Formalizing Constructive Quantifier Elimination in Agda

Jeremy Pope
University of Gothenburg
Gothenburg, Sweden
guspopje@student.gu.se

In this paper a constructive formalization of quantifier elimination is presented, based on a classical formalization by Tobias Nipkow. The formalization is implemented and verified in the programming language/proof assistant Agda. 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.

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 of predicate logic 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 Classical and Constructive Logic

If quantifier elimination is proven to be possible for a theory, that proof is in turn carried out in another theory, referred to as the *metatheory*.

Both the theory and metatheory can vary with respect to being classical or constructive. A classical (meta)theory, by virtue of the law of excluded middle, allows the use of quantifier dualities, De Morgan's laws, proof by contradiction, and other results of classical logic.

In a constructive (meta)theory, however, the above are no longer a priori available. Despite such constraints, there are several advantages. One of the most significant—beside any philosophical arguments for constructivism—is that a proof of existence requires a *witness*, an actual value that meets the

R. Atkey and S. Lindley (Eds.): Mathematically
Structured Functional Programming (MSFP 2018)
EPTCS 275, 2018, pp. 2–17, doi:10.4204/EPTCS.275.2

specified criteria. For example, a constructive proof of $\exists x.x > 2$ would necessarily consist of a value for x, and a proof that it is greater than two. As a consequence, not only does a constructive decision procedure determine whether a proposition is true or not, but it provides significantly more information about *how*.

1.3 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.

1.4 History, and Related Work

The technique of quantifier elimination has been used over the last century to prove the decidability of a number of theories. Notable examples include: real and algebraically closed fields, by Alfred Tarski [4]; several theories on the natural numbers under a constructive metatheory, by Jacques Herbrand [6]; and Presburger arithmetic (addition on the natural numbers), by Mojżesz Presburger [10].

Following Gödel's incompleteness theorems [5] came several negative results in decidability. One such result is the essential undecidability of Robinson arithmetic [9] (addition and multiplication on the natural numbers), which effectively rules out the possibility of a decision procedure for general arithmetic.

More recently, several (positive) results have been revisited in the context of computer-verified proofs. Examples include Tobias Nipkow's framework for and application of quantifier elimination in Isabelle [8]; Assia Mahboubi and Cyril Cohen's Coq formalizations of the decidability of real [2] and algebraically [7] closed fields; and Guillaume Allais' constructive proof of the decidability of Presburger arithmetic in Agda [1].

This paper presents a general framework somewhat similar to that of Nipkow [8], however carried out in Agda under a constructive metatheory.[1] It is applied to one of the theories Herbrand [6] showed to be decidable, using a technique similar to the one that he presented. It also bears substantial similarity to Allais's Presburger solver [1], despite being developed largely independently: correspondance between this author and Allais, during which the latter generously shared the source code of his (at the time unpublished) work, only occurred after the majority of the work presented in this paper had been completed. Certain presentational aspects were influenced by Allais' work, however. More importantly, several apparent novelties of this work—such as the trick used to obviate the need for prenex form—in fact appeared earlier in the former. The primary contribution of this work is therefore its relative generality.

[1] And, alas, without reflection for the time being.

1.5 Organization

The remainder of this paper is organized as follows: First, brief background information is given on several theoretical aspects (Section 2). Next, a theory-independent formalization of quantifier elimination is shown (Section 3), followed by an application to a theory on the natural numbers (Sections 4 and 5). Finally, possibilities for further development are discussed (Section 6).

The source code for the project (excluding the Agda standard library) is available on GitHub.[2] At the time of this paper's writing, the code compiles with Agda version 2.5.2, and version 0.13 of the standard library.

2 Theoretical Background

2.1 Quantifier Elimination

Rather than attempting to remove all quantifiers at once, an incremental approach is usually 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.[3]

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).

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

[3] This is actually quite important; transforming a proposition into prenex form uses quantifier dualities that are not valid in constructive logic.

2.2 Agda

Agda [13], the programming language/proof assistant used in this paper, is based on intuitionistic type theory. As a result it is constructive, with the consequences described in Section 1.2. Further information about the language is available from Agda's website and the various tutorials listed there [14].

2.3 Theory, Metatheory, and Semantics

In quantifier elimination, and proofs about logic systems in general, there are frequently 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 [6]. This allows a syntactic treatment, notions such as "equivalent in T" and "provable in T", and consequently strong separation between theory and metatheory.[4]

Another option, as taken by Tarski [11], Nipkow [8], and this project, is to instead define the semantics 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 paper, T is an arbitrary theory of first order logic (subject to minor constraints), and M is the flavor of Intuitionistic Type Theory used by Agda. A notable consequence of M being constructive (and the definition of $[\![\cdot]\!]$ used in this paper) is that 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 of its own). As the semantics lie in M, this means that decidability is shown for a fragment of M.

