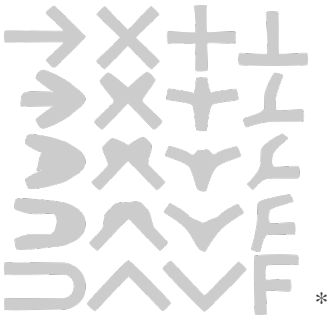
Submitted for the Degree of B.Sc. in Computer Science, 2017-2018

Evidence-providing problem solvers in Agda



*

201434138
Uma Zalakain

Supervised by
Conor McBride

Except where explicitly stated all the work in this report, including appendices, is my own and was carried out during my final year. It has not been submitted for assessment in any other context. I agree to this material being made available in whole or in part to benefit the education of future students.

*The Curry-Howard homeomorphism, by Luca Cardelli, adapted by Iune Trecet

Abstract

The Curry-Howard correspondence draws a direct link between logic and computation: propositions are modelled as types and proofs as programs; to prove a proposition is to construct a program inhabiting its corresponding type. Several computer-assisted theorem provers have been developed under this idea. They are not just used to verify human reasoning: they are also often capable of generating proofs automatically.

This project considers the development of such automated theorem provers in Agda, a dependently typed programming language. As a warm-up, I present a verified solver for equations on monoids. Then, I comment on the solver for commutative rings included in Agda's standard library. Finally, I develop a verified decision procedure for Presburger arithmetic — a decidable first-order predicate logic.

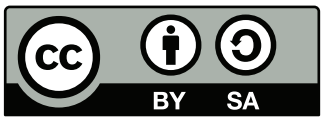
Acknowledgements

This project is an attempt to distill all the support, attention, knowledge, dedication and love I was given into concrete ideas in printable format. Despite the disclaimer saying otherwise, this project is far from being just my own work. At least a dozen people have contributed to it, either unknowingly, directly, or by contributing to my well-being.

My supervisor has been a key figure, first as the lecturer of the 12 week introduction to Agda I was lucky to receive, then as a supervisor who has a keen interest in the subject and is willing to share it. This project was the perfect excuse for countless hours of education.

Frustration, loneliness, self-doubt, dysphoria and my neighbour's barking dog have played the role of creatures of the darkness, and deserve to be acknowledged too. Luckily for me, my friends, both local and remote, and my parents, on the other side of this planet, gave me enough resolution to shed some bright light on them. My neighbour's dog, along with their owner, has finally moved away.

Needless to say, this project, of little importance to anyone but me, is based on large amounts of previous science and countless hours of accumulated human effort. To all those people who have kept the candle burning, I am forever grateful.



Contents

1. Introduction

Formal proofs construct theorems by applying the rules of a formal system. Computers can assist this process and make theorem proving a conversation between the human and the computer, which checks the correctness of their proof. Yet, theorem proving can often be boring and tedious: certain theorems are trivial or uninteresting but require many steps.

It is in these cases where automated theorem proving shines strongest: instead of applying inference rules manually, the user can provide an automated solver with a proposition and get a verified solution back. These decision procedures are often based on some meta-theory about the system, and thus can result in fewer rewriting steps than the repeated application of inference rules from inside the system.

The four color theorem was the first notable problem to be solved with the help of a computer program. Since 1976, there remained doubts of the correctness of such program until Georges Gonthier used a proof assistant to prove the theorem in 2005.

This project embarks upon constructing verified problem solvers. Three different problems are considered: the first two involve solving equalities on algebraic structures; the third deciding a first-order predicate logic — Presburger arithmetic. The aim is to better understand automated theorem proving as seen through the Curry-Howard lens.

§ 2 provides a brief introduction to the relationship between machine programs and formal proofs, illustrated with accompainying Agda programs. It also includes a short introduction to programming in Agda, and establishes some of the base ground required for the formal verification of programs.

§ 3 starts with a simple example: a verified solver for equations on monoids. § 4 comments on a more involved solver for commutative rings found in Agda's standard library. This project then culminates with § 5, where the heart of a Presburger arithmetic solver written in Agda is presented. With some additional work, I am optimistic of its inclusion into Agda's standard library.

Concluding, § 6 reiterates on the correctness that the precision of dependently typed specifications are able to guarantee, and § 7 and § 8 contain meta-analyses of the project's development process.

2. Background

This chapter starts by briefly introducing the case for the use of type-checkers as theorem verifiers. Next, a succinct primer on programming in Agda is given. In itself, such introduction is probably not enough to get the unexperienced reader entirely comfortable reading Agda code. Only more in-depth reading and hands-on practice are likely to achieve that. Nevertheless, it is my hope that it facilitates enough understanding to intuitively grasp some of the ideas put forward in later sections of this report.

2.1. Proofs as programs; propositions as types

If a computer is to verify the proof of some proposition, some computational model relating proofs and propositions must exist. One such model was first devised by Haskell Curry [Curry, 1934] and later strengthened by William Alvin Howard [Howard, 1980]. It establishes a two way correspondence between type theory and constructive logic: propositions are isomorphic to types and proofs are to programs; to prove a proposition is to construct a program inhabiting its corresponding type; a proposition is not proven unless a program of the corresponding type is given. Type-checkers can verify formal proofs.

Ignoring — for now — all details specific to Agda, here are some examples relating types to logical propositions:

```
-- Truth: a set with a single element trivial to construct
data ⊤ : Set where
  tt : ⊤

-- Falsehood: an uninhabited (empty) set
data ⊥ : Set where

-- Disjunction
data _⊎_ (A B : Set) : Set where
  inj₁ : A → A ⊎ B
  inj₂ : B → A ⊎ B

-- Conjunction
data _×_ (A B : Set) : Set where
  _,_ : A → B → A × B

module Laws {A : Set} where
```

```
-- Principle of explosion
-- There is no constructor for ⊥, pattern matching on the
-- argument renders the case absurd
explosion : ⊥ → A
explosion ()

-- Law of non-contradiction
-- AKA implication elimination
-- AKA function application
lnc : A → (A → ⊥) → ⊥
lnc a a→⊥ = a→⊥ a

-- No proof by contradiction in constructive mathematics
-- A witness in A is needed, but there is none
dne : ((A → ⊥) → ⊥) → A
dne f = {!!}

-- No law of excluded middle in constructive mathematics
-- (Decidability is not universal)
lem : A ⊎ (A → ⊥)
lem = {!!}
```

