CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Formalizing Constructive Quantifier Elimination in Agda

Master's thesis in Computer Science

Jeremy Pope

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2018

Master's thesis 2018

Formalizing Constructive Quantifier Elimination in Agda

Jeremy Pope

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2018

Formalizing Constructive Quantifier Elimination in Agda
Jeremy Pope

Supervisors: Thierry Coquand and Simon Huber, Department of Computer Science and Engineering
Examiner: Andreas Abel, Department of Computer Science and Engineering

Master's Thesis 2018
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2018

Formalizing Constructive Quantifier Elimination in Agda
Jeremy Pope
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

In this thesis a constructive formalization of quantifier elimination is presented, based on a classical formalization by Tobias Nipkow [16]. The formalization is implemented and verified in the programming language/proof assistant Agda [1]. It is shown that, as in the classical case, the ability to eliminate a single existential quantifier may be generalized to full quantifier elimination and consequently a decision procedure. The latter is shown to have strong properties under a constructive metatheory, such as the generation of witnesses and counterexamples. Finally, this is demonstrated on a minimal theory on the natural numbers.

Keywords: Agda, decidability, semantics, constructive, successor.

Acknowledgements

I would like to thank my supervisors Thierry Coquand and Simon Huber for the idea of the project, and for helping me to understand constructive logic, semantics, and how to navigate the distinction between theory and metatheory.

I would also like to thank my family for their love and support, and the opportunity to study in Göteborg and work on this project.

Jeremy Pope, Gothenburg, November 2017

Contents

Chapter 1

Introduction

1.1 Predicate Logic and Quantifier Elimination

A proposition in predicate logic is formed in one of three ways: from an atom, by linking propositions together using a connective (such as $\vee$ or $\Rightarrow$), or by quantifying a proposition with $\forall$ or $\exists$. Neglecting the internal structure of atoms, it is the third that sets predicate logic apart from propositional logic; the quantifiers greatly enhance the expressiveness of the language.

A drawback is that the truth of a proposition is no longer easy to determine. With propositional logic an exhaustive enumeration is possible, but this is not so in predicate logic: to do so on a proposition such as $\forall x.(x \neq x + 1)$ would require verifying $x \neq x + 1$ for every possible value of x, which—depending on our choice of domain—could be infinite.

There is not always a way around this; predicate logic is indeed undecidable in the general case. However, a number of specific theories within predicate logic are in fact decidable—and not simply by admitting exhaustive enumeration. Rather, decidability is shown through *quantifier elimination*.

The idea behind quantifier elimination is to devise a method to transform any given proposition into an equivalent one without quantifiers. The latter can typically be decided very easily, and by virtue of the equivalence the decision applies to the original proposition as well. This allows any proposition in the theory to be decided, rendering the theory decidable.

1.2 Formalization

The difficulty of quantifier elimination depends on the theory in question, but even for simple theories it is quite high—great care must be taken to ensure that the method is sound. Moreover, applying the procedure to a complicated proposition is likely impractical for a human (especially if it involves conversion to disjunctive normal form, which can result in a large growth in the number of terms). These factors encourage computer formalization of quantifier elimination—both in implementing the procedures, and verifying that they are correct.

Implementation by itself is conceptually straightforward: propositions are represented by some datatype, and quantifier elimination as procedure(s) that manipulate

objects of that datatype. Verification makes matters more complicated; the implementation must be accompanied by a proof of its correctness, which certifies that the quantifier elimination procedure always produces a proposition that is equivalent to the input (and quantifier-free). To facilitate this, both the implementation and correctness proof are typically written in a proof assistant.

1.3 History of Quantifier Elimination

According to Doner and Hodges [10] the technique of quantifier elimination was first shown by Leopold Löwenheim in 1915, and developed further by Thoralf Skolem in 1919. It was then used by Cooper Harold Langford in 1927 to prove the decidability of (some variations on) dense linear orders, a result built upon shortly thereafter by Tarski.

In 1930, Tarski discovered a decision procedure for real closed fields (fields that are elementarily equivalent to $\mathbb{R}$), by way of quantifier elimination [10]. This theory is also modeled by Tarski's axiomatization of Euclidean geometry, thereby proving the decidability of the latter [21]. Around the same time, Tarski also found a decision procedure for algebraically closed fields (fields for which every non-constant polynomial has a root) [10].

In the world of discrete numbers, Mojżesz Presburger (a student of Tarski) used quantifier elimination in his 1930 Master's thesis [20] to prove the completeness and decidability of *Presburger arithmetic*, a theory of addition on the natural numbers. That same year Jacques Herbrand [12] employed the technique to prove that several simpler theories on the natural numbers are consistent, complete, and decidable under a constructive metatheory.

Following Gödel's incompleteness theorems [11] came several negative results in decidability. Church [6] and Turing [24] independently showed in 1936 that Hilbert's *Entscheidungsproblem* is unsolvable; there are undecidable problems in first-order arithmetic. In 1949, it was proven that Robinson arithmetic [18] (addition and multiplication on the natural numbers) is moreover *essentially undecidable*—any system that can even interpret it is undecidable. These results impose very strong limitations on the decidability of theories on the natural numbers, ruling out the possibility of quantifier elimination on general arithmetic.

The first known use of computers for quantifier elimination (and theorem proving in general), according to Stansifer [20], is a program written in 1954 by Martin Davis to "prove theorems of Presburger arithmetic". Since then there have been a number of computer implementations of quantifier elimination, with and without verification. Two examples relevant to this project are the formalizations of quantifier elimination by Tobias Nipkow [16] (for several theories) and Guillaume Allais [3] (for Presburger arithmetic), discussed further in the following section. There have also been formalizations of Tarski's famous results, notably those carried out in the proof assistant Coq [8] by Assia Mahboubi and Cyril Cohen for algebraically closed fields [15] and real closed fields [7].

1.4 Relevant Work

As mentioned previously, quantifier elimination under a constructive metatheory was explored by Jacques Herbrand in his 1930 doctoral thesis [12], in which he proved the consistency, completeness, and decidability of various (classical) theories on the natural numbers. While much of the paper is not amenable to formalization,[1] a technique specific to one theory serves as the basis of the method used in Chapter 4.

Nipkow's formalization [16], in which he implements and verifies quantifier elimination on a number of theories, serves as the primary basis of this project. The formalization is quite general—the core is entirely theory-independent, and is simply instantiated for each of the specific theories. It differs significantly from the one presented here in that it is carried out in the proof assistant Isabelle [14], under a classical metatheory.

Correspondence with Guillaume Allais [3] revealed that he had in fact developed a constructive formalization of quantifier elimination in Agda, specifically on Presburger arithmetic (a small part of which available online [4]). While this would have served as a good basis for this thesis, it was not discovered until after the project had been mostly completed. With respect to Allais' work, this thesis hopes to contribute primarily through generality: constructive quantifier elimination is explored largely irrespective to the theory under consideration, and as a result is applicable to many theories in first-order logic.

1.5 Organization

The remainder of this thesis is organized as follows: First, background information is given on theoretical aspects relevant to the project (Chapter 2). Next, a theory-independent formalization of quantifier elimination is shown (Chapter 3), followed by an application to a theory on the natural numbers (Chapter 4). Finally, the project's results and possible improvements are discussed (Chapter 5).

The source code for the project (excluding the Agda standard library) is available on GitHub[2].

[1] The treatment of variables in particular poses challenges.

[2] https://github.com/guspopje/agda-qelim

Chapter 2

Theoretical Background

2.1 Quantifier Elimination

Rather than attempting to remove all quantifiers at once, an incremental approach can be taken, dramatically reducing the scope of the problem. A procedure is devised to remove a single quantifier, often $\exists$, from an otherwise quantifier-free proposition:

$$\exists x.\phi \iff \psi,$$

where ϕ and ψ are quantifier-free. Using the quantifier duality $\forall x.\phi \iff \neg\exists x.\neg\phi$ (in a classical theory) this can be adapted to remove $\forall$ as well. If the full proposition in question (which may contain many quantifiers) is placed into *prenex form*, where all of its quantifiers are pushed as far out as possible, then repeated application of the single-step procedure can clearly be used to eliminate all quantifiers from the "inside out":

$$\exists z.\forall y.\exists x.\phi \iff \exists z.\forall y.\rho \iff \exists z.\sigma \iff \psi,$$

noting that ϕ, ρ, σ, and ψ are all quantifier-free. The same recursive strategy can just as well be used without placing the proposition into prenex form, at the cost of being less clearly inductive.

To narrow the problem even further, the quantifier-free sub-proposition ϕ can be placed into disjunctive normal form (DNF):

$$\phi \iff C_1 \vee C_2 \vee \ldots \vee C_n$$

where each C_i is a conjunction of literals (a literal being an atomic formula or its negation). This is useful because existential quantification distributes across disjunction:

$$\exists x.\phi \iff \exists x.(C_1 \vee C_2 \vee \ldots \vee C_n) \iff (\exists x.C_1) \vee (\exists x.C_2) \vee \ldots \vee (\exists x.C_n).$$

As a result, elimination can be carried out on each conjunction separately, reducing the problem to quantifier elimination on conjunctions of literals.

Once a quantifier elimination procedure has been shown, decidability of the theory is obtained—provided that all quantifier-free propositions are decidable. The latter requirement is trivially true for theories where atomic formulae represent decidable relations (e.g. equality on the natural numbers), such as the theory to be presented in Chapter 4.

Proposition	**Proof**
$A \wedge B$	A proof of A and a proof of B.
$A \vee B$	Either a proof of A or a proof of B.
$A \to B$	A way of transforming a proof of A into a proof of B.
$\neg A$	A way of transforming a proof of A into a proof of $\bot$ ($\neg A$ is shorthand for $A \to \bot$).
$\bot$	(No proof.)
$\exists x.A(x)$	An object e and a proof of $A(e)$.
$\forall x.A(x)$	A way to, given any object e in the domain of quantification, produce a proof of $A(e)$.

Table 2.1: The BHK interpretation.

2.2 Constructive Logic

Constructivism is based on the idea that existence (and truth) is shown by construction [23]. In other words, a proof of existence must provide a means to construct such an object, as opposed to merely proving that its non-existence is impossible.