2.4 De Bruijn Indices

One of the difficulties in formalizing a theory is the handling of free and bound variables. With a traditional "named variable" approach, extra conditions must be added to prevent substitutions that would capture free variables.[5]

An alternative is to use De Bruijn indices, where each occurrence of a bound variable is denoted instead by a number that indicates how many variable-binders "deep" the occurrence is from the binder to which it refers. For example, the proposition

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

is represented as

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

Here it is noted that within the scope of the $\exists$, the index $\boxed{0}$ refers to the $\exists$ (i.e., the variable y), and $\boxed{1}$ refers to the quantifier one layer out, namely $\forall$ (i.e., the variable x).

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

[5] An example from lambda calculus is the (invalid) beta reduction of $(\lambda x.\lambda y.x)y$ to $\lambda y.y$.

Free variables can be treated the same way; their indices simply point outside of the visible formula (for example, the rightmost variable in $\exists.(\boxed{0} \geq \boxed{3})$). The benefits include dramatically simpler rules for substitution, alpha equivalence for free, and ease in associating values with free variables.

With named variables, the latter (referred to as an *environment*) is accomplished via a mapping from names to values, while with de Bruijn indices only a list of values is required—the list corresponds to the "missing" layers of quantifiers to which the free variables refer. For example, the proposition $\forall.\exists.(\boxed{4} \geq \boxed{7})$ requires a list with at least six values.

In this paper, such a bound is referred to as the *arity* of a proposition, arising from the interpretation of a proposition as a function of its free variables.

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 2.4):

$$\mathsf{Atom} : \mathbb{N} \to \mathsf{Set}$$

The internal structure of an Atom is completely unspecified.

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

$$\mathsf{Y} : \mathsf{Set}$$

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

$$[\![\_]\!]_a : \{n : \mathbb{N}\} \to \mathsf{Atom}\ n \to \mathsf{Vec}\ \mathsf{Y}\ n \to \mathsf{Set}$$

The implicit parameter $n : \mathbb{N}$ 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 2.4). 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:

$$[\![\_]\!]_a? : \{n : \mathbb{N}\}\ (a : \mathsf{Atom}\ n)\ (e : \mathsf{Vec}\ \mathsf{Y}\ n) \to \mathsf{Dec}\ ([\![\ a\ ]\!]_a\ e)$$

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 $x : \neg A$, i.e. $x : A \to \bot$).

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 Prop. Its constructors allow the formation of a proposition from an atom, or from other propositions by way of the typical connectives and quantifiers.

```
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 would be typical in a classical setting.

The quantifiers E_ and A_ reflect the use of de Bruijn indices (Section 2.4): 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).

3.3 Semantics of Propositions

The semantics of a proposition is then defined recursively from ⟦_⟧ₐ:

```
⟦_⟧ : {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 is 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 is defined similarly, but using a (dependent) function type[6] 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 ⇒ ⊥⊥
```

QFree ϕ is inhabited if and only if ϕ is quantifier-free.

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:

```
_→?_ : {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)
```

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

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
    elim   : {n : ℕ} (ϕ : Prop (suc n)) → QFree ϕ → Prop n
    qfree  : {n : ℕ} (ϕ : Prop (suc n)) (qf : QFree ϕ) → QFree (elim ϕ qf)
    equiv  : {n : ℕ} (ϕ : Prop (suc n)) (qf : QFree ϕ) (e : Vec Y n) →
      ⟦ E ϕ ⟧ e ↔ ⟦ elim ϕ qf ⟧ e
```

The field elim 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 elim does not contain the existential quantifier to eliminate, rather it is implied—for example, to eliminate the quantifier from E ϕ, the elim procedure is invoked on just ϕ. The field qfree represents a proof that elim always produces a quantifier-free proposition. Finally, equiv establishes elim's correctness—that the propositions E ϕ and elim ϕ ... are semantically equivalent.[7]

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 elim 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 ~).[8]

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 a)    = atom a
lift-qe qe (ϕ₁ ∨ ϕ₂)  = (lift-qe qe ϕ₁) ∨ (lift-qe qe ϕ₂)
```

[7] The notation $A \leftrightarrow B$ is defined as $(A \to B) \times (B \to A)$.

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

```
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.elim qe (lift-qe qe ϕ) (lift-qe-qfree qe ϕ)
lift-qe qe (A ϕ)      = ~ (QE.elim qe (~ lift-qe qe ϕ) (~-qf lift-qe-qfree qe ϕ))
```

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

3.5.1 Correctness

The correctness of the lifted procedure—that lift-qe $qe\ \phi$ 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
```

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.elim is applied to. The reasoning behind this is as follows:

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

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 $\forall x.(B(x) \Rightarrow C(x))$, 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):

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

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.elim (QE.equiv). Conceptually, this corresponds to treating $\forall$ as $\neg\exists\neg$ from the outset.