Many variants exist on both sides of the isomorphism. The type theory of simply typed lambda calculus — where → is the only type constructor — is in itself enough to model propositional logic. Type theories with dependent types — where the definition of a type may depend on a value — model predicate logics that contain quantifiers. [Sørensen and Urzyczyn, 2006] is a comprehensive introduction to these ideas.

```
-- Natural numbers, defined inductively
data ℕ : Set where
  zero : ℕ
  suc : ℕ → ℕ

-- A predicate, or a type that depends on a value
Even : ℕ → Set
Even zero = ⊤
Even (suc zero) = ⊥
Even (suc (suc n)) = Even n

-- The type of t depends on the value n
half : (n : ℕ) → (t : Even n) → ℕ
half zero tt = zero
half (suc zero) () -- Even (suc zero) is empty
half (suc (suc n)) t = suc (half n t)
```

Proofs should not suffer from the halting problem — they must be rejected if they do not clearly show that they will eventually reach termination. If programs are considered to be proofs, programs for which termination cannot be verified must be rejected.

One way of showing termination is by making recursive calls on structurally smaller arguments. If the data is defined inductively, this assures that a base case is always eventually reached, and that therefore recursion always eventually terminates.

```
_+_ : ℕ → ℕ → ℕ
zero + m = m -- Base case of first argument
suc n + m = suc (n + m) -- First argument gets smaller

-- Would never terminate
-- nonsense : {!!}
-- nonsense = nonsense
```

2.2. Reasoning in Agda

Agda is a **purely functional** (no side-effects) **dependently typed** (types contain values) **totally defined** (functions must terminate and be defined for every possible case) language based on Per Martin-Löf's intuitionistic type theory. It was first developed by Catarina Coquand in 1999 and later rewriten by Ulf Norell in 2007. It compiles to multiple languages, but Haskell is regarded as its main backend. [Norell and Chapman, 2009] is an excellent introduction to Agda; technical documentation can be found at https://agda.readthedocs.io. This section briefly covers the basics of what theorem proving in Agda looks like and, in the spirit of a tutorial, occasionally uses the second person to avoid verbose references to some third person programmer or the excessive use of the passive voice.

2.2.1. Hole-driven development

Development in Agda happens inside Emacs, and is a two way conversation between the compiler and you. Wherever a definition is required, you may instead write `?` and request the type-checker to reload the file. A "hole" will appear where the `?` was. You can then:

- examine the type of that goal;
- examine the types of the values in context;
- examine the type of any other arbitrary expression;
- pattern match on a type;
- supply a value, possibly containing further holes;
- attempt to refine the goal; or
- attempt to solve the goal automatically.

This interactive way of programming is often described as "hole driven". Type-checking definitions before writing them down promotes the construction of well-formed expressions — instead of the construction and subsequent debugging of malformed ones. Allowing holes in those definitions makes the development model realistic.

2.2.2. Some peculiarities

For subsequent arguments to depend on it, an argument must be named. If an argument can be inferred by the type-checker, you may choose to make it implicit by naming it inside enclosing curly braces. Implicit arguments can later still be explicitly provided and pattern matched against. If the type of an argument can be inferred by the type-checker, you may omit it and use ∀:

```
-- All numbers are even or not even
prf₁ : ∀ {n} → Even n ⊎ (Even n → ⊥)
prf₁ {zero} = inj₁ tt
prf₁ {suc zero} = inj₂ (λ b → b)
prf₁ {suc (suc n)} = prf₁ {n}
```

Multiple arguments sharing the same type can be grouped by using multiple names for them. With the exception of whitespace and a few other special symbols, names in Agda may contain arbitrary unicode symbols. In addition, function names can use the so-called "misfix" notation, where underscores are used as placeholders that determine where the function's arguments are placed.

```
|_−_| : (x y : ℕ) → ℕ
| zero − y | = y
| suc x − zero | = suc x
| suc x − suc y | = | x − y | -- Or |_-_| x y
```

An anonymous function can be provided wherever a function is expected. You can pattern match against its arguments by wrapping the arguments and body in curly brances.

```
pred : ℕ → ℕ
pred = λ { zero → zero
         ; (suc n) → n
         }
```

2.2.3. Datatypes and pattern matching

Algebraic data types are introduced by the `data` keyword. They may contain multiple constructors, all of which must be of the declared type.

```
data Bool : Set where
  true : Bool
  false : Bool
```

Constructors can accept arguments, which may be recursive:

```
data Bools : Set where
  [] : Bools
  _::_ : Bool → Bools → Bools
```

Datatypes may accept parameters. If they do, every constructor in the datatype has to have that same parameter in its return type. Hence these parameters need to be named:

```
data List (A : Set) : Set where
  [] : List A
  _::_ : A → List A → List A
```

Data types with a single constructor can be defined as records. Below, a record type where the type of one of the fields depends on the value of the other:

```
record Σ (A : Set) (B : A → Set) : Set where
  constructor _,_
  field
    proj₁ : A
    proj₂ : B proj₁
```

Datatypes can be indexed. Each of these indices is said to introduce a family of types. Constructors have the liberty to choose any index for the type they are constructing. While parameters must remain unaltered by constructors, indices must not.

```
-- Parametrised by A : Set, indexed by ℕ
data Vec (A : Set) : ℕ → Set where
  [] : Vec A zero
  _::_ : ∀ {n} → A → Vec A n → Vec A (suc n)
```

Pattern matching deconstructs a type, which creates one case for each constructor capable of constructing that type:

```
-- Both constructors match Vec A n
map : {A B : Set}{n : ℕ} → (A → B) → Vec A n → Vec B n
map f [] = []
map f (x :: xs) = f x :: map f xs

-- Only _::_ matches Vec A (suc n)
head : {A : Set}{n : ℕ} → Vec A (suc n) → A
head (x :: xs) = x
```

Computation is advanced by pattern matching. The right hand side of each pattern match case will have the type of the terms in its context refined by the information obtained from the left hand side.

```
-- Note that xs, ys and the result have the same length
zipWith : {A B C : Set}{n : ℕ} (f : A → B → C) → Vec A n → Vec B n → Vec C n
-- zipWith f xs ys = {!!}
-- -- If xs was constructed with [], it has length zero
-- zipWith f [] ys = {!!}
```

```
-- -- If xs has length zero, so does ys
-- zipWith f [] [] = {!!}
-- -- And so does the result
zipWith f [] [] = []
-- -- If xs was constructed with _::_, it has length (suc n)
-- zipWith f (x :: xs) ys = {!!}
-- -- If xs has length (suc n), so does ys
-- zipWith f (x :: xs) (y :: ys) = {!!}
-- -- And so does the result
-- zipWith f (x :: xs) (y :: ys) = {!!} :: {!!}
zipWith f (x :: xs) (y :: ys) = f x y :: zipWith f xs ys
```