Constructive/intuitionistic logic reflects this by, in relation to classical logic, rejecting the law of excluded middle ($A \vee \neg A$) or equivalent. This eliminates proof by contradiction, as well as quantifier dualities such as $\exists x.A(x) \iff \neg\forall x.\neg A(x)$ which would yield non-constructive proofs.[1]

To illustrate this, the following is a canonical example of a *non*-constructive proof, attributed to Dov Jarden [13]:

Claim: There exist two irrational numbers a and b such that a^b is rational.

Proof: Consider $\sqrt{2}^{\sqrt{2}}$. If it is rational, then let $a = b = \sqrt{2}$. If it is irrational, then the choice of $a = \sqrt{2}^{\sqrt{2}}$, $b = \sqrt{2}$ is satisfactory, since $(\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = (\sqrt{2})^2 = 2$, which is rational. Either way, the desired result is proven.

This proof is non-constructive because without further knowledge it does not provide a means to construct a specific pair (a, b)—it is not possible *a priori* to determine which of the two cases holds.

The Brouwer-Heyting-Kolmogorov (BHK) interpretation gives an informal definition of what *does* constitute a constructive proof, defined recursively on the structure of the proposition in question. Table 2.1 shows the cases of the BHK interpretation (adapted from Troelstra [23] and Bridges [5]).

Two cases that stand out are disjunction and existential quantification. Disjunction differs from its classical counterpart in that a constructive proof of $A \vee B$ must

[1] For that particular duality, the left-to-right direction is constructively provable, but the right-to-left direction is not.

Propositions	Types
Conjunction ($\wedge$)	(Cartesian) product type ($\times$)
Disjunction ($\vee$)	Sum type (disjoint union, $\uplus$)
Implication ($\rightarrow$)	Function type ($\rightarrow$)
Absurdity ($\bot$)	Empty type ($\bot$)
Existential quantification ($\exists$)	Dependent sum (pair) type (Σ)
Universal quantification ($\forall$)	Dependent product (function) type (Π)

Table 2.2: Correspondence between propositions and types.

be a proof of one of the two, implying that the disjunction is decidable. A proof of existence, as discussed previously, must provide (construct) a *witness*—an object for which the quantified proposition holds. The latter in particular has strong implications in the context of quantifier elimination: if a decision procedure produces a constructive proof that $\exists x.\phi(x)$ is true, that proof comes with such a value for x.

2.3 Propositions as Types

Proofs in intuitionistic logic correspond to well-typed terms of typed lambda calculus [23], a result known as the Curry-Howard correspondence. Martin Löf's Intuitionistic Type Theory (ITT) [17] takes this a step further, providing a system which may be seen as both an intuitionistic logic and a programming language. What is viewed as a proposition in the former perspective is viewed as a type in the latter.

The correspondence, given in Table 2.2, follows the BHK interpretation described in the previous section; if a type T corresponds to a proposition P, then the objects of T precisely match the informal definition of proofs for P.

The programming language/proof assistant Agda, based on intuitionistic type theory, makes it possible to write proofs as programs and verify them by way of type-checking: a well-typed program corresponds to a valid proof.

As it would be virtually impossible to give a sufficient introduction to the language itself in this thesis, readers unfamiliar with it are instead referred to the collection of tutorials linked to on the Agda Wiki webpage [2].

2.4 Theory, Metatheory, and Semantics

In quantifier elimination, and proofs about logic systems in general, there are often two "layers": the theory T under consideration, expressed in the *object language*, and the metatheory M in which T is analyzed, expressed in the *metalanguage*.

The notions of equivalance and decidability (as related to quantifier elimination) necessitate that a notion of provability or truth be associated with T. One option is to define a proof system directly for T, as in Herbrand's thesis [12]. This allows

a syntactic treatment, notions such as "equivalent in T" and "provable in T", and consequently strong separation between theory and metatheory.[2]

Another option, as taken by Tarski [22], Nipkow [16], and this project, is to instead define the semantics (or interpretation) of propositions of T, *as propositions in* M:

$$[\![\cdot]\!] : T \to M.$$

This is typically accomplished recursively, mapping each each connective or quantifier in T to the corresponding one in M. In the case of this project, since M is ITT, the correspondence given in Table 2.2 is used. A consequence is that since M is constructive, and disjunction in T is mapped to disjunction in M, the semantics of T are constructive as well.

With this approach, quantifier elimination produces a proposition that is *semantically* equivalent to the original, and in the end it is the *semantics* of T that are proven to be decidable (as opposed to T itself, which is not possible without a proof system). As the semantics lie in M, this means that decidability is shown for a fragment of M.

In the context of this project, the object language is a datatype `Prop`, representing propositions in predicate logic, and the metalanguage is the entirety of Agda. The semantics of `Prop` are propositions in Agda, so roughly:

$$[\![\cdot]\!] : \texttt{Prop}\ \textit{objects} \to \textit{Agda propositions}.$$

Semantic equivalence on `Prop` is therefore logical equivalence on the corresponding Agda propositions, and decidability of the semantics of a `Prop` is decidability of the Agda proposition it represents. `Prop` is thus said to *code* for (a subset of) propositions in Agda.

2.4.1 Reflection

One interesting benefit of the semantic approach is that if an inverse of $[\![\cdot]\!]$ can be provided—only possible with help from the "meta-metatheory"—then M may effectively operate on a fragment of itself, a process referred to as *reflection*. Reflection allows (among other things) a suitable proposition in M to be decided without manually converting it to its representation in T; the representation is rather derived from the quoted (reflected) structure of the proposition. In doing so, the need for the user to be familiar with the underlying representation (T) is removed, resulting in simpler invocations of the decision procedure(s).

While reflection is not implemented in this project, it is in Nipkow's work [16], and a great deal of information on the reflection mechanisms in Agda and their uses is available in Paul van der Walt's Master's thesis [25].

Similar (but non-reflective) metaprogramming is used to provide *tactics*, a powerful tool in interactive theorem proving. One such example is the `omega` [9] tactic in Coq, which solves propositions of Presburger arithmetic.

[2] In Herbrand's case, this allows the analysis of a classical theory under a constructive metatheory.

2.5 De Bruijn Indices

While the general structure of propositions—atoms, connectives, and quantifiers—is readily formalizable, a number of concerns arise surrounding variables.

Each variable in a proposition is said to be either *bound* or *free*. Bound variables are ones that reference a quantifier in the proposition, and free variables are ones that do not. For example, in $(\exists x.x = y + 1) \wedge x = 6$, the first x is bound, while y and the second x are free.[3] This property depends on context; if only the (sub-)proposition $y = x + 1$ is considered, then both variables are considered free.

With that in mind, we turn our attention to how variables themselves are represented. The typical and most readable approach is to give variables names, such as x or y. As it turns out, this introduces considerable difficulty in the context of formalized systems, even the names are chosen from an amenable set such as the natural numbers.

De Bruijn indices are an alternative way to refer to variables in a proposition, without using names (in the usual sense). The idea is that each reference to a variable is replaced with a natural number indicating how many binders—in this case, quantifiers—are between it and the quantifier to which it refers. The proposition

$$\forall x.(x \leq 4 \vee (\exists y.x = y + 5))$$

is therefore represented using de Bruijn indices as

$$\forall.(\boxed{0} \leq 4 \vee (\exists.\boxed{1} = \boxed{0} + 5)).$$

The first $\boxed{0}$ indicates skipping zero quantifiers, and thus refers to the variable associated with the $\forall$. The subsequent $\boxed{1}$ indicates that one quantifier, the $\exists$, should be skipped, and likewise refers to the $\forall$. The last index, $\boxed{0}$, indicates to skip zero quantifiers, and therefore refers to the $\exists$.

Free variables are ones where the de Bruijn index extends past the head of the proposition. In the example above, if the scope is narrowed to

$$\boxed{0} \leq 4 \vee (\exists.\boxed{1} = \boxed{0} + 5),$$

then the first $\boxed{0}$ and the single $\boxed{1}$ are considered free.

While the appeal of "one variable, one name" is lost, several advantages are found. First, there is no longer any concern about the naming of bound variables. For example, the equivalent propositions $\exists y.x + 1 = y$ and $\exists z.x + 1 = z$ are encoded identically; there is no need to convert between one and the other.

Second, it is much easier to avoid the issue of *free variable capture*, where a clash between free and bound variables during a substitution produces "unexpected" results. Consider the proposition $\forall y.\exists x.x \neq y$—if it is applied to z, one obtains $\exists x.x \neq z$, as expected. However, if it is instead applied to x, the result is $\exists x.x \neq x$. This occurs because the existential quantifier has "captured" the substituted x. With named variables the solution is to restrict the rules of substitution so that such substitutions may not be made, which complicates manipulation considerably.

[3] Note that the two instances of x in fact refer to different variables.

If de Bruijn indices are used, however, all that needs to be done to ensure a safe substitution is to adjust the indices in the substituted expression appropriately (to take into account being inside more/fewer quantifiers). The latter therefore results in a cleaner and simpler treatment of substitution.

A third benefit of de Bruijn indices relates to free variables and their association with values. If one considers a proposition ϕ with free variables x, y, and z, it is clear that the semantics of ϕ is a function of those variables. On the other hand, the semantics of $\exists x.\phi$ is only a function of y and z. This dependence on the values of free variables necessitates an *environment*; a mapping from variables to values. With named variables, this means an association between variable names and values, such as a list of (*name*, *value*) pairs. With de Bruijn indices the environment may merely be a list of values—element i in the list corresponds to the variable with de Bruijn index i. As will be seen in Section 3.3, this also leads to an elegant definition of the semantics of $\exists$ and $\forall$.

An immediate concern with the use of a list of values as an environment is that the list is of sufficient length.[4] To ensure that this is the case, a notion of "arity" is introduced for propositions:

- The arity of an atom is any natural number n such that all de Bruijn indices in the atom are less than n.
- $\bot$ may have any arity.
- $\wedge$, $\vee$, and $\Rightarrow$ join two propositions with the same arity n to produce a proposition with arity n.
- Quantifiers $\exists$ and $\forall$, when applied to a proposition of arity $n + 1$, produce a proposition of arity n.

Defined as above, the arity of a proposition provides an upper bound on the necessary length of the environment to guarantee that each free variable is associated with a value. It is consequently also an upper bound on the number of distinct free variables in the proposition.

[4] The analogous concern with named variables is ensuring that every variable used in the proposition is defined in the environment.