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:

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

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.[9] The correctness proof 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:

[9] While it could have been formulated to use the weaker requirement that $\neg\neg B\ a \to B\ a$, there is no particular benefit to doing so in this case.

```
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)
```

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)
```

The theory in question, whatever it may be, is thus proven decidable.

3.6.1 Consequences

With decidability, the law of excluded middle can be proven for the semantics of Prop:

```
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⟧
```

Using the law of excluded middle, the hitherto unavailable classical results become provable—with all of the benefits of a constructive metatheory (see Section 1.2). For example, for any proposition ϕ, $(\forall x.\phi) \vee (\exists x.\neg\phi)$ is provable:

```
∀-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)))
```

While trivial in a classical setting, in a constructive setting this produces either *(i)* a proof that ϕ is true for every x, or *(ii)* a counterexample—a value for x which causes ϕ to be false (and a proof thereof). Analogous results are obtained for the theorem $(\exists x.\phi) \vee (\forall x.\neg\phi)$.

3.7 Disjunctive Normal Form and Products

The most basic formulation of a single-step elimination procedure (QE) 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 works of Herbrand [6], Nipkow [8], and Allais [1].

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. The utility of this form is that since existential quantification distributes across disjunction, the problem of quantifier elimination on DNF formulae trivially reduces to quantifier elimination on conjunctions of literals:

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

The mechanics and correctness of the conversion to DNF are not discussed here, as they do not differ significantly from under a classical metatheory.[10] The results are a function dnf that transforms any proposition into DNF, and proof of its correctness:

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

The DNF datatype employed above is a list of lists of literals, which can be "interpreted" as a proposition using the function D.i.[11]

Single-step elimination on propositions in DNF is defined much like the more general QE:

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

Single-step elimination on conjunctions of literals (referred to here as *products*) is also defined:

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

The Prod datatype is shorthand for a list of literals, and, analogously to DNF, is interpreted as a proposition using P.i.

The distribution of ∃ across disjunctions allows single-step elimination on products to be trivially generalized to single-step elimination on propositions in DNF (proof omitted):

```
Prod⇒DNF.lift : ProdQE → DNFQE
```

The conversion of any quantifier-free proposition to DNF (dnf, dnf-fwd, dnf-bwd above) allows single-step elimination on propositions in DNF to be generalized to single-step elimination on any quantifier-free proposition (proof omitted):

```
lift-dnf-qe : DNFQE → QE
```

[10] Recall that the semantics of an atom is decidable, so De Morgan's laws hold.

[11] An alternative approach is to use an actual Prop, along with a proof that it is in disjunctive normal form, but this can make manipulation cumbersome.

Finally, composing these two functions allows single-step elimination on products to be generalized to single-step elimination on any quantifier-free proposition:

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

The scope of the problem is therefore narrowed considerably: to obtain full quantifier elimination and decidability (recalling the results of Sections 3.5 and 3.6, lift-qe and ⟦_⟧? in particular), one need only create a single-step quantifier elimination procedure that acts on products. This can be put to direct use on a number of theories, as described by Nipkow [8]. The following section gives an overview of one such application.

4 The Theory of Successor

The example theory to which this framework is applied is the theory of successor on the natural numbers ("SN"), as presented by Herbrand [6]. Atomic formulae are equalities between terms, each of which is the application of the successor function S some (known) number of times to either a variable or zero. For example:

$$S(S(S(x))) = S(S(S(S(0))))$$

or, more concisely:

$$S^3(x) = S^4(0).$$

For convenience the previous atomic formula may be written as $x+3=4$, but it is important to note that—unlike in the case of Presburger arithmetic—one may not add variables together, as this would represent an unknown number of applications of S. Thus $x+3=y+7$ corresponds to a valid atomic formula, while $x+y=z+5$ does not.

The machinery developed in Section 3 allows the scope of quantifier elimination to be reduced to elimination of a single variable x from a conjunction of literals (equalities or negated equalities, i.e. inequalities). For this theory, that can be accomplished via substitution; if an equality involving x is found, such as $x+5=y+3$, then substitutions are made accordingly throughout the conjunction, thereby removing x, and the inequalities $y \neq 0$ and $y \neq 1$ are added. On the other hand, if x only occurs in inequalities, then those inequalities may be dropped from the conjunction—each one is only false for at most one value of x, so there is always some value for x which satisfies them all.[12]

The (rather verbose) formalization of the above, omitted from this paper in the interest of brevity, results in a ProdQE object. This is then lifted to a QE object via lift-prod-qe, leading to full quantifier elimination via lift-qe and decidability via ⟦_⟧?.

5 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. The syntax for SN propositions leaves much to be desired; what they code for is indicated with comments. First, a simple system of equalities:

[12] This excludes trivial inequalities such as $x+3 \neq x+3$ and $x+2 \neq x+4$, which are simplified instead of dropped.