If a type has no constructors capable of constructing it, the type-checker will recognise the case as absurd and no definition will be required on the RHS. This, together with the precision that dependent types grant, makes handling erroneous input unnecessary.

```
-- The successor of an even number cannot be even
prf₂ : ∀ {n} → Even n → Even (suc n) → ⊥
prf₂ {zero} p ()
prf₂ {suc zero} () sp
prf₂ {suc (suc n)} p sp = prf₂ {n} p sp
```

If pattern matching against a type uniquely implies the constructor of some other argument, the type-checker will substitute the argument by the value preceded by a dot. If a term on the RHS can be inferred by the type-checker, you may replace it by an underscore. Additionally, underscores can be used as a non-binding catch-all pattern on the LHS of a definition.

```
-- Pattern matching on xs determines n
zipWith′ : {A B C : Set} (n : ℕ) (f : A → B → C) → Vec A n → Vec B n → Vec C n
zipWith′ .zero f [] [] = []
zipWith′ .(suc _) f (x :: xs) (y :: ys) = f x y :: zipWith′ _ f xs ys
```

“With abstraction” allows you to pattern match on the LHS against arbitrary computations. This is often used to refine the rest of the arguments and then perform further pattern matching on them. The following example is adapted from the standard library and was originally presented in [McBride and McKinna, 2004]:

```
-- Ordering n m is a proof…
data Ordering : ℕ → ℕ → Set where
  less    : ∀ m k → Ordering m (suc (m + k))
  equal   : ∀ m → Ordering m m
  greater : ∀ m k → Ordering (suc (m + k)) m

-- …that can be generated for any two numbers
compare : ∀ m n → Ordering m n
```

```
compare zero zero     = equal zero
compare (suc m) zero = greater zero m
compare zero (suc n) = less    zero n
compare (suc m) (suc n) with compare m n
compare (suc .m) (suc .(suc m + k)) | less    m k = less (suc m) k
compare (suc .m) (suc .m)           | equal m = equal (suc m)
compare (suc .(suc m + k)) (suc .m) | greater m k = greater (suc m) k
```

Pattern matching on `compare m n` uniquely defines `m` and `n`. This is the key difference between with abstraction and ordinary case splitting on the RHS. [Oury and Swierstra, 2008] contains other interesting examples of views.

2.2.4. Intensional equality

Intensional equality judges two terms equal based on how they were constructed. Two terms with identical behaviour but of different construction are considered different.

```
data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x
```

In `x ≡ y`, `x` is the parameter and `y` the index. The single constructor `refl` constructs types where the parameter `x` is provided as the index too. This means that for `x ≡ y` to be well-formed, Agda has to be able to unify `x` and `y`: once both terms are normalised into a tree of constructors, they must be syntactically equal.

```
-- Both sides normalise to suc (suc zero)
prf₃ : (suc zero + suc zero) ≡ suc (suc zero)
prf₃ = refl
```

You can now start writing functions that compute proofs involving equality:

```
-- zero + n immediately normalises to n
prf₄ : ∀ n → (zero + n) ≡ n
prf₄ n = refl
```

However, not all equations immediately unify. Consider the following:

```
prf₅ : ∀ n → (n + zero) ≡ n
```

`n + zero` cannot be normalised: as a consequence of the definition of `_+_`, it needs to be known whether `n` was constructed with `zero` or `suc`. The computation can be advanced by pattern matching on `n`. While the base case is now trivial (`zero + zero` unifies with `zero`), the problem persists in the inductive case, where `suc (n + zero)` has to unify with `suc n`. By recursing on the inductive hypothesis and on the subject of such hypothesis, `n + zero` and `n` can be unified:

```
prf₅ zero = refl
prf₅ (suc n) with n + zero | prf₅ n
prf₅ (suc n) | .n | refl = refl
```

This recursion on the induction hypothesis is common enough that special syntax exists for it:

```
prf₆ : ∀ n → (n + zero) ≡ n
prf₆ zero = refl
prf₆ (suc n) rewrite prf₆ n = refl
```

2.2.5. The inspect idiom

The so-called inspect idiom is occasionally used throughout the present work and deserves to be briefly mentioned. When you pattern match on some expression `e`, a case will be generated for every constructor `c` capable of constructing such an expression `e`. On the RHS of each of these cases, it is clear that `c ≡ e`. You might want to pattern match on this proof or hand it over to other functions. However, the case split does not add such a proof to context.

In such cases, the inspect idiom can be used to "remember" the result of pattern matching:

```
f n m with pred n | inspect pred n
f n m | zero   | [ eq ] = {!!} -- eq : pred n ≡ zero
f n m | suc n′ | [ eq ] = {!!} -- eq : pred n ≡ suc n′
```

2.2.6. Tools for reasoning

To aid reasoning, tools that enable top-down whiteboard-style deductions have been developed. These functions exploit the transitivity of the binary relation they are defined for — may be it equality or another preorder relation like ≤ or ⇒. Compared to the bare application of transitivity, this style of reasoning leaves a clear "trail" of interleaving types and justifications for their relation. Together with the congruent property of functions, it is used extensively throughout this work.

```
cong : ∀ {A B : Set} (f : A → B) {x y} → x ≡ y → f x ≡ f y
cong f refl = refl


prf₇ : ∀ l n m → ((zero + (l + zero)) + (n + zero)) + m ≡ (l + n) + m
prf₇ l n m = begin
  -- LHS of the equality
  ((zero + (l + zero)) + (n + zero)) + m
    ≡⟨⟩ -- The rewrite needs no justification, both types unify
  ((l + zero) + (n + zero)) + m
    ≡⟨ cong (λ ● → (● + (n + zero)) + m) (prf₆ l) ⟩
```

```
(l + (n + zero)) + m
  ≡⟨ cong (λ ● → (l + ●) + m) (prf₆ n) ⟩
(l + n) + m
-- RHS of the equality
  ■
```

2.2.7. Proof by reflection

Procedures that try to automatically solve goals require some notion of what their target theorem is. To prove the goal within Agda, this notion has to be manipulated and inspected by pattern matching. To do so, it needs to be translated into an inductive data type — a process often called *metaification* or *reflection*. Both [Grégoire and Mahboubi, 2005] and [Boutin, 1997] introduce this idea.