Chapter 3

Theory-Independent Work

3.1 Atoms

In the interest of generality, atomic formulae are not represented by a fixed type, but by a type given as a *module parameter*. The type is indexed by a natural numbers n representing its arity (an upper bound on its free variables, as discussed in Section 2.5):

```
Atom : ℕ → Set
```

The internal structure of an `Atom` is completely unspecified.

The semantics of `Atom` are also given by way of module parameters. First, the set of values which variables may take:

```
Y : Set
```

Then, a function which gives the semantics for an atom:

```
⟦_⟧ₐ : {n : ℕ} → Atom n → Vec Y n → Set
```

The implicit parameter `n : ℕ` is the arity of the atom, and the following parameter of type `Atom n` is the atom itself. The last parameter, of type `Vec Y n`, is the *environment*: a list ("vector") of length `n` of values for the free variables in the atom (see Section 2.5). This, in a sense, forces `Atom` to use de Bruijn indices internally—no names are associated with the values in the enviroment. Moreover, since the enviroment for an `Atom n` is a list of `n` values, the effective arity of the atom is restricted to `n`, as intended.

Additionally, it is required that the semantics of `Atom` be decidable under any given enviroment. This is often the case (as discussed in Section 2.1, and is equivalent to the semantics of all quantifier-free propositions being decidable. As it turns out, for a constructive theory this is important not only for decidability but for quantifier elimination itself, as will be seen later on. Another module parameter is used to implement this requirement:

```
⟦_⟧ₐ? : {n : ℕ} (a : Atom n) (e : Vec Y n) → Dec (⟦ a ⟧ₐ e)
```

The `Dec` type family, from Agda's standard library, is indexed by a type (`A`). An object of type `Dec A` is a decision for `A`: either a proof that `A` is inhabited (`yes a`, where `a : A`), or a proof that it is not (`no x`, where `x : ¬ A`, i.e. `x : A → ⊥`).

For organizational purposes the above are grouped into a record type, forming an abstract representation of atoms with decidable semantics:

```
record DecAtom : Set₁ where
  field
    Atom : ℕ → Set
    Y : Set
    ⟦_⟧ₐ : {n : ℕ} → Atom n → Vec Y n → Set
    ⟦_⟧ₐ? : {n : ℕ} (a : Atom n) (e : Vec Y n) → Dec (⟦ a ⟧ₐ e)
```

A single module parameter of type `DecAtom` is used in lieu of four separate parameters.

3.2 Representation of Propositions

Propositions are represented by following datatype. Constructors allow the formation of a proposition from an atom, or from other propositions by way of the typical connectives and quantifiers.[1]

```
data Prop (n : ℕ) : Set where
  atom : Atom n → Prop n
  ⊥⊥   : Prop n
  _∪_  : Prop n → Prop n → Prop n
  _&_  : Prop n → Prop n → Prop n
  _⇒_  : Prop n → Prop n → Prop n
  E_   : Prop (suc n) → Prop n
  A_   : Prop (suc n) → Prop n
```

Negation is defined for convenience:

```
~_ : {n : ℕ} → Prop n → Prop n
~ φ = φ ⇒ ⊥⊥
```

It is noted that because the semantics of a proposition is not a priori decidable, under a constructive (meta)theory propositions cannot be reduced to a more minimal set of connectives/quantifiers, as they typically would be in a classical setting.

The quantifiers `E_` and `A_` reflect the use of de Bruijn indices (Section 2.5): neither constructor accepts any indication of which variable is to be quantified (recall that with de Bruijn indices this is not needed), and both decrement the arity (by virtue of binding one of the free variables in the quantified proposition).

[1] The notation used in the constructors is an attempt to avoid clashing with reserved symbols (such as `∀` and `→`) and symbols from Agda's standard library which are used extensively in the proof (such as `⊥`).

3.3 Semantics of Propositions

The semantics of propositions are then defined recursively from `⟦_⟧ₐ` in accordance with the BHK interpretation:

```
⟦_⟧ : {n : ℕ} → Prop n → Vec Y n → Set
⟦ ⊥⊥ ⟧      ys = ⊥
⟦ atom a ⟧  ys = ⟦ a ⟧ₐ ys
⟦ φ₁ ∪ φ₂ ⟧ ys = (⟦ φ₁ ⟧ ys) ⊎ (⟦ φ₂ ⟧ ys)
⟦ φ₁ & φ₂ ⟧ ys = (⟦ φ₁ ⟧ ys) × (⟦ φ₂ ⟧ ys)
⟦ φ₁ ⇒ φ₂ ⟧ ys = (⟦ φ₁ ⟧ ys) → (⟦ φ₂ ⟧ ys)
⟦ E φ ⟧     ys = Σ Y (λ y → ⟦ φ ⟧ (y ∷ ys))
⟦ A φ ⟧     ys = (y : Y) → (⟦ φ ⟧ (y ∷ ys))
```

Absurdity, disjunction, conjunction, and implication are respectively mapped to the empty, disjoint union, cartesian product, and function types.

The semantics of existential quantification are represented using a Σ (dependent sum/pair) type. Members of the resulting type are pairs consisting of a value `y:Y` and an element of the inner proposition's semantics with `y` prepended to the environment, i.e., proof that the inner proposition is true with the first free variable "set to `y`".

The semantics of universal quantification are defined similarly, but use a (dependent) function type[2] in place of the Σ type—*all* values for `y` must result in the inner proposition being true.

3.4 Quantifier-Free Propositions

As quantifier-free propositions are of importance, a representation of this quality is defined:

```
data QFree {n : ℕ} : Prop n → Set where
  ⊥⊥  : QFree ⊥⊥
  atom : (a : Atom n) → QFree (atom a)
  _∪_ :  {φ₁ φ₂ : Prop n} → QFree φ₁ → QFree φ₂ → QFree (φ₁ ∪ φ₂)
  _&_ :  {φ₁ φ₂ : Prop n} → QFree φ₁ → QFree φ₂ → QFree (φ₁ & φ₂)
  _⇒_ :  {φ₁ φ₂ : Prop n} → QFree φ₁ → QFree φ₂ → QFree (φ₁ ⇒ φ₂)

~-qf_ : {n : ℕ} {φ : Prop n} → QFree φ → QFree (~ φ)
~-qf qf = qf ⇒ ⊥⊥
```

The constructors are chosen with the same names as those in `Prop`; this emphasizes the fact that `QFree` can be thought of both as a proof that a proposition is quantifier-free (`QFree φ` is inhabited if and only if `φ` is quantifier-free), and as an actual datatype for quantifier-free propositions.

Semantically speaking, all of the connectives preserve decidability: the result of joining two semantically decidable propositions with `_∪_`, `_&_`, or `_⇒_` is also semantically decidable. This is shown for `_⇒_` (with semantics `→`) as follows:

[2] A Π type, though Agda's syntax makes it of little use to write it as such.

```
_→?_ : {A B : Set} → Dec A → Dec B → Dec (A → B)
_       →? (yes b) = yes (λ _ → b)
(yes a) →? (no ¬b) = no (λ f → ¬b (f a))
(no ¬a) →? (no ¬b) = yes (λ a → contradiction a ¬a)
```

The same property can be shown for `_∪_` and `_&_` (with semantics `_⊎_` and `_×_`) in a similar manner, resulting in the following two functions:

```
_⊎?_ : {A B : Set} → Dec A → Dec B → Dec (A ⊎ B)
_×?_ : {A B : Set} → Dec A → Dec B → Dec (A × B)
```

It is also noted that the semantics of `⊥⊥`, namely `⊥`, is trivially decidable.

Given the above and that the semantics for atoms are decidable (`⟦_⟧ₐ?`), it follows by induction that the semantics of any quantifier-free proposition is decidable:

```
qfree-dec : {n : ℕ} → (φ : Prop n) → QFree φ → (e : Vec Y n) →
            Dec (⟦ φ ⟧ e)
```

3.5 Quantifier Elimination

As discussed in Section 2.1, quantifier elimination is typically accomplished by eliminating existential quantifiers one by one, from the "inside out". It is performed in that order so that when a quantifier is being eliminated, the enclosed proposition is already quantifier-free, simplifying the problem significantly.

The method by which a single quantifier is eliminated depends on the theory under consideration, making it impossible to directly define (whilst maintaining generality). Instead—in a similar manner to `DecAtom`—it is defined abstractly with a record type `QE` which captures the necessary properties of a single-step elimination procedure. A specific implementation takes the form of an object `qe : QE`.

```
record QE : Set where
  field
    step  : {n : ℕ} (φ : Prop (suc n)) → QFree φ → Prop n
    qfree : {n : ℕ} (φ : Prop (suc n)) (qf : QFree φ) →
              QFree (step φ qf)
    equiv : {n : ℕ} (φ : Prop (suc n)) (qf : QFree φ) (e : Vec Y n) →
              ⟦ E φ ⟧ e ↔ ⟦ step φ qf ⟧ e
```

The field `step` represents the single-step elimination procedure itself, accepting a quantifier-free proposition with up to $n+1$ free variables and producing one with up to n. It is noted that the input to `step` does not contain the existential quantifier to eliminate, rather it is implied—for example, to eliminate the quantifier from `E φ`, the `step` procedure is invoked on just `φ`. The field `qfree` represents a proof that `step` always produces a quantifier-free proposition. Finally, `equiv` establishes `step`'s correctness—that the propositions `E φ` and `step φ ...` are semantically equivalent.[3]

[3]The notation `A ↔ B` is defined as `(A → B) × (B → A)`.