```
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 ⟦ *test*$_0$ ⟧? [] yields:[13]

```
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:

```
- ∀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)))))
```

⟦ *test*$_1$ ⟧? [] 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 test$_2$. As test$_2$ is not true for all values, a counterexample is produced instead: inj$_2$ (1 , ...), 1 being a value for which the proposition does not hold, and ... consisting of a trivial proof that $1 \neq 0$ and a lengthy proof that no y exists such that $1 = 2 + y$.

6 Future Work

While the code for the "core" of the theory-independent portion is relatively well-organized, the same cannot be said for the DNF-conversion portion. The handling of trivially true or false factors in the latter is also suboptimal: those that are trivially true can (and should) be removed from products, as they have no effect other than fueling 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 the aforementioned DNF explosion).

Another improvement would be the application of the framework to more expressive theories, such as Presburger arithmetic or any of the other theories discussed in Section 1.4. In the former case, it may be possible to adapt Allais's proof [1].

A final area of future work noted here is the implemention of *reflection*, where propositions in the metatheory can be manipulated directly (or, seemingly so). This would allow quantifier elimination to

[13] Technically, each refl appeared as .Agda.Builtin.Equality._≡_.refl.

be applied to suitable Agda propositions directly, without the need to first encode them as Prop. Such a strategy is employed in Nipkow's framework [8] in Isabelle, and the Omega solver [3] provides similar functionality for Presburger arithmetic in Coq (albeit as a tactic, rather than through reflection). Agda's reflection mechanism is discussed in a more general context by van der Walt [12].

7 Acknowledgements

This paper is adapted from the author's 2018 master's thesis in Computer Science at the University of Gothenburg. The idea for the project came from the author's thesis supervisors Thierry Coquand and Simon Huber, with additional help and encouragement from Andreas Abel.

References

[1] G. Allais (2018): *Deciding Presburger arithmetic in agda*. Available at `https://github.com/gallais/agda-presburger`.

[2] C. Cohen & A. Mahboubi (2012): *Formal Prrofs in Real Algebraic Geometry: from Ordered Fields to Quantifier Elimination*. *Logical Methods in Computer Science* 8, pp. 1–40, doi:10.2168/LMCS-8(1:2)2012.

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

[4] J. Doner & W. Hodges (1988): *Alfred Tarski and Decidable Theories*. *The Journal of Symbolic Logic* 53(1), pp. 20–35, doi:10.1017/S0022481200028905.

[5] K. Gödel (1931): *Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I*. *Monatshefte für Mathematik und Physik* 38, pp. 173–198, doi:10.1007/BF01700692.

[6] J. Herbrand (1971): *Logical Writings*. D. Reidel Publishing Company, Dordrecht, Netherlands, doi:10.1007/978-94-010-3072-4.

[7] A. Mahboubi & C. Cohen (2010): *A Formal Quantifier Elimination for Algebraically Closed Fields*. In: *Intelligent Computer Mathematics*, Calculemus 2010, Paris, France, pp. 189–203, doi:10.1007/978-3-642-14128-7_17.

[8] T. Nipkow (2010): *Linear Quantifier Elimination*. *Journal of Automated Reasoning* 45, pp. 189–212, doi:10.1007/s10817-010-9183-0.

[9] R. M. Robinson (1950): *An Essentially Undecidable Axiom System*. In: *Proceedings of the International Congress of Mathematics*, 1, pp. 729–730.

[10] R. Stansifer (1984): *Presburger's Article on Integer Arithmetic: Remarks and Translation*. Technical Report TR84-639, Cornell University, Computer Science Department. Available at `https://cs.fit.edu/~ryan/papers/presburger.pdf`.

[11] A. Tarski (1944): *The semantic conception of truth*. *Philosophy and Phenomenological Research* 4, pp. 341–376, doi:10.2307/2102968.

[12] P. van der Walt & W. Swierstra (2012): *Engineering Proof by Reflection in Agda*. In Ralf Hinze, editor: *IFL - 24th International Symposium on Implementation and Application of Functional Languages*, *Lecture Notes in Computer Science* 8241, Springer, Oxford, United Kingdom, pp. 157–173, doi:10.1007/978-3-642-41582-1_10.

[13] The Agda Wiki (2017): *The Agda Wiki*. Available at `http://wiki.portal.chalmers.se/agda/pmwiki.php`.

[14] The Agda Wiki (2017): *The Agda Wiki - Tutorials*. Available at `http://wiki.portal.chalmers.se/agda/pmwiki.php`.