This is in contrast with proof assistants like Coq, which often supply externally defined "tactics": in Agda, automated theorem provers must be defined within the system.

The support for reflection that Agda offers gives the programmer the ability to "quote" arbitrary parts of the program into abstract terms representing them. In the other direction, these abstract terms can be procedurally built and later "unquoted" into concrete Agda code. Additionally, Agda also offers means to directly control type checking and unification.

Agda's reflection mechanism is most commonly used to satisfy proof goals automatically. For this common use case, Agda provides "macros": functions that take their target quoted goal as an argument and hand back some computation that solves it.

The next example from Agda's documentation shows how the macro `by-magic` uses `magic` to construct values of a given type. Note that `magic` returns a `Term` inside a `TC` monad: this allows `magic` to throw type errors with custom error messages.

```
postulate magic : Type → TC Term

macro
  by-magic : Term → TC ⊤
  by-magic hole =
    bindTC (inferType hole) λ goal →
    bindTC (magic goal) λ solution →
    unify hole solution
```

Both [van der Walt, 2012] and [van der Walt and Swierstra, 2012] are in-depth introductions to Agda's reflection mechanism and come supplemented with examples. [Kokke and Swierstra, 2015] uses reflection to, given a list of hints, conduct automatic first-order proof search on a goal type.

2.2.8. Builtins, Stdlib and Prelude

Agda is distributed together with a set of builtin data types and functions found under the `Agda.Builtin` module. These builtins are then referenced by a set of directives (or *pragmas*), so

that Agda can, for instance, translate numerical literals into terms of type $\mathbb{N}$. Agda.Builtin does not provide any proofs of the properties related to these data types.

The development of Agda-Stdlib happens in close coordination to Agda's. Unlike Agda.Builtin's conservative approach, Agda-Stdlib provides a large library of commonly used data structures and functions. It abstracts aggressively which, together with its heavy use of unicode symbols and infix notation, can often result in code challenging to read for the inexperienced user. It contains a rather vast set of already proven theorems for all of its data types.

In comparison, Agda-Prelude is less abstract and more readable and efficient, but by far not as complete.

This project uses Agda-Stdlib as its sole dependency.

2.2.9. Miscellanea

Universes

To avoid Russell's paradox, Agda introduces a hierarchy of universes $\mathtt{Set} : \mathtt{Set}_1 : \mathtt{Set}_2 \ldots$ where Set is the type of all small types like Bool or $\mathbb{N}$.

Postulates and safe mode

In Agda, any proposition can be introduced as a postulate. Some postulates can lead to inconsistencies:

```
postulate ¬LEM : {A : Set} → A ⊎ (A → ⊥) → ⊥
LEM : {A : Set} → A ⊎ (A → ⊥)
LEM with ¬LEM (inj₁ tt)
LEM | ()
```

Executing Agda with the `--safe` switch deactivates those features that may lead to inconsistencies, like postulates, accepting unsolved proofs or $\mathtt{Set} : \mathtt{Set}$. Unfortunately, Agda's standard library does not quarantine unsafe definitions, so any module that depends on it is considered unsafe too — even if does not use any of its unsafe features. There is work in progress to address this.

2.3. Problem solvers and their domains

This report presents evidence providing problem solvers for three distinct domains — namely monoids, commutative rings, and Presburger arithmetic. The background and the work related to each of these domains is relevant only to itself. For that reason, and because I judge it beneficial to have those introductory sections close to the work that depends on them, I present the background of each problem inside its dedicated chapter.

3. Solving monoids

Monoids are common algebraic structures found in many problems. A monoid solver is an procedure that automatically generates a proof of the equality of two monoids. Constructing such a solver is a good first approach to proof automation: it lacks the complexity of many other problems but still has their same high-level structure.

3.1. Problem description and specification

Agda-Stdlib's definition of a monoid is based on notions about many other algebraic structures, and is therefore fairly complex. Instead, I present a self-contained and fairly simple definition:

```
-- A monoid is a set
record Monoid (M : Set) : Set where
  infixl 25 _·_
  field
    -- Together with an associative binary operation
    _·_ : M → M → M
    law-·-· : (x y z : M) → (x · y) · z ≡ x · (y · z)
    -- And a neutral element absorbed on both sides
    ε : M
    law-ε-· : (m : M) → ε · m ≡ m
    law-·-ε : (m : M) → m ≡ m · ε
```

M, the set on which the monoid is defined, is often referred to as the carrier. $(\mathbb{N},+,0)$ and $(\mathbb{N},\cdot,1)$ are both examples of monoids. These examples also happen to be commutative, while monoids need not be — more on solving commutative monoids later. Lists together with the concatenation operation form non-commutative monoids.

```
LIST-MONOID : (T : Set) → Monoid (List T)
LIST-MONOID T = record
                  { ε = []
                  ; _·_ = _++_
                  ; law-ε-· = λ xs → refl
                  ; law-·-ε = right-[]
                  ; law-·-· = assoc
                  } where

                  right-[] : (xs : List T) → xs ≡ xs ++ []
```

```
right−[] [] = refl
right−[] (x :: xs) = cong (x ::_) (right−[] xs)

assoc : (xs ys zs : List T) → (xs ++ ys) ++ zs ≡ xs ++ (ys ++ zs)
assoc [] ys zs = refl
assoc (x :: xs) ys zs rewrite assoc xs ys zs = refl
```

An equation on monoids cannot be decided by unification alone: the monoid laws show that definitionally distinct propositions might in fact have the same meaning.

```
eqn₁ : {T : Set}(xs : List T) → [] ++ xs ≡ xs ++ []
eqn₁ {T} xs = begin
  [] ++ xs
    ≡⟨ law−ε−· xs ⟩
  xs
    ≡⟨ law−·−ε xs ⟩
  xs ++ []
    ■
  where
  open ≡−Reasoning
  open Monoid (LIST−MONOID T)
```