Such a single-step procedure can then be "lifted" to eliminate all quantifiers from a proposition via recursion on the proposition's structure (the general approach, as stated before, being to eliminate quantifiers from the inside out). The cases are as follows:

1. The absurd proposition (`⊥⊥`); it is left unchanged.
2. An atom; it is left unchanged.
3. A proposition formed from disjunction, conjunction, or implication (`∪`, `&`, or `⇒`); the sub-proposition(s) are quantifier-eliminated recursively.
4. An existentially-quantified proposition (`E φ`); `φ` is quantifier-eliminated recursively, and `step` is applied to the result.
5. A universally-quantified proposition (`A φ`); `φ` is quantifier-eliminated recursively, and the quantifier is treated as its (classical) existential dual (`~ E ~`).[4]

This procedure is formalized as the function `lift-qe`:

```
lift-qe : {n : ℕ} → QE → Prop n → Prop n
lift-qe-qfree : {n : ℕ} (qe : QE) (φ : Prop n) → QFree (lift-qe qe φ)

lift-qe _ ⊥⊥ = ⊥⊥
lift-qe _ (atom atm) = atom atm
lift-qe qe (φ₁ ∪ φ₂) = (lift-qe qe φ₁) ∪ (lift-qe qe φ₂)
lift-qe qe (φ₁ & φ₂) = (lift-qe qe φ₁) & (lift-qe qe φ₂)
lift-qe qe (φ₁ ⇒ φ₂) = (lift-qe qe φ₁) ⇒ (lift-qe qe φ₂)
lift-qe qe (E φ)
  = QE.step qe (lift-qe qe φ) (lift-qe-qfree qe φ)
lift-qe qe (A φ)
  = ~ (QE.step qe (~ lift-qe qe φ) (~-qf lift-qe-qfree qe φ))
```

The function `lift-qe-qfree` (contents omitted) simply affirms that `lift-qe` does indeed eliminate quantifiers, via recursion on the proposition's structure and use of `QE.qfree`.

3.5.1 Correctness

The correctness of the lifted procedure—that `lift-qe qe φ` is equivalent to `φ`—is proven recursively based on the correctess of the single-step procedure. This takes the form of two functions, proving each direction of the equivalence:

```
lift-qe-fwd : {n : ℕ} (qe : QE) (φ : Prop n) (e : Vec Y n) →
  ⟦ φ ⟧ e → ⟦ lift-qe qe φ ⟧ e

lift-qe-bwd : {n : ℕ} (qe : QE) (φ : Prop n) (e : Vec Y n) →
  ⟦ lift-qe qe φ ⟧ e → ⟦ φ ⟧ e
```

[4] The validity of this under a constructive metatheory is not immediately obvious, and will be addressed Section 3.5.1.

For both directions, the cases `⊥⊥` and `atom` are trivial; the former is impossible and the latter is unchanged by `lift-qe`. For `∪`, `&`, and `⇒`, correctness of `lift-qe` is proven recursively on each sub-proposition, and then combined:

```
lift-qe-fwd qe (φ₁ ∪ φ₂) e
  = Sum.map (lift-qe-fwd qe φ₁ e) (lift-qe-fwd qe φ₂ e)
lift-qe-fwd qe (φ₁ & φ₂) e
  = Product.map (lift-qe-fwd qe φ₁ e) (lift-qe-fwd qe φ₂ e)
lift-qe-fwd qe (φ₁ ⇒ φ₂) e
  = λ f → lift-qe-fwd qe φ₂ e ∘ f ∘ lift-qe-bwd qe φ₁ e

lift-qe-bwd qe (φ₁ ∪ φ₂) e
  = Sum.map (lift-qe-bwd qe φ₁ e) (lift-qe-bwd qe φ₂ e)
lift-qe-bwd qe (φ₁ & φ₂) e
  = Product.map (lift-qe-bwd qe φ₁ e) (lift-qe-bwd qe φ₂ e)
lift-qe-bwd qe (φ₁ ⇒ φ₂) e
  = λ f → lift-qe-bwd qe φ₂ e ∘ f ∘ lift-qe-fwd qe φ₁ e
```

In the case of existential quantification, `lift-qe` recurses on `φ`, producing an equivalent, quantifier-free `ψ`, which `QE.step` is applied to. The reasoning behind this is as follows:

$$\exists x.\phi \iff \exists x.\psi \iff step(\psi).$$

The first equivalence is justified by the correctness of `lift-qe` on `φ`, obtained recursively, and the second by the correctness of the the single-step procedure, given by `QE.equiv`. Formalized:

```
lift-qe-fwd qe (E φ) e =
      proj₁ (QE.equiv qe (lift-qe qe φ) (lift-qe-qfree qe φ) e)
      ∘ Σ-map (λ y → lift-qe-fwd qe φ (y ∷ e))

lift-qe-bwd qe (E φ) e =
      Σ-map (λ y → lift-qe-bwd qe φ (y ∷ e))
      ∘ proj₂ (QE.equiv qe (lift-qe qe φ) (lift-qe-qfree qe φ) e)
```

where `Σ-map` proves that if $B(x) \Rightarrow C(x)$, then $\exists x.B(x) \Rightarrow \exists x.C(x)$, in this case used to obtain $\exists x.\phi \iff \exists x.\psi$ from $\phi \iff \psi$:

```
Σ-map : {A : Set} {B C : A → Set}
        ((a : A) → B a → C a) → Σ A B → Σ A C
Σ-map f (a , b) = (a , f a b)
```

The case of universal quantification is cause for mild concern, however: `lift-qe` treats the quantifier `A` as its classical dual `~ E ~`.

In a classical metatheory, correctness could be obtained as follows (once again taking ψ to be the quantifier-free equivalent of ϕ):

$$\forall x.\phi \iff \neg\exists x.\neg\phi \iff \neg\exists x.\neg\psi \iff \neg step(\neg\psi).$$

The first equivalence is justified by quantifier duality, the second by the correctness of `lift-qe` on `φ` ($\phi \iff \psi$, obtainable via recursion), and the third by the correctness of `QE.step` (`QE.equiv`). Conceptually, this corresponds to the idea of treating $\forall$ as $\neg\exists\neg$ from the start.

Under a constructive metatheory, though, the first equivalence is not valid due to the lack of complete quantifier duality: while $\forall x.\phi \Rightarrow \neg\exists x.\neg\phi$, the converse is not provable. However, in this case this can be neatly sidestepped by rearranging things slightly:

$$\forall x.\phi \iff \forall x.\psi \iff \neg\exists x.\neg\psi \iff \neg step(\neg\psi).$$

The difference here is that the quantifier duality is applied to ψ, instead of ϕ. ψ, being quantifier-free, has decidable semantics (by `qfree-dec`), and as a consequence the necessary quantifier duality can in fact be proven. General forms of the duality are formalized as follows:

```
∀-duality-fwd : {A : Set} {B : A → Set} →
  ((a : A) → B a) → ¬ Σ A (¬_ ∘ B)
∀-duality-fwd all-true (a , is-false) = is-false (all-true a)

∀-duality-bwd : {A : Set} {B : A → Set} → ((a : A) → Dec (B a)) →
  ¬ Σ A (¬_ ∘ B) → ((a : A) → B a)
∀-duality-bwd decide none-false a with decide a
... | yes a-true = a-true
... | no a-false = ⊥-elim (none-false (a , a-false))
```

It is noted that the "backward" direction requires that `B` be decidable.[5] The correctness then proceeds as outlined above:

```
lift-qe-fwd qe (A φ) e =
  contraposition
      (proj₂ (QE.equiv qe (~ lift-qe qe φ) (~-qf lift-qe-qfree qe φ) e))
  ∘ ∀-duality-fwd
  ∘ Π-map (λ y → lift-qe-fwd qe φ (y ∷ e))

lift-qe-bwd qe (A φ) e =
  Π-map (λ y → lift-qe-bwd qe φ (y ∷ e))
  ∘ ∀-duality-bwd
      (λ y → qfree-dec (lift-qe qe φ) (lift-qe-qfree qe φ) (y ∷ e))
  ∘ contraposition
      (proj₁ (QE.equiv qe (~ lift-qe qe φ) (~-qf lift-qe-qfree qe φ) e))
```

where `Π-map` is the dependent product/universal quantification counterpart of `Σ-map`:

```
Π-map : {A : Set} {B C : A → Set} →
  ((a : A) → B a → C a) → ((a : A) → B a) → ((a : A) → C a)
Π-map f g a = f a (g a)
```

[5] While it could have been formulated to use the weaker requirement that `¬ ¬ B a → B a`, there is no particular benefit to doing so in this case.

3.6 Decidability

Given a single-step elimination procedure `qe : QE`, the decidability of any proposition `φ` follows: `lift-qe qe φ` produces an equivalent, quantifier-free proposition `ψ`. As such, `ψ` is decidable (`qfree-dec`). Because `φ` and `ψ` are semantically equivalent (`lift-qe-fwd`, `lift-qe-bwd`), this immediately results in the decidability of `φ`.

```
⟦_⟧? : {n : ℕ} → (φ : Prop n) → (e : Vec Y n) → Dec (⟦ φ ⟧ e)
⟦ φ ⟧? e with qfree-dec (lift-qe qe φ) (lift-qe-qfree qe φ) e
... | yes ⟦ψ⟧ = yes (lift-qe-bwd qe φ e ⟦ψ⟧)
... | no ¬⟦ψ⟧ = no (¬⟦ψ⟧ ∘ lift-qe-fwd qe φ e)
```

With decidability, a number of interesting "classical" results can be proven. Under a constructive metatheory, however, these results are considerably stronger than under a classical one, due to the stricter notion of proof (see Section 2.2).

First, the law of excluded middle is proven:

```
LEM : {n : ℕ} (φ : Prop n) (e : Vec Y n) → ⟦ φ ∪ (~ φ) ⟧ e
LEM φ e with ⟦ φ ⟧? e
... | yes ⟦φ⟧ = inj₁ ⟦φ⟧
... | no ¬⟦φ⟧ = inj₂ ¬⟦φ⟧
```

Recall that under a constructive metatheory, this is the same as decidability; a proof that ϕ is true, or a proof that ϕ is false.

Second, for any proposition ϕ, $(\exists x.\phi) \vee (\forall x.\neg\phi)$:

```
∃-or-∀¬ : {n : ℕ} (φ : Prop (suc n)) (e : Vec Y n) →
  ⟦ (E φ) ∪ (A (~ φ)) ⟧ e
∃-or-∀¬ φ e with ⟦ E φ ⟧? e
... | yes ⟦Eφ⟧ = inj₁ ⟦Eφ⟧
... | no ¬⟦Eφ⟧ = inj₂ (λ y → λ ⟦φ⟧ → contradiction (y , ⟦φ⟧) ¬⟦Eφ⟧)
```

This produces either (i) a value (witness) y and a proof that it renders ϕ true, that is to say a "solution", or (ii) a proof that ϕ is false for any given y.

