ext &:= \forall s_1\, s_2 : Stream\ bool,\ s_1 \stackrel{\circ}{=} s_2 \to s_1 = s_2. \\ Sig\_Ext &:= \forall \sigma_1\, \sigma_2 : nat \to bool,\ \sigma_1 \equiv \sigma_2 \to \sigma_1 = \sigma_2. \end{aligned}$$

Theorem 5.13. *We assume Str_Ext and Sig_Ext.*
SetoidIsomorphism $(Stream\ bool, =)\ (nat \to bool, =)$ $str2sig\ sig2str$.

Proof.

- $\forall s,\ sig2str\ (str2sig\ s) = s$: We have $\forall s,\ sig2str\ (str2sig\ s) \stackrel{\circ}{=} s$ by Theorem 5.11. Str_Ext then gives us the claim.
- $\forall \sigma,\ str2sig\ (sig2str\ \sigma) = \sigma$: We have $\forall \sigma,\ str2sig\ (sig2str\ \sigma) \equiv \sigma$ by Theorem 5.11. Sig_Ext then gives us the claim.

■

For the isomorphism between $(BF, =)$ and $(BF', =)$, assuming Sig_Ext and Str_Ext are not strong enough. We will need FE and Str_Ext. Since FE obviously implies Sig_Ext, we are still able to use the previous theorem. Using $=$ as equality on BF and BF', enables us to prove a *SetoidIsomorphism* between $(BF, =)$ and $(BF', =)$ instead of just a *PartialSetoidIsomorphism* as before.

Theorem 5.14. *We assume FE and Str_Ext.*
SetoidIsomorphism $(BF', =)$ $(BF, =)$ *BF'2BF BF2BF.*

Proof. We have

$$\begin{aligned} \mathcal{H}_1 &:= \forall s,\ sig2str\ (str2sig\ s) = s \\ \mathcal{H}_2 &:= \forall \sigma,\ str2sig\ (sig2str\ \sigma) = \sigma \end{aligned}$$

through (Theorem 5.13).

- $\forall \phi,\ BF2BF'\ (BF'2BF\ \phi) = \phi$: By unfolding the definitions of $BF2BF'$ and $BF'2BF$ we have to prove $(fun\ s : Stream\ bool \Rightarrow \phi\ (sig2str\ (str2sig\ s))) = \phi$. Using FE it is enough to prove $\phi\ (sig2str\ (str2sig\ s)) = \phi\ s$. By $\mathcal{H}_1$ we have this.
- $\forall \phi,\ BF'2BF\ (BF2BF'\ \phi) = \phi$: By unfolding the definitions of $BF2BF'$ and $BF'2BF$ we have to prove $(fun\ \sigma : nat \to bool \Rightarrow \phi\ (str2sig\ (sig2str\ \sigma))) = \phi$. Using FE it is enough to prove $\phi\ (str2sig\ (sig2str\ \sigma)) = \phi\ \sigma$. By $\mathcal{H}_2$ we have this.

■

Chapter 6

Conclusion and Future Work

6.1 The isomorphisms

- $(\Sigma_n\ bool^n \to bool, \equiv_\Sigma)$ and $(\Sigma_n\ PT\ n, =)$

  In Chapter 4 we showed that $\Sigma_n\ bool^n \to bool$ and $\Sigma_n\ PT\ n$ are setoid-isomorphic. We quickly discovered that modelling decision trees as a dependent inductive type $DT : nat \to Type$ required a complicated inversion function, which did a level dependent case analysis on the given decision tree. This showcased the lack of support for such dependent types in Coq, as tactics like *inversion* or *injection* failed to deliver.

  - $(DT\ n, =)$ and $(DT_{rec}\ n, =)$
    A recursive type for decision trees $DT_{rec} : nat \to Type$ was clearly the better approach compared to the dependent inductive one. No inversion function similar to DT_Inv was required as Coq's usual case analysis did the job. The setoid-isomorphism itself was straightforward to show. The idea to come up with a recursive type for decision trees was inspired by Adams [1].

  By design, every decision tree was already ordered, such that only reducing was necessary to define prime trees. The draw-back of the types $\Sigma_n\ bool^n \to bool$ and $\Sigma_n\ PT\ n$ is that we have several copies of the same function and the same prime tree, as both $\lambda\_\_\_.\ true$ and $\lambda\_.\ true$ are constant *true* functions as well as both $\top^2$ and $\top^1$ are constant *true* prime trees. For the setoid-isomorphism between $(\Sigma_n\ bool^n \to bool, \equiv)$ and $(\Sigma_n\ PT\ n, =)$ no axioms were required. When we changed $\equiv$ to $=$, however, we needed FE as expected.