Without an automated solver, the number of law applications and hence the length of the proof grows with respect to the size of the term. An automated solver should allow to effortlessly satisfy a proposition like the following:

```
eqn₂ : {T : Set}(xs ys zs : List T)
      → (xs ++ []) ++ ([] ++ (ys ++ (ys ++ zs)))
      ≡ xs ++ ((ys ++ ys) ++ (zs ++ []))

eqn₂ xs ys zs = begin
  (xs ++ []) ++ ([] ++ (ys ++ (ys ++ zs)))
    ≡⟨ cong (_++ ([] ++ (ys ++ (ys ++ zs)))) (sym (law−·−ε xs)) ⟩
  xs ++ ([] ++ (ys ++ (ys ++ zs)))
    ≡⟨ cong (xs ++_) (law−ε−· (ys ++ (ys ++ zs))) ⟩
  xs ++ (ys ++ (ys ++ zs))
    ≡⟨ cong (xs ++_) (sym (law−·−· ys ys zs)) ⟩
  xs ++ ((ys ++ ys) ++ zs)
    ≡⟨ cong (λ zs′ → xs ++ ((ys ++ ys) ++ zs′)) (law−·−ε _) ⟩
  xs ++ ((ys ++ ys) ++ (zs ++ []))
    ■
  where
  open ≡−Reasoning
  open Monoid (LIST−MONOID _)
```

3.2. A verified decision procedure

A proposition containing variables and monoid operators can be normalised into a canonical form. The characteristics that make two propositions definitionally distinct — when they are equal modulo the monoid axioms — can be eliminated. It is crucial that this process — normalisation into a canonical form — guarantees the preservation of meaning. After normalisation, two results can be compared: if they are equal, the original expressions must be too. This is the sketch of the decision procedure.

I use an abstract syntax tree to represent equations, and finite indices to refer to variables — the type `Fin n` contains `n` distinct values. Moreover, I use a type parameter on `Eqn` to *push in* this limitation on the number of indices.

```
data Expr (n : ℕ) : Set where
  var′ : Fin n → Expr n
  ε′ : Expr n
  _·′_ : Expr n → Expr n → Expr n

data Eqn (n : ℕ) : Set where
  _≡′_ : Expr n → Expr n → Eqn n
```

Consider the following two expressions:

$$P = ((\varepsilon \cdot x) \cdot (x \cdot y)) \qquad Q = ((x \cdot x) \cdot y)$$

Neutral elements do not have any meaning and can be absorbed:

$$P = (x \cdot (x \cdot y)) \qquad Q = ((x \cdot x) \cdot y)$$

Elements can always be re-associated: association does not have any meaning and can be removed:

$$P = x \cdot x \cdot y \qquad Q = x \cdot x \cdot y$$

Both propositions can now be seen to be equal. It is worth remembering that these are not commutative monoids, and that thus the order of the elements can matter.

Lists are a suitable data structure for representing flat elements — indices here — that can appear multiple times and whose order carries meaning. In the case of commutative monoids, where order does not carry any meaning, a matrix of indices and the number of occurrences of each could be represented as a vector of integers — where the position in the vector represents the index and the content represents the number of occurrences.

```
NormalForm : ℕ → Set
NormalForm n = List (Fin n)
```

The normalising function ignores neutral elements and preserves order:

```
normalise : ∀ {n} → Expr n → NormalForm n
normalise (var′ i) = i :: []
normalise ε′       = []
normalise (e₁ ·′ e₂) = normalise e₁ ++ normalise e₂
```

From here on, I work with a concrete monoid (monoid) on a concrete carrier M. This results in all of the definitions inside of the module having M and monoid defined. When called from the outside of this module, these definitions have {M : Set} (monoid : Monoid M) prepended to their type. I also make the definitions in monoid directly accessible by opening it as if it were a module.

```
module _ {M : Set} (monoid : Monoid M) where
  open Monoid monoid
```

To evaluate an expression, a concrete assignment for the variables contained within is needed. This is often called an environment. An environment is a lookup table where each of the indices has an associated value in the carrier M.

```
Env : ℕ → Set
Env n = Vec M n
```

Now that expressions, normal forms and environments are defined, their evaluation can be defined too. Note that both definitions rule out expressions and normal forms with more indices than the environment contains — every index in the expression has to have a corresponding value in the environment.

```
-- lookup x ρ ≔ value at index x in ρ
⟦_⟧ : ∀ {n} → Expr n → Env n → M
⟦ var′ i ⟧ ρ = lookup i ρ
⟦ ε′ ⟧     ρ = ε
⟦ e₁ ·′ e₂ ⟧ ρ = ⟦ e₁ ⟧ ρ · ⟦ e₂ ⟧ ρ

⟦_⇓⟧ : ∀ {n} → NormalForm n → Env n → M
⟦ [] ⇓⟧    ρ = ε
⟦ (i :: e) ⇓⟧ ρ = (lookup i ρ) · ⟦ e ⇓⟧ ρ
```

Below, the formal specification of soundness for the decision procedure. If two monoids are decided equal, they must evaluate to an equal value given any environment. However, no claims can be made if they are not decided equal: the carrier may have properties other than the monoidal. (Take, for instance, the natural numbers with addition, where $a+b$ is equivalent to $b+a$.)

```
Solution : ∀ {n} → Eqn n → Set
Solution {n} (e₁ ≡′ e₂) with (normalise e₁) ≟ (normalise e₂)
```

```
...        | no _ = ⊤
...        | yes _ = ∀ (ρ : Env n) → ⟦ e₁ ⟧ ρ ≡ ⟦ e₂ ⟧ ρ
```

The decidable equality of normal forms (here $\_\stackrel{?}{=}\_$) is defined as the decidable equality of lists of finite indices, which in turn relies on the decidable equality of finite indices.