Third, it can be proven that $(\forall x.\phi) \vee (\exists x.\neg\phi)$:

```
∀-or-∃¬ : {n : ℕ} (φ : Prop (suc n)) (e : Vec Y n) →
  ⟦ (A φ) ∪ (E (~ φ)) ⟧ e
∀-or-∃¬ φ e with ⟦ E (~ φ) ⟧? e
... | yes ⟦E~φ⟧ = inj₂ ⟦E~φ⟧
... | no ¬⟦E~φ⟧ = inj₁ (λ y →
  [ id , (λ ¬⟦φ⟧ → contradiction (y , ¬⟦φ⟧) ¬⟦E~φ⟧) ]′
  (LEM φ (y ∷ e)))
```

Under a constructive metatheory this produces either (i) a function which proves ϕ to be true for any given value y, or (ii) a counter-example; a value y and a proof that it makes ϕ false.

3.7 Disjunctive Normal Form and Products

Having proven that a single step of existential quantifier elimination can be generalized to full quantifier elimination, our attention turns to the details of the former.

The most basic formulation of a single-step procedure is one that accepts a quantifier-free proposition ϕ, and produces a quantifier-free proposition ψ such that $(\exists x.\phi) \iff \psi$. There are no restrictions on the form of ϕ, other than that it is quantifier-free. In practice, many quantifier elimination procedures require that ϕ be transformed into a special form first, as seen in the work of Herbrand [12] and of Nipkow [16].

As discussed in Section 2.1 one such form is Disjunctive Normal Form (DNF), where propositions take the shape of a disjunction of conjunctions of literals. Literals are often referred to here as "factors", terminology borrowed from Herbrand's thesis [12]. For example, if A, B, C and D are atoms, the proposition

$$A \Rightarrow (B \wedge (C \Rightarrow D))$$

can be transformed into the proposition in DNF

$$(\neg A) \vee (B \wedge \neg C) \vee (B \wedge D).$$

The transformation makes extensive use of double negation elimination and De Morgan's laws, which are available in a constructive theory if quantifier-free propositions are decidable (which is the case here; see Section 3.4).

Disjunctive normal form is particularly useful for existential quantifier elimination because the quantifier distributes across disjunction:

$$\exists x.(C_1 \vee C_2 \vee \ldots \vee C_n) \iff (\exists x.C_1) \vee (\exists x.C_2) \vee \ldots \vee (\exists x.C_n)$$

This allows elimination to be carried out separately on each of the conjunctions ("products") $C_1, C_2, \ldots, C_n$, reducing the problem to elimination on conjunctions of factors. Disjunctive normal form can therefore be viewed in this context as a means by which single-step elimination on *products* can be generalized to single-step elimination on *any quantifier-free proposition*.

From the standpoint of actually carrying out quantifier elimination (as opposed to only proving that it is possible), there is a downside: conversion to DNF causes an (exponential) explosion in the size of the proposition. As conversion to DNF is performed every time a quantifier is removed, nested quantifiers cause this explosion to be iterated, resulting in a tower-of-exponents proposition size [16]. While there are some single-step procedures that mitigate this by using less explosive normal forms, they are outside the scope of this project (but could well be integrated at a later time).

Because of the rigid structures of DNF and CNF, propositions in these forms can be represented as two-layer list structures—in the case of DNF, a list (implicit disjunction) of lists (implicit conjunction) of factors. While they could also be represented by a standard `Prop` along with a proof that it is in the specified form, this often makes manipulation awkward, so the list-based approach is used instead:

```
-- An atom or its negation
data Factor : ℕ → Set where
  +_ : {n : ℕ} → Atom n → Factor n
  -_ : {n : ℕ} → Atom n → Factor n

-- Product (conjunction) of factors
Prod : ℕ → Set
Prod n = List (Factor n)

-- Sum (disjunction) of factors
Sum : ℕ → Set
Sum n = List (Factor n)

-- Disjunctive Normal Form
DNF : ℕ → Set
DNF n = List (Prod n)

-- Conjunctive Normal Form
CNF : ℕ → Set
CNF n = List (Sum n)
```

The downside of this choice is that interpretation functions are necessary to convert each of the above forms into the proposition (in the `Prop` sense) that it represents. The respective interpretation functions are `F.i`, `P.i`, `S.i`, `D.i`, and `C.i`, and are defined as expected. `Prod` and `Sum`, and `CNF` and `DNF` are given different names—despite representing the same types—in order to clarify the intended interpretations of propositions in those forms.

Conversion to DNF/CNF is then defined (mutually) recursively:

```
dnf : {n : ℕ} (p : Prop n) (qf : QFree p) → DNF n
cnf : {n : ℕ} (p : Prop n) (qf : QFree p) → CNF n

dnf _ ⊥⊥ = []
dnf .(atom a) (atom a) = [ [ + a ] ]
dnf (p₁ ∪ p₂) (qf₁ ∪ qf₂) = dnf p₁ qf₁ ++ dnf p₂ qf₂
dnf (p₁ & p₂) (qf₁ & qf₂) = mix (dnf p₁ qf₁) (dnf p₂ qf₂)
dnf (p₁ ⇒ p₂) (qf₁ ⇒ qf₂) = (dual₂ (cnf p₁ qf₁)) ++ (dnf p₂ qf₂)

cnf _ ⊥⊥ = [ [] ]
cnf .(atom a) (atom a) = [ [ + a ] ]
cnf (p₁ ∪ p₂) (qf₁ ∪ qf₂) = mix (cnf p₁ qf₁) (cnf p₂ qf₂)
cnf (p₁ & p₂) (qf₁ & qf₂) = cnf p₁ qf₁ ++ cnf p₂ qf₂
cnf (p₁ ⇒ p₂) (qf₁ ⇒ qf₂) = mix (dual₂ (dnf p₁ qf₁)) (cnf p₂ qf₂)
```

The function `dual₂` produces the dual of a DNF/CNF by negating each factor. `mix` is similar to a cartesian product, but concatenates each of the resulting pairs of lists. It is perhaps best shown by example, in this case on two DNF formulae. From a list perspective:

```
mix [A₁, A₂, A₃] [B₁, B₂, B₃]
= [A₁ ++ B₁, A₁ ++ B₂, ... , A₃ ++ B₂, A₃ ++ B₃]
```

and in terms of the represented propositions:

$$mix(A_1 \vee A_2 \vee A_3,\, B_1 \vee B_2 \vee B_3)$$

$$= (A_1 \wedge B_1) \vee (A_1 \wedge B_2) \vee \ldots \vee (A_3 \wedge B_2) \vee (A_3 \wedge B_3).$$

Correctness of `dnf` and `cnf` is proven in a straightforward but tedious manner, noting that classical results such as De Morgan's laws can be used because the propositions being converted are quantifier-free (and thus decidable). The signatures of the resulting lemmas are:

```
dnf-fwd : {n : ℕ} (p : Prop n) (qf : QFree p) (e : Vec Y n) →
  ⟦ p ⟧ e → ⟦ D.i (dnf p qf) ⟧ e
dnf-bwd : {n : ℕ} (p : Prop n) (qf : QFree p) (e : Vec Y n) →
  ⟦ D.i (dnf p qf) ⟧ e → ⟦ p ⟧ e
cnf-fwd : {n : ℕ} (p : Prop n) (qf : QFree p) (e : Vec Y n) →
  ⟦ p ⟧ e → ⟦ C.i (cnf p qf) ⟧ e
cnf-bwd : {n : ℕ} (p : Prop n) (qf : QFree p) (e : Vec Y n) →
  ⟦ C.i (cnf p qf) ⟧ e → ⟦ p ⟧ e
```

Single-step elimination on DNF and on products may now be defined. The types closely resemble `QE`, but accept propositions in DNF/product form. `equiv` is modified accordingly.

```
record DNFQE : Set where
  field
    step  : {n : ℕ} → DNF (suc n) → Prop n
    qfree : {n : ℕ} (φ : DNF (suc n)) → QFree (step φ)
    equiv : {n : ℕ} (φ : DNF (suc n)) (e : Vec Y n) →
      ⟦ E (D.i φ) ⟧ e ↔ ⟦ step φ ⟧ e

record ProdQE : Set where
  field
    step  : {n : ℕ} → Prod (suc n) → Prop n
    qfree : {n : ℕ} (φ : Prod (suc n)) → QFree (step φ)
    equiv : {n : ℕ} (φ : Prod (suc n)) (e : Vec Y n) →
      ⟦ E (P.i φ) ⟧ e ↔ ⟦ step φ ⟧ e
```

Single-step elimination on products (`ProdQE`) can be lifted to single-step elimination on DNF (`DNFQE`) by applying it to each disjunct of the DNF. Correctness of this follows from the distribution of $\exists$ across disjunction. The result is a function `Prod⇒DNF.lift : ProdQE → DNFQE`.

Elimination on DNF can be generalized to single-step elimination on any quantifier-free proposition (`QE`) by simply first converting the given proposition to DNF:

```
lift-dnf-qe : DNFQE → QE
lift-dnf-qe qe = record
  { step  = λ φ qf → DNFQE.step qe (dnf φ qf)
  ; qfree = λ φ qf → DNFQE.qfree qe (dnf φ qf)
  ; equiv = λ φ qf e →
    ( proj₁ (DNFQE.equiv qe (dnf φ qf) e)
      ∘ Σ-map (λ y → dnf-fwd φ qf (y ∷ e))
    , Σ-map (λ y → dnf-bwd φ qf (y ∷ e))
      ∘ proj₂ (DNFQE.equiv qe (dnf φ qf) e))
  }
```

Combining these stages, a proof is obtained that single-step elimination on products can be generalized to single-step elimination on arbitrary quantifier-free propositions:

```
lift-prod-qe : ProdQE → QE
lift-prod-qe = lift-dnf-qe ∘ Prod⇒DNF.lift
```

Chapter 4

The Theory of Successor

4.1 Overview

An example theory to which this framework is applied is the theory of successor on the natural numbers (SN). Atoms are equalities between terms, which in turn take one of three forms: a variable, a constant, or a variable plus a constant. All constants are natural numbers, and variables take on natural numbers as values. The following is an example of a proposition in SN:

$$\forall x.(x = 0 \vee \exists y.x = y + 1).$$

An equivalent formulation for terms—which leads to a more convenient representation—is to define a term as a natural number plus a "base", the latter being either the constant zero or a variable. As it simplifies the proof somewhat, this alternative formulation is used.