- $(BF, \equiv)$ and $(SPT, =)$

  In Chapter 5, we defined boolean functions as $BF := (nat \to bool) \to bool$. Decision trees were defined by a simple inductive type $SDT : Type$ without dependency. It turns out that without additional assumptions, we are unable to define a useful mapping (denotational completeness for prime trees) from boolean functions to simple prime trees. A first solution was to assume that every boolean function is continuous such that we obtain a modulus of continuity of the function in question. Assuming the *Axiom of Continuity*, while drastic, seemed plausible since it appeared impossible to define non-continuous members of $(nat \to bool) \to bool$. Unfortunately CTS is inconsistent with other axioms, like CDP, that allow us to migrate members of $(nat \to bool) \to Prop$ to $(nat \to bool) \to bool$, like described in Section 5.3.1.
  Approaching the end of this development, we had to put in more work as for the previous isomorphism involving cascaded functions. This holds especially for the proof of $\forall t_1\ t_2\ :$

$SPT,\ t_1 \neq t_2 \to [\![t_1]\!] \not\equiv [\![t_2]\!]$ (Theorem 5.2). This is mainly due to the fact that simple decision trees were not ordered by definition and that the branching constructor explicitly indicated the variable on which it branches. A few additional lemmas relating orderedness and variable occurrences in a tree were required to establish Theorem 5.2. For a setoid-isomorphism between $(BF, =)$ and $(SPT, =)$, we had to assume FE in addition.

- $(BF_{cts}, \equiv)$ and $(SPT, =)$

  In this version we tried to replace CTS by *description*. The *Axiom of Description* makes it possible to turn functional relations into functions. We used this fact to obtain a mapping from continuous boolean functions to prime trees by turning the proof of $\exists! t : SPT,\ [\![t]\!] \equiv \phi$ into a mapping. The rest of the development was similar to the previous one, since many of the results could be reused or easily adapted.
  Major differences to the previous isomorphism are, that we have to prove continuity of denotations of decision trees (by computing a modulus of continuity) and that PI has to be assumed in addition to FE to prove a setoid-isomorphism between $(BF_{cts}, =)$ and $(SPT, =)$.

- $(BF_{ctsT}, \equiv)$ and $(SPT, =)$

  Having identified that the modulus of continuity has to be available in order to avoid the assumption of axioms, we made the necessary changes to the type for boolean functions by making the modulus of continuity part of the identity of the function. Almost every result from the previous two developments could be reused or easily adapted. Unfortunately an isomorphism using the setoid $(BF_{ctsT}, =)$ instead of $(BF_{ctsT}, \equiv)$ cannot be proven since the modulus of continuity is not unique.
  No axioms were necessary for this development.

6.2 Possible improvements

- By pairing up every boolean function with its smallest possible modulus of continuity, thereby making it unique, we could prove that $(BF_{ctsT}, =)$ and $(SPT, =)$ are isomorphic assuming FE and PI.

- Theorem 5.1 already hinted that with our notion of isomorphism using setoids it might be possible to establish isomorphisms that are not meaning preserving. Since we where looking for representatives of boolean functions and we possibly want to do computations on them we also want them to behave the same as the boolean function they represent. Therefore we wish to define a type of representatives of boolean functions, whatever their definition (we will denote boolean functions with $\mathcal{BF}$).
  We extend a setoid by adding a denotation for the carrier type T and an additional proof component saying that if two members of T are equal, then they should have the same denotation.

$$Booloid := \left\{ \begin{array}{lcl} T & : & Type \\ =_T & : & T \to T \to Prop \\ [\![\cdot]\!] & : & T \to \mathcal{BF} \\ ER & : & =_T \text{ is (partial) equivalence relation} \\ P & : & \forall t_1\, t_2 : T,\ t_1 =_T t_2 \to [\![t_1]\!] \equiv_{\mathcal{BF}} [\![t_2]\!] \end{array} \right\}$$

  A morphism between *Booloids* (denoted by $\twoheadrightarrow_{\mathcal{B}}$) would then be a mapping that is meaning

preserving and equality preserving:

$$(T, =_T, [\![\cdot]\!]_T) \twoheadrightarrow_{\mathcal{B}} (T', =_{T'}, [\![\cdot]\!]_{T'}) := \left\{ \begin{array}{lcl} \varrho & : & T \to T' \\ EP & : & \forall t_1\, t_2 : T,\ t_1 =_T t_2 \to \varrho\ t_1 =_{T'} \varrho\ t_2 \\ MP & : & \forall t : T,\ [\![t]\!]_T \equiv_{\mathcal{BF}} [\![\varrho\ t]\!]_{T'} \end{array} \right\}$$

An isomorphism between *Booloids* consists of two *Booloid*-morphisms.
It is, in fact, easy to see that all the setoids mentioned in the previous section (Section 6.1) are also *Booloids* and that all the morphisms are meaning preserving.
This makes $(\Sigma_n\ PT\ n, =, \lambda(n,t).\ [\![t]\!])$ and $(SPT, =, [\![\cdot]\!])$ legitimate representatives of their respective boolean functions, namely $(\Sigma_n\ bool^n \to bool, \equiv_\Sigma, id)$ and, for example $(BF_{cts}, \equiv, id)$.

Bibliography

[1] Robin Adams. Formalized metatheory with terms represented by an indexed family of types. In Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner, editors, *Types for Proofs and Programs*, volume 3839 of *Lecture Notes in Computer Science*, pages 1–16. Springer Berlin / Heidelberg, 2006.

[2] The Coq Proof Assistant. Standard library. `http://coq.inria.fr/stdlib/`.

[3] Gilles Barthe, Venanzio Capretta, and Olivier Pons. Setoids in type theory, 2000.

[4] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004. `http://www.labri.fr/publications/l3a/2004/BC04`.

[5] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[6] Adam Chlipala. Certified programming with dependent types. `http://adam.chlipala.net/cpdt/`.

[7] Gert Smolka and Chad E. Brown. Introduction to computational logic lecture notes, 2009.