`Solution` is a specification defined for a given equation. Such specification must be met for all equations:

```
solve : ∀ {n} (eqn : Eqn n) → Solution eqn
```

If the evaluation of an expression can be shown to be decomposable into its normalisation followed by the evaluation of such normal form, then by congruence of functions, an equivalence of normal forms implies an equivalence of terms after evaluation:

```
solve (e₁ ≡′ e₂) with (normalise e₁) ≟ (normalise e₂)
... | no _ = tt
... | yes eq = λ ρ →
  ⟦ e₁ ⟧ ρ
    ≡⟨ correct e₁ ρ ⟩
  ⟦ normalise e₁ ⇓⟧ ρ
    ≡⟨ cong (λ ● → ⟦ ● ⇓⟧ ρ) eq ⟩
  ⟦ normalise e₂ ⇓⟧ ρ
    ≡⟨ sym (correct e₂ ρ) ⟩
  ⟦ e₂ ⟧ ρ
    ■
  where open ≡−Reasoning
```

Put in a diagrammatic form, the following diagram must be shown to commute:

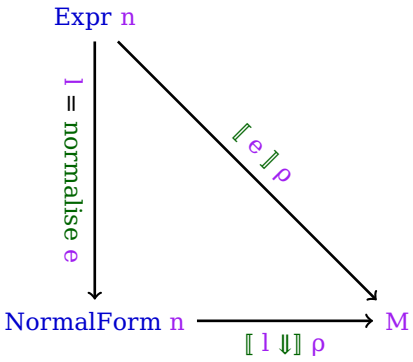


Figure 3.1.: ∀ e ρ → correct e ρ

For every expression constructor, it has to be shown that its corresponding normalisation function does not affect the structure of the monoid. In the case of the normalisation of `a · b` expressions, the proof for both sub-expressions is obtained inductively. For greater clarity, I use this top-down whiteboard-style reasoning instead of the shorter `rewrite` directives.

```
correct : ∀ {n} (e : Expr n) (ρ : Env n)
        → ⟦ e ⟧ ρ ≡ ⟦ normalise e ⇓⟧ ρ

correct ε′        ρ = refl
correct (var′ x) ρ = law−·−ε (lookup x ρ)
correct (e₁ ·′ e₂) ρ = begin
  ⟦ e₁ ⟧ ρ · ⟦ e₂ ⟧ ρ
    ≡⟨ cong (λ ● → ● · _) (correct e₁ ρ) ⟩
  ⟦ normalise e₁ ⇓⟧ ρ · ⟦ e₂ ⟧ ρ
    ≡⟨ cong (λ ● → _ · ●) (correct e₂ ρ) ⟩
  ⟦ normalise e₁ ⇓⟧ ρ · ⟦ normalise e₂ ⇓⟧ ρ
    ≡⟨ ++−homo (normalise e₁) (normalise e₂) ρ ⟩
  ⟦ normalise e₁ ++ normalise e₂ ⇓⟧ ρ
    ■
  where open ≡−Reasoning
```

Proving that _++_ preservers the monoid's structure must be done by induction on the first argument. Unsurprisingly, together these two proofs use all of the monoid laws.

```
++−homo : ∀ {n} (e₁ e₂ : NormalForm n) → (ρ : Env n)
        → ⟦ e₁ ⇓⟧ ρ · ⟦ e₂ ⇓⟧ ρ ≡ ⟦ e₁ ++ e₂ ⇓⟧ ρ

++−homo [] e₂ ρ = law−ε−· (⟦ e₂ ⇓⟧ ρ)
++−homo (i :: e₁) e₂ ρ = begin
  ((lookup i ρ) · ⟦ e₁ ⇓⟧ ρ) · ⟦ e₂ ⇓⟧ ρ
    ≡⟨ law−·−· _ _ _ ⟩
  (lookup i ρ) · (⟦ e₁ ⇓⟧ ρ · ⟦ e₂ ⇓⟧ ρ)
    ≡⟨ cong (λ ● → lookup i ρ · ●) (++−homo e₁ e₂ ρ) ⟩
  (lookup i ρ) · ⟦ e₁ ++ e₂ ⇓⟧ ρ
    ■
  where open ≡−Reasoning
```

3.3. Results and usage

Proofs for arbitrary equations on monoids can automatically be generated now:

```
eqn₁−auto : {T : Set}(xs : List T) → [] ++ xs ≡ xs ++ []
eqn₁−auto xs = solve (LIST−MONOID _)
                ((ε′ ·′ var′ zero) ≡′ (var′ zero ·′ ε′)) (xs :: [])
```

However, the user still needs to manually build the expressions that represent the target theorem. This includes appropriately handling the indices that refer to variables. As shown by [Bove et al., 2009] at http://www.cse.chalmers.se/~ulfn/code/tphols09/, index references

can be set up automatically, partially alleviating this problem and resulting in the following usage:

```
eqn₂−auto : {T : Set}(xs ys zs : List T)
          → (xs ++ []) ++ ([] ++ (ys ++ (ys ++ zs)))
          ≡ xs ++ ((ys ++ ys) ++ (zs ++ []))

eqn₂−auto xs ys zs = solve (LIST−MONOID _) (build 3 λ xs ys zs
                   → ((xs ·′ ε′) ·′ (ε′ ·′ (ys ·′ (ys ·′ zs))))
                   ≡′ (xs ·′ ((ys ·′ ys) ·′ (zs ·′ ε′)))) (xs :: ys :: zs :: [])
```

Agda's support for reflection can be used to build a macro that inspects the type of the goal and translates it into a data structure that the proof generating procedure can inspect. This would result in the following example usage:

```
eqn₂−magic : {T : Set}(xs ys zs : List T)
           → (xs ++ []) ++ ([] ++ (ys ++ (ys ++ zs)))
           ≡ xs ++ ((ys ++ ys) ++ (zs ++ []))

eqn₂−magic = magic−solve (LIST−MONOID _)
```

4. Solving commutative rings

A commutative ring is a carrier set `R` together with two binary operations generalising multiplication and addition. Under multiplication, `R` is a commutative monoid; under addition, an abelian group — providing an extra inverse law; multiplication distributes over addition.

```
record CommutativeRing (R : Set) : Set where
  infixl 5 _+_
  infixl 10 _*_
  infix 15 −_
  field
    _*_ : R → R → R
    1# : R
    *−assoc : (x y z : R) → (x * y) * z ≡ x * (y * z)
    *−comm : (x y : R) → x * y ≡ y * x
    *−identity : (x : R) → x * 1# ≡ x

    _+_ : R → R → R
    0# : R
    +−assoc : (x y z : R) → (x + y) + z ≡ x + (y + z)
    +−comm : (x y : R) → x + y ≡ y + x
    +−identity : (x : R) → x + 0# ≡ x
    −_ : R → R
    +−inverse : (x : R) → x + − x ≡ 0#

    distrib : (x y z : R) → (y + z) * x ≡ (y * x) + (z * x)
```