As discussed in Section 3.1, the definitions and behaviors of `Prop`, `⟦_⟧`, and `⟦_⟧ₐ` to a great degree force the use of de Bruijn indices for variables. This is convenient for the theory-dependent code as well: the variable to eliminate has index zero, which (as well as removing the necessity of a "variable-to-eliminate" parameter) allows dependence on the variable—by a term or atom, for example—to be determined by pattern-matching, rather than more involved comparisons.

Moving forward with de Bruijn indices, we recall that atoms (and propositions) are indexed by an upper bound `n : ℕ` on the number of free variables they have (i.e., their arity). This means that any variables in an atom of type `Atom n` have de Bruijn indices in the range $0, 1, \ldots, n-1$. This suggests the use of the indexed type `Fin` for variables,[1] as `Fin n` represents exactly that set.

The following datatypes are therefore used to represent SN:

```
data Base (n : ℕ) : Set where
  ∅ : Base n
  var : Fin n → Base n

data Term (n : ℕ) : Set where
  S : ℕ → Base n → Term n
```

[1] `Data.Fin` from the Agda standard library.

```
data Atom (n : ℕ) : Set where
  _==_ : Term n → Term n → Atom n
```

The constructors for `Base` (`⌀` and `var`) respectively represent the constant zero and a variable. The constructor used for terms, `S`, takes after the notation $S^k(x)$, i.e. iteration of the successor function, and was chosen in lieu of `_+_` to avoid ambiguity with the (metatheory-level) addition function on natural numbers.

Various helper function are defined on `Base` and `Term`, in the respective modules `B` and `T`. First, a predicate `dep₀` to indicate that a base (or term) contains variable zero. For bases (`B.dep₀`), it is defined as:

```
dep₀ : {n : ℕ} → Base n → Set
dep₀ ⌀ = ⊥
dep₀ (var zero) = ⊤
dep₀ (var (suc _)) = ⊥
```

A trivial decision procedure `B.dep₀?` is defined as well. The definitions for terms (`T.dep₀`, `T.dep₀?`) simply apply the above to the term's base.

Second, functions `ξ` and `ξ⁻¹` are defined to (respectively) increment and decrement the de Bruijn indices of variables. They are first defined on bases (`B.ξ and B.ξ⁻¹`):

```
ξ : {n : ℕ} → Base n → Base (suc n)
ξ ⌀ = ⌀
ξ (var i) = var (suc i)
ξ⁻¹ : {n : ℕ} (b : Base (suc n)) → ¬ dep₀ b → Base n
ξ⁻¹ ⌀ _ = ⌀
ξ⁻¹ (var zero) x = contradiction tt x
ξ⁻¹ (var (suc i)) _ = var i
```

It is noted that `ξ⁻¹` requires that the base is not `var zero`. Versions for terms are also defined; they apply `B.ξ` or `B.ξ⁻¹` to the term's base.

The semantics for bases and terms—in both cases natural numbers—are then defined:[2]

```
⟦_⟧b : {n : ℕ} → Base n → Vec ℕ n → ℕ
⟦ ⌀ ⟧b _ = zero
⟦ var k ⟧b e = lookup k e

⟦_⟧ₜ : {n : ℕ} → Term n → Vec ℕ n → ℕ
⟦ S k b ⟧ₜ e = k + ⟦ b ⟧b e
```

The semantics of atoms can be defined from the above using the equality relation `_≡_`:

```
⟦_⟧ₐ : {n : ℕ} → Atom n → Vec ℕ n → Set
⟦ t₁ == t₂ ⟧ₐ e = ⟦ t₁ ⟧ₜ e ≡ ⟦ t₂ ⟧ₜ e
```

[2] As of this time, Unicode appears not to have a subscript "b" character, hence the slightly inconsistent notation.

It is noted here that this produces an Agda-proposition, rather than a boolean, as discussed in Section 2.4. The requisite decision procedure is obtained from the decidability of equality on natural numbers, `_≟_`:

```
⟦_⟧ₐ? : {n : ℕ} (a : Atom n) (e : Vec ℕ n) → Dec (⟦ a ⟧ₐ e)
⟦ t₁ == t₂ ⟧ₐ? e = ⟦ t₁ ⟧ₜ e ≟ ⟦ t₂ ⟧ₜ e
```

The choice of natural numbers as values, `Atom`, `⟦_⟧ₐ`, and `⟦_⟧ₐ?` together meet the requirements for atoms as set out in Section 3.1:

```
sn-da : DecAtom
sn-da = record
  { Y = ℕ
  ; Atom = Atom
  ; ⟦_⟧ₐ = ⟦_⟧ₐ
  ; ⟦_⟧ₐ? = ⟦_⟧ₐ?
  }
```

Given the lifting procedures developed in the previous sections (specifically `lift-prod-qe : ProdQE → QE`, and `lift-qe : QE → Prop n → Prop n`), full quantifier elimination—and therefore decidability—of this theory can be obtained if a single-step elimination procedure on products can be formalized as a `ProdQE`.

4.2 Elimination on Products

In contrast with previous sections, for much of the elimination procedure (and its accompanying correctness proofs) the formal representations do not convey the underlying reasoning nearly as effectively as a slightly more abstract mathematical notation. One of the main causes of this is "constructor bloat", as exemplified by the difference between:

```
- (S a (var zero) == S b (var (suc zero)))
```

and:

$$x + a \neq y + b.$$

Consequently, the latter notation is used when possible. In this notation, the symbol x is used to denote `var zero`, the variable to eliminate.[3]

To restate the problem, there is a product $\phi = F_1 \wedge F_2 \wedge \ldots \wedge F_m$, possibly containing the variable x, and the goal is to produce a proposition ψ that is equivalent to $\exists x.\phi$ and does not contain x. The approach used is a simplified form of the one used by Herbrand [12].

There are two main cases, depending on whether or not the product contains a factor F_{sub} of the form $x + k = term$ (or, symmetrically, $term = x + k$), where $term$ does not contain x. A factor of this form is referred to as a "substitutable factor", or a "non-trivial equality".

[3] x is not to be confused with `x` as used in code.

Case 1

If no such factor is present, then x may only occur in three kinds of factors: (i) trivial equalities ($x + a = x + b$), (ii) trivial inequalities ($x + a \neq x + b$), and (iii) non-trivial inequalities ($x + a \neq term$ or symmetrical, where $x \notin term$). As x can be eliminated from factors of the first two kinds by simplifying them, this leaves only non-trivial inequalities.

To deal with those, it is noted that each non-trivial inequality is satisfied by all but perhaps one value for x—the value for which both sides of the inequality are in fact equal (if such a value exists). For example,

$$x + 3 \neq y + 4$$

is satisfied for every value of x other than $y + 1$. More generally, for a factor $x + a \neq term$, this value is $term - a$ if $term \geq a$ (and does not exist if not). Since there are a finite number of such factors in the product, this presents a finite set of values which x must avoid in order to satisfy the factors in which it appears. As there are an infinite number of natural numbers, this is always possible (a constructive approach is to let m denote the maximum "forbidden" value,[4] and let x take on the value $m + 1$). Therefore there always exists an x that makes the factors containing it true, so these factors (along with the quantifier) can be dropped, leaving only the factors that do not depend on x. Symbolically:

$$\psi = \mathit{filter}_{x \notin}(\mathit{simplify}(\phi))$$

Correctness in the "forward" direction ($(\exists x.\phi) \Rightarrow \psi$) is trivial, as factors are only removed from the product (or simplified). For the "backward" direction ($\psi \Rightarrow \exists x.\phi$), the value of x can be chosen as discussed above to satisfy the inequalities that are present in ϕ but not ψ.

Case 2

In the case that such a factor F_{sub}, of the form $x + k = term$ with $x \notin term$ (or symmetrical) does occur in the product, it is first observed that F_{sub} implies that $term \geq k$. The inequalities $term \neq 0, term \neq 1, \ldots, term \neq (k - 1)$ are added to the product accordingly. Then, F_{sub} is used as a basis for substitution throughout the product: when an x is encountered, it is replaced by $term$ and k is added to the other side of the factor. For example:

$$x + a \neq y + b$$

becomes

$$term + a \neq y + (k + b).$$

It is noted that a is absorbed into the constant portion of $term$, and $k+b$ is condensed into one natural number, so the result is still a valid factor. Such substitution having

[4] The value of m depends on the valuation of the other variables, but the approach works for *any* valuation.

been carried out on every factor, x is no longer present in the product, and the procedure is complete. Symbolically:

$$\psi = (term \neq 0) \wedge (term \neq 1) \wedge \ldots \wedge (term \neq k-1) \wedge substitute(x, k, term, \phi)$$

As for correctness, the substitution process is easily proven sound with basic arithmetic (given that $x + k = term$, which is a priori the case in the forward direction). The introduced factors $term \neq 0, term \neq 1, \ldots, term \neq (k-1)$ are trivially true in the forward direction, and in the backward direction ensure that a value for x can be chosen that satisfies $x + k = term$, thereby rendering the substitution sound for the backward direction as well.

4.3 Formalization

As the code directly implements the higher-level descriptions above, but with greatly increased verbosity, this section focuses on the involved types and structure, glossing over some of the less conceptually valuable implementation details.

4.3.1 Procedure

An important first consideration is that since variable zero (x) is being eliminated and its quantifier removed, the de Bruijn indices of the other variables must be decremented in order to continue pointing to their respective quantifiers. One way of handling this is to eliminate variable zero from the product *without* changing the indices of other variables, and then to decrement them afterward (the latter step typically requiring a proof that variable zero is no longer present in order to guarantee correctness).

Another approach is to "decrement as you go", where each factor of arity $n + 1$ is transformed into one with arity n as the elimination is being carried out. This removes the need to prove that variable zero is gone (instead, it is implicit in the reduction in arity). Due to the relative simplicity of the elimination procedure, this method is used.