4.1. Problem description and specification

Proving equalities on commutative rings can be tedious:

```
open CommutativeRing INT−COMM−RING
eqn₃ : (x y z : ℤ) → y * (− ((+ 2) * x) + z + (+ 2) * x) ≡ y * z
eqn₃ x y z = begin
  y * ((− ((+ 2) * x) + z) + ((+ 2) * x))
    ≡⟨ cong (λ ● → y * (● + (+ 2) * x)) (+−comm (− ((+ 2) * x)) z) ⟩
  y * ((z + − ((+ 2) * x)) + ((+ 2) * x))
    ≡⟨ cong (λ ● → y * ●) (+−assoc z _ _) ⟩
```

```
y * (z + (- ((+ 2) * x) + ((+ 2) * x)))
  ≡⟨ cong (λ ● → y * (z + ●)) (+-comm _ _) ⟩
y * (z + ((+ 2) * x + - ((+ 2) * x)))
  ≡⟨ cong (λ ● → y * (z + ●)) (+-inverse ((+ 2) * x)) ⟩
y * (z + (+ 0))
  ≡⟨ cong (y *_) (+-identity z) ⟩
y * z
  ■
where open ≡-Reasoning
```

The goal of a problem solver for equalities on commutative rings is to generate these proofs automatically for any commutative ring.

4.2. A verified decision procedure

Soon after I started to develop a solver in Agda, I found out that Agda's standard library already includes one, and that it is far more general than anything I could have written. Consequently, I decided to comment on their solution instead.

An automated solver for equations on commutative rings was provided in [Boutin, 1997] as an example use of reflection in automated theorem proving. Coq's `ring` tactic implemented such solver. Later, [Grégoire and Mahboubi, 2005] proposed a more efficient solution, which Coq adopted. [Russinoff, 2017] adapts Grégoire and Mahboubi's solution to the theorem prover ACL2 in a structured manner and is clarifying in some regards.

Expressions are represented as polynomials that are indexed by the number of variables in them. Shortcut functions for common operations like addition, multiplication and subtraction are provided.

```
data Op : Set where
  [+] : Op
  [*] : Op

data Polynomial (m : ℕ) : Set r₁ where
  op  : (o : Op) (p₁ : Polynomial m) (p₂ : Polynomial m) → Polynomial m
  con : (c : C.Carrier) → Polynomial m
  var : (x : Fin m) → Polynomial m
  _:^_ : (p : Polynomial m) (n : ℕ) → Polynomial m
  :-_ : (p : Polynomial m) → Polynomial m
```

The solver's high-level structure is similar to the monoid solver's one described in § 3. The heart of it proves that evaluating a polynomial within an environment ρ is equal to first normalising it and then evaluating its normal form within ρ — it shows that normalisation is structure-preserving. Akin to `solve` in § 3.2, this proof is then used to conclude that if two normal forms are equivalent, so must the original polynomials after evaluation be.

Polynomials with a single variable can be represented as *Horner normal forms*:

$$
\begin{aligned}
&a_n x^n + a_{n-1} x^{n-1} + \ldots + a_0 \\
\equiv &((a_n x + a_{n-1}) x + \ldots) x + a_0
\end{aligned}
$$

```
mutual
  data HNF : ℕ → Set r₁ where
    ∅ : ∀ {n} → HNF (suc n)
    _*x+_ : ∀ {n} → HNF (suc n) → Normal n → HNF (suc n)
```

To make the solution multivariate, coefficients are replaced by polynomials containing the additional variables. Integer coefficients form a commutative ring too, and thus this results in an opportunity to handle both integer coefficients and coefficients containing additional variables uniformly, as commutative rings.

$$
\begin{aligned}
&y^2 x^2 + y^2 + yx + 2x + 2 \\
\equiv &((0 + y\,y + 1)x + y + 2)x + y\,y + 2
\end{aligned}
$$

```
  data Normal : ℕ → Set r₁ where
    con : C.Carrier → Normal zero
    poly : ∀ {n} → HNF (suc n) → Normal (suc n)
```

In fact, the module does not require constant coefficients to be integers. Any commutative ring that can be evaluated into the main ring in a law-respecting manner and has decidable equality suffices.

```
module CommutativeRings
  {r₁ r₂ r₃}
  (Coeff : RawRing r₁)                  -- Coefficient "ring".
  (R : AlmostCommutativeRing r₂ r₃) -- Main "ring".
  (morphism : Coeff −Raw−AlmostCommutative⟶ R)
  (_coeff≟_ : Decidable (Induced−equivalence morphism))
  where
private module C = RawRing Coeff
```

The module handles equality generically, as a binary relation on the carrier set. This and the need to evaluate constant coefficients, results in an inductive definition of equality of normal forms being necessary.

```
mutual
  data _≈H_ : ∀ {n} → HNF n → HNF n → Set (r₁ ⊔ r₃) where
    ∅ : ∀ {n} → _≈H_ {suc n} ∅ ∅
    _*x+_ : ∀ {n} {p₁ p₂ : HNF (suc n)} {c₁ c₂ : Normal n} →
```

```
p₁ ≈H p₂ → c₁ ≈N c₂ → (p₁ *x+ c₁) ≈H (p₂ *x+ c₂)

data _≈N_ : ∀ {n} → Normal n → Normal n → Set (r₁ ⊔ r₃) where
  con : ∀ {c₁ c₂} → ⟦ c₁ ⟧′ ≈ ⟦ c₂ ⟧′ → con c₁ ≈N con c₂
  poly : ∀ {n} {p₁ p₂ : HNF (suc n)} → p₁ ≈H p₂ → poly p₁ ≈N poly p₂
```

Evaluation within an environment of both polynomial expressions and normal forms is then defined. Similar to monoids, environments are vectors of elements belonging to the carrier set, and need to be of the same length as the number of unknowns in the polynomial or normal form being evaluated. Evaluation of normal forms is then shown to be congruent with respect to the inductive equality.

The exact choice of normal form influences both performance and the complexity of proofs. The data type presented previously does not in itself ensure the uniqueness of those normal forms that evaluate to 0 — $0x$ can be represented both as `∅` and `∅ *x+ con C.0#`. To remedy this (and keep the size of terms small) a wrapper function that (if pertinent) minimises univariate Horner normal forms to `∅` is defined around `_*x+_`.

Operations like addition and multiplication are defined for Horner normal forms and then used by the normalisation process, which operates inductively. Both the operations and the normalisation process use the simplifying variant of `_*x+_` to keep their results canonical.