The primary cases of the elimination procedure detailed in the previous subsection depend on whether or not a "substitutable factor"—a factor of the form $x+k = term$ (or symmetrical, with $x \notin term$)—is in the product. Therefore, a predicate for such factors is introduced:

```
SubFactor : {n : ℕ} → Factor n → Set
SubFactor (+ (t₁ == t₂))
  = (T.dep₀ t₁ × ¬ T.dep₀ t₂) ⊎ ((¬ T.dep₀ t₁) × T.dep₀ t₂)
SubFactor (- _) = ⊥
```

`SubFactor` therefore holds for any equality where x occurs on just one side. It is shown that `SubFactor` is decidable via `SubFactor?` (omitted). The relevant details of a factor for which `SubFactor` holds, namely `k` and `term`, can be stored in a record `Sub`:

```
record Sub (n : ℕ) : Set where
  constructor Subst
  field
    k : ℕ
    term : Term n
```

A function `getSub` is used to extract a `Sub` from a factor for which SubFactor holds, and a function `iSub` turns a `Sub` into the corresponding factor ($x + k = term$). `iSub` and `getSub` are shown to be inverses of each other (up to semantic equivalence).

To assist in searching for a `SubFactor` in a product, a datatype for locating an item in a list is created:

```
data _∈_ {A : Set} (a : A) : List A → Set where
  here : (as : List A) → a ∈ (a ∷ as)
  there : {as : List A} (b : A) → a ∈ as → a ∈ (b ∷ as)
```

It is used to define a function `first` which, given a list and a decidable predicate P, either finds the first item in the list for which P holds, or produces a proof that P does not hold for any item in the list:

```
first : {A : Set} (P : A → Set) → ((a : A) → Dec (P a)) →
  (as : List A) →
  (Σ A (λ a → (P a × a ∈ as))) ⊎ (allP (¬_ ∘ P) as)
```

This is used in conjunction with `SubFactor?` to determine whether or not the product contains a substitutable factor, and therefore which of the two cases described in the previous subsection applies.

Case 1

If no such factor exists, a function `reduce-factor-nosubs` is mapped to every factor in the product. The function has two effects: trivial (in)equalities involving variable zero (such as $x+a = x+b$) are reduced by dropping the variable from both sides, and non-trivial inequalities involving variable zero (such as $x + a \neq y + b$) are effectively dropped from the product entirely by being replaced by a trivially true factor:[5]

```
reduce-factor-nosubs : {n : ℕ}
  (f : Factor (suc n)) → ¬ SubFactor f → Factor n
reduce-factor-nosubs (+ (S a x == S b y)) ¬sub
  with B.dep₀? x | B.dep₀? y
... | yes x0 | yes y0 = + (S a ∅ == S b ∅)
... | yes x0 | no ¬y0 = contradiction (inj₁ (x0 , ¬y0)) ¬sub
... | no ¬x0 | yes y0 = contradiction (inj₂ (¬x0 , y0)) ¬sub
... | no ¬x0 | no ¬y0 = + (S a (B.ξ⁻¹ x ¬x0) == S b (B.ξ⁻¹ y ¬y0))
reduce-factor-nosubs (- (S a x == S b y)) _
  with B.dep₀? x | B.dep₀? y
```

[5] While this is convenient for proofs, it is very inefficient. Such considerations are discussed in Section 5.1.

```
... | yes x0 | yes y0 = - (S a ø == S b ø)
... | yes x0 | no ¬y0 = + trueₐ
... | no ¬x0 | yes y0 = + trueₐ
... | no ¬x0 | no ¬y0 = - (S a (B.ξ⁻¹ x ¬x0) == S b (B.ξ⁻¹ y ¬y0))
```

The cases are divided based on whether the factor represents an equality or an inequality (`+ ...` or `- ...`), and whether or not the left and/or right term contain variable zero. Trivial (in)equalities are therefore the `yes`/`yes` cases (variable zero is on both sides), non-trivial (in)equalities the `yes`/`no` and `no`/`yes` cases (variable zero occurs on just one side), and (in)equalities without variable zero the `no`/`no` cases. Non-trivial equality is not possible, as the factor would then constitute a `SubFactor`, which we know not to be present. `trueₐ`, substituted in the non-trivial inequality cases, is the trivially true atom $0 = 0$.

Case 2

If, on the other hand, such a factor does exist, then substitution is carried out on each factor:

```
reduce-atom-sub : {n : ℕ} → Sub n → Atom (suc n) → Atom n
reduce-atom-sub (Subst k term) (S a x == S b y)
  with B.dep₀? x | B.dep₀? y
... | yes x0 | yes y0 = S a ø == S b ø
... | yes x0 | no ¬y0 = T.add a term == S (k + b) (B.ξ⁻¹ y ¬y0)
... | no ¬x0 | yes y0 = S (k + a) (B.ξ⁻¹ x ¬x0) == T.add b term
... | no ¬x0 | no ¬y0 = S a (B.ξ⁻¹ x ¬x0) ==  S b (B.ξ⁻¹ y ¬y0)

reduce-factor-sub : {n : ℕ} → Sub n → Factor (suc n) → Factor n
reduce-factor-sub sub (+ a) = + (reduce-atom-sub sub a)
reduce-factor-sub sub (- a) = - (reduce-atom-sub sub a)
```

Once again, trivial terms involving variable zero are simplified, and de Bruijn indices of other variables are decremented. When variable zero occurs on just one side of the (in)equality, substitution is performed using `k` and `term`.

The inequalities $term \neq 0, term \neq 1, \ldots, term \neq (k-1)$ are generated by the following functions:

```
ineqs′ : {n : ℕ} → ℕ → Term n → List (Factor n)
ineqs′ zero _ = []
ineqs′ (suc m) term = (- (term == S m ø)) ∷ (ineqs′ m term)

ineqs : {n : ℕ} → Sub n → List (Factor n)
ineqs (Subst k term) = ineqs′ k term
```

They are prepended to the results of substitution:

```
elim-with-sub : {n : ℕ} → Sub n → List (Factor (suc n)) → List (Factor n)
elim-with-sub sub fs = (ineqs sub) ++ (map (reduce-factor-sub sub) fs)
```

Assembling the two cases, the single-step elimination procedure is obtained:[6]

```
elim-prod : {n : ℕ} → List (Factor (suc n)) → List (Factor n)
elim-prod fs with first SubFactor SubFactor? fs
elim-prod fs | inj₁ (f , fsub , _) = elim-with-sub (getSub f fsub) fs
elim-prod fs | inj₂ none-sub = mapWithP reduce-factor-nosubs fs none-sub
```

4.3.2 Correctness

The correctness proof is broken into the two directions, forward ($\exists x.\phi \Rightarrow \psi$) and backward ($\psi \Rightarrow \exists x.\phi$):

```
elim-prod-fwd : {n : ℕ} (φ : Prod (suc n))
  → (e : Vec ℕ n)
  → ⟦ E (P.i φ) ⟧ e
  → ⟦ P.i (elim-prod φ) ⟧ e

elim-prod-bwd : {n : ℕ} (φ : Prod (suc n))
  → (e : Vec ℕ n)
  → ⟦ P.i (elim-prod φ) ⟧ e
  → ⟦ E (P.i φ) ⟧ e
```

Each direction is broken into the same two cases as the elimination procedure.

Forward, case 1

As the only substantial effect of the procedure is the replacement of certain factors by `trueₐ`, correctness in this direction is trivial.

Backward, case 1

In the backward direction, it must be proven that a value v for variable zero can be chosen so that the (non-trivial) inequalities in which it occurs in ϕ are satisfied (recall that these were dropped by the elimination procedure; otherwise ϕ and ψ are the same[7]). This is accomplished by determining the set of values which fail to satisfy the inequalities, and then choosing a natural number not in that set.

For a single factor, the list of "forbidden" values is computed by the `forbidden` function:

```
forbidden : {n : ℕ} → Vec ℕ n → Factor (suc n) → List ℕ
```

For factors which (when simplified) do not depend on the value of variable zero, the empty list is returned. For non-trivial inequalities, the forbidden value (if any) is computed as described in Section 4.2. Non-trivial equalities are, as discussed previously, impossible; they would constitute `SubFactor`s, which are known not to be present.

[6] Technically, as the result is a product it must be turned back into a `Prop` using `P.i`.

[7] Trivial simplification nonwithstanding.

It is proven (`forbidden-lemma`) that a non-trivial inequality Q, containing variable zero, is false (equal) only when the value for variable zero is in $forbidden(Q)$. Contraposition proves that Q is true (not equal) if the value is not in $forbidden(Q)$.

These values can be aggregated (concatenated) for each factor in the product, resulting in a list of all values which variable zero cannot be:

```
forbiddens : {n : ℕ} → Vec ℕ n → List (Factor (suc n)) → List ℕ
```

A function `fresh` provides a way to select a number *not* in a list:

```
fresh : List ℕ → ℕ
fresh-∉ : (xs : List ℕ) → ¬ (fresh xs) ∈ xs
```

Choosing the value `v = fresh (forbiddens φ)` for variable zero is then proven to satisfy each of the inequalities: `fresh-∉` proves that `v ∉ forbiddens φ`, which implies that `v ∉ forbidden f` for each factor `f`, which means each `f` is satisfied by virtue of `forbidden-lemma`.

Forward, case 2

The forward direction is straightforward: the presence of the equality $x + k = term$ justifies the substitutions performed on each term:

```
reduce-factor-sub-fwd : {n : ℕ} → (sub : Sub n) (f : Factor (suc n))
  → (v : ℕ) (e : Vec ℕ n)
  → ⟦ iSub sub ⟧ₐ (v ∷ e)
  → ⟦ F.i f ⟧ (v ∷ e)
  → ⟦ F.i (reduce-factor-sub sub f) ⟧ e
```

It also justifies the introduction of the inequalities $term \neq 0, term \neq 1, \ldots, term \neq (k-1)$:

```
ineqs-fwd : {n : ℕ} (sub : Sub n) (v : ℕ) (e : Vec ℕ n)
  → ⟦ iSub sub ⟧ₐ (v ∷ e)
  → ⟦ P.i (ineqs sub) ⟧ e
```

Backward, case 2

First it is proven that the added inequalities $term \neq 0, term \neq 1, \ldots, term \neq (k-1)$ imply that $k \leq term$:

```
ineqs′-bwd : {n : ℕ} (k : ℕ) (term : Term n) (e : Vec ℕ n)
  → ⟦ P.i (ineqs′ k term) ⟧ e
  → k ≤ ⟦ term ⟧ₜ e
```

Further manipulation proves that a value v exists such that $v + k = term$:[8]

```
ineqs-bwd : {n : ℕ} (sub : Sub n) (e : Vec ℕ n)
  → ⟦ P.i (ineqs sub) ⟧ e
  → Σ ℕ (λ v → ⟦ iSub sub ⟧ₐ (v ∷ e))
```

[8] Recall that `sub : Sub` encapsulates k and $term$, and `iSub sub` is the factor $x + k = term$.

This value v is chosen for variable zero, and the proof that $v + k = term$ is used to justify the (reverse) substitutions on each of the other factors:

```
reduce-factor-sub-bwd : {n : ℕ} → (sub : Sub n) (f : Factor (suc n))
  → (v : ℕ) (e : Vec ℕ n)
  → ⟦ iSub sub ⟧ₐ (v ∷ e)
  → ⟦ F.i (reduce-factor-sub sub f) ⟧ e
  → ⟦ F.i f ⟧ (v ∷ e)
```

The elimination procedure, a proof that the result is quantifier-free,[9] and its proofs of equivalence constitute a `ProdQE`:

```
sn-prod-qe : ProdQE
sn-prod-qe = record
  { step  = P.i ∘ elim-prod
  ; qfree = P.qf ∘ elim-prod
  ; equiv = λ φ e → (elim-prod-fwd φ e , elim-prod-bwd φ e)
  }
```

This can then be lifted to a general single-step elimination procedure:

```
sn-qe : QE
sn-qe = lift-prod-qe sn-prod-qe
```

And finally, the "box of results" from Section 3.6 that follow can be opened:

```
open WithQE sn-qe public
```

4.4 Demonstration

The resulting decision procedure and consequences are demonstrated on several small propositions in order to give a sense of the benefits offered by a constructive approach. First, a simple "system of equations":

```
test₀ : Prop zero
test₀ = E E (                                          -- ∃x.∃y.
    (atom (S 3 (var (fsuc fzero)) == S 1 (var fzero))) -- 3+x=1+y
  & (atom (S 8 ∅ == S 4 (var fzero))))                 -- 8=4+y
```

Normalizing `⟦ test₀ ⟧? []` (i.e., running the decision procedure) yields:[10]

```
  yes (2 , 4 , refl , refl)
```

The `2` and `4` are witnesses to the existential quantifiers, which is to say values for x and y, and the pair of `refl` constitute a proof of the inner conjunction (under the environment `[4,2]`).

Next, a proposition with a universal quantifier is decided:

[9] An immediate consequence of the fact that the result is a product of factors.

[10] Technically, each `refl` appeared as `.Agda.Builtin.Equality._≡_.refl`, but as such fluff is of little value it will be ignored in this section.

```
-- ∀x.(x=0 ∨ ∃y.x=y+1)
test₁ : Prop zero
test₁ = A ((atom (S 0 (var fzero) == S 0 ∅))
  ∪ (E (atom (S 0 (var (fsuc fzero)) == S 1 (var fzero)))))
```

`⟦ test₁ ⟧? []` normalizes to `yes` followed by a (424-line) function that proves the inner proposition for any given x.

Finally, a proposition with a free variable is examined:

```
-- x=0 ∨ ∃y.x=y+2
test₂ : Prop 1
test₂ = ((atom (S 0 (var fzero) == S 0 ∅))
  ∪ (E (atom (S 0 (var (fsuc fzero)) == S 2 (var fzero)))))
```

The function `∀-or-∃¬` (Section 3.6) is run on `test₂`. As `test₂` is not true for all values, a counterexample is produced instead: `inj₂ (1 , ...)`, where `...` is a trivial proof that $1 \neq 0$ and a lengthy proof that no y exists such that $1 = y + 2$.

Chapter 5

Conclusion

5.1 Remarks and Improvements

While the majority of the code for the theory-independent portion (except perhaps the DNF conversion) appears to be concise and effective, the application to SN has a number of drawbacks. In the elimination procedure, several steps were merged into a single pass: simplification of trivial factors, elimination of the quantified variable, and decrementing other variables' de Bruijn indices. This makes the procedure and accompanying correctness proof difficult to read, and masks the fact that several parts could likely be factored out into a theory-independent module which other theories could make use of. Adjustment of the interface between theory-independent and theory-specific portions, in line with Nipkow's implementation [16], could result in the elimination procedure and proof being cleaner and also more efficient than at present.

On the topic of efficiency, the handling of trivially true or false factors is sub-optimal: those that are trivially true can (and should) be removed from products, as they have no effect other than fueling future DNF explosion, and those that are false imply that the product as a whole is false and therefore equivalent to $\bot$ (the recognition of which would greatly speed up the procedure, as well as contributing less to aforementioned DNF explosion). While the decision procedure for SN has not been tested on deeply nested quantifiers, it is expected to be extremely slow (to the point of intractability) for the reasons stated above.

On a higher level, a significant improvement would be the application of the framework to a more expressive theory, such as Presburger Arithmetic. While more challenging to implement and verify, the constructive decidability of this theory has more practical uses than that of SN. Ideally, the formalization by Allais [3] could be adapted to use our framework.

Finally, as decidability is shown for the semantics of `Prop`, which are Agda-propositions, there is a good opportunity for reflection, as in Nipkow's work [16]. This would effectively give the ability to directly decide certain Agda-propositions from within the language. This possibility benefits further from the advantages of a constructive metatheory as outlined in Section 3.6.

5.2 Conclusion

In this thesis, quantifier elimination under a constructive metatheory is formalized in a theory-independent manner. Specifically, it is proven that for a given theory T in intuitionistic predicate logic, a procedure to remove a single existential quantifier (necessarily theory-specific) can be transformed into a procedure to perform full quantifier elimination on arbitrary propositions in T. This yields a decision procedure for the semantics $[\![\cdot]\!]$ of T (a fragment of Agda propositions), from which several powerful results follow:

- The law of excluded middle holds (constructively) for the semantics of T, implying equivalence to a classical formulation.
- For a proposition ϕ of suitable arity, either a witness to $\exists x.[\![\phi]\!]$ can be produced, or a proof to the contrary.
- For a proposition ϕ of suitable arity, either a proof that $[\![\phi]\!]$ holds for all x can be produced, or a counterexample.

The above is applied to the theory of successor on the natural numbers, resulting a verified decision procedure, and the properties described above.

Bibliography

[1] The Agda Wiki. (2017, September 11). *The Agda Wiki* [Online]. Available: http://wiki.portal.chalmers.se/agda/pmwiki.php

[2] The Agda Wiki. (2015, December 14). *The Agda Wiki - Tutorials* [Online]. Available: http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Main.Othertutorials

[3] G. Allais, private communication, June 2017.

[4] G. Allais. (2012, July 11). *Deciding Presburger arithmetic in agda* [Online]. Available: https://github.com/gallais/agda-presburger

[5] D. Bridges and E. Palmgren, "Constructive Mathematics", in *The Stanford Encyclopedia of Philosophy* (Winter 2016 Edition), E.N. Zalta, Ed [Online]. Available: https://plato.stanford.edu/archives/win2016/entries/mathematics-constructive/

[6] A. Church, "An Unsolvable Problem of Elementary Number Theory", in *American Journal of Mathematics*, vol. 58, no.1, Apr., pp. 345-363, 1936.

[7] C. Cohen and A. Mahboubi, "Formal Proofs in Real Algebraic Geometry: from Ordered Fields to Quantifier Elimination", in *Logical Methods in Computer Science*, vol. 8, pp.1–40, 2012. Available: https://arxiv.org/abs/1201.3731

[8] Coq developers. *The Coq Proof Assistant* [Online]. Available: https://coq.inria.fr

[9] P. Crégut. *Omega: a solver of quantifier-free problems in Presburger Arithmetic* [Online]. Available: https://coq.inria.fr/refman/omega.html

[10] J. Doner and W. Hodges, "Alfred Tarski and Decidable Theories", in *The Journal of Symbolic Logic*, vol. 53, no. 1, Mar., pp. 20–35, 1988.

[11] K. Gödel, "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I", in *Monatshefte für Mathematik und Physik*, vol. 38, pp. 173–198, 1931.

[12] J. Herbrand and W.D. Goldfarb, *Logical Writings*. Dordrecht: D. Reidel Publishing Company, 1971.

[13] J. R. Hindley. (2015). *The root-2 proof as an example of non-constructivity* [Online]. Available: http://www.users.waitrose.com/ hindley/Root2Proof2015.pdf

[14] Isabelle developers. (2017, October 27). *Isabelle* [Online]. Available: http://isabelle.in.tum.de

[15] A. Mahboubi and C. Cohen, "A Formal Quantifier Elimination for Algebraically Closed Fields", in *Intelligent Computer Mathematics*, Calculemus 2010, Paris, France, pp. 189–203, 2010.

[16] T. Nipkow. "Reflecting Quantifier Elimination for Linear Arithmetic", in *Formal Logical Methods for System Security and Correctness*, O. Grumberg, T. Nipkow, C. Pfaller. IOS Press, 2008, pp.245–266.

[17] B. Nordström, K. Petersson, J.M. Smith, *Programming in Martin-Löf's Type Theory*. Oxford University Press, 1990, pp.9–11.

[18] R. M. Robinson, "An Essentially Undecidable Axiom System", in *Proceedings of the International Congress of Mathematics*, vol. 1, pp. 729–730, 1950.

[19] K. Simmons, "Tarski's Logic", in *Logic from Russell to Church*, D. Gabbay, J. Woods, Eds. Elsevier, 2009, pp.511–616.

[20] R. Stansifer, "Presburger's Article on Integer Arithmetic: Remarks and Translation", Cornell University, 1984.

[21] A. Tarski, "A Decision Method for Elementary Algebra and Geometry", RAND Corporation, 1948.

[22] A. Tarski, "The semantic conception of truth", in *Philosophy and Phenomenological Research*, vol. 4, pp. 341–376, 1944.

[23] A.S. Troelstra, "History of Constructivism in the Twentieth Century", Univ. of Amsterdam, ITLI Prepublication Series ML-91-05, 1991.

[24] A. M. Turing, "On Computable Numbers, With An Application To The Entscheidungsproblem", in *Proceedings of the London Mathematical Society*, ser. 2, vol. 42, pp. 230–265, 1937.

[25] P. van der Walt, "Reflection in Agda", M.Sc. thesis, Universiteit Utrecht, Utrech, Netherlands, 2012.