For each operation, a homomorphism lemma is proven, showing that evaluating the given operation on any two normal forms is equivalent to evaluating both normal forms separately and then applying the given operation to them. Finally, the main correctness proof uses these lemmas to inductively proof that as a whole, normalisation respects the structure of commutative rings.

4.3. Usage

An example usage of a ring solver for integers follows. The last argument is an equality proof between the target theorem and the theorem proven by the solver. This allows later rewrites and adjustments.

```
open Data.Integer.Properties.RingSolver

ex₁ : (x y z : ℤ)
    → x ^ 3 + y * x ^ 2 − x ^ 2 + + 2 * x * y + y ^ 2 − + 2 * x − + 2 * y
    ≡ (x + y − + 2) * (x ^ 2 + x + y)
ex₁ = solve 3 (λ x y z →
                  x :^ 3 :+ y :* x :^ 2 :− x :^ 2 :+ con (+ 2) :* x :* y
                  :+ y :^ 2 :− con (+ 2) :* x :− con (+ 2) :* y
                := (x :+ y :− con (+ 2)) :* (x :^ 2 :+ x :+ y))
                refl
```

5. Solving Presburger arithmetic

In 1929, Mojżesz Presburger presented and proved decidable a predicate logic on natural numbers (expandable to integers, rational numbers or real numbers) with addition as its only operation. The original paper [Presburger, 1929] is in Polish and uses outdated notation; [Stansifer, 1984] contains an English translation and comments clarifying the original. Several procedures capable of deciding Presburger arithmetic exist, some of them I introduce later on. Nevertheless, [Fischer and Rabin, 1974] showed that the worst case run time of any such procedure is doubly exponential.

Here are some example simple predicates that better illustrate the expressiveness of Presburger arithmetic.

$$\forall x.\, \exists y.\, x = 2y \;\vee\; x = 2y + 1 \tag{5.1}$$
$$\forall x.\, \neg\exists y.\, 2x = 2y + 1 \tag{5.2}$$
$$\forall x.\, 4|x \;\Rightarrow\; 2|x \tag{5.3}$$
$$\forall x.\, x < x + 1 \tag{5.4}$$

To our knowledge, there is no other implementation of a decision procedure for Presburger arithmetic written in Agda. In this chapter, I introduce two decision procedures on integers and partly implement one of them, then verify its correctness.

5.1. Problem description and specification

To solve Presburger arithmetic is to create a verified procedure capable of deciding any well-formed Presburger predicate where all variables are bound. Without an automated procedure, proving a predicate like Equation 5.1 can already become burdensome:

```
pred₁ : ∀ n → ∃ λ m → ((n ≡ 2 * m) ⊎ (n ≡ 2 * m + 1))
pred₁ zero = 0 , inj₁ refl
pred₁ (suc zero) = 0 , inj₂ refl
pred₁ (suc (suc n))                    with pred₁ n
pred₁ (suc (suc .(m′ + (m′ + 0)))) | m′ , inj₁ refl =
  suc m′ , inj₁ (cong suc (sym (+-suc m′ (m′ + 0))))
pred₁ (suc (suc .(m′ + (m′ + 0) + 1))) | m′ , inj₂ refl =
  suc m′ , inj₂ (cong suc (cong (_+ 1) (sym (+-suc m′ (m′ + 0)))))
```

I define Presburger predicates as any formulae built using the following syntax:

```
data Atom (i : ℕ) : Set where
  num′ : ℤ   → Atom i
  _+′_ : Atom i → Atom i → Atom i
  _−′_ : Atom i → Atom i → Atom i
  _∗′_ : ℤ → Atom i → Atom i
  var′ : Fin i → Atom i

data Rel : Set where
  <′ >′ ≤′ ≥′ =′ : Rel

data Formula (i : ℕ) : Set where
  -- Divisibility
  _∣′_     : ℕ → Atom i      → Formula i
  _[_]_    : Atom i → Rel → Atom i → Formula i
  _∧′_ _∨′_ _⇒′_ : Formula i → Formula i → Formula i
  ¬′_      : Formula i       → Formula i
  -- Introduction of new variables
  ∃′_ ∀′_ : Formula (suc i) → Formula i
```

I use de Bruijn indices [de Bruijn, 1972] to refer to bindings by their proximity: a variable with index `0` refers to the variable introduced by the most immediate binding to its left; index `n` refers to the variable introduced `n` bindings away. Using de Bruijn indices instead of variable names has two main advantages:

- there is no need to rename variables on substitution; and
- the choice of variable names does not affect equality.

For any formula of type `Formula n`, `n` indicates the number variables introduced outside of that formula. Quantifiers `∀′_` and `∃′_`make a new variable available to their arguments.

Equation 5.1 can be transcribed as follows:

```
pred′₁ : Formula 0
pred′₁ = ∀′ ∃′ ((x [ =′ ] ((+ 2) ∗′ y))
           ∨′ (x [ =′ ] (((+ 2) ∗′ y) +′ (num′ (+ 1)))))
  where
    x = var′ (suc zero)
    y = var′ zero
```

5.2. Decision procedures

There exist numerous procedures capable of deciding Presburger arithmetic. They are primarily distinguished by the domain of their formulae and their normalisation requirements. The satisfiability of Presburger formulae in any domain gets carried onto superset domains; the unsatisfiability gets carried onto subset domains, as noted in [Janičić et al., 1997].

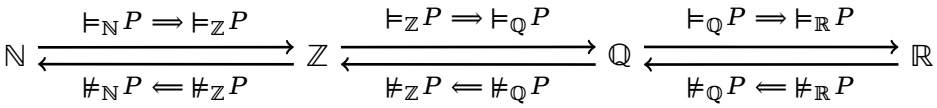


Figure 5.1.: Decidability across domains

Some Presburger formulae are valid on integers but invalid on natural numbers: $\exists x.\, x{+}1{=}0$. Others are valid on rational numbers but invalid on integers: $\exists x.\, 2x{=}1$. When considering which decision procedures to explore, I immediately discarded the ones acting on real numbers — irrational numbers are not straightforward to handle in constructive mathematics. The most well-documented procedures are on integers, and the usage of integer Presburger arithmetic is common enough for an automated solver to be of great value. Given a solver for problems on integers, one just needs add the extra condition $0{\leq}x$ to every existential quantifier to solve problems on natural numbers.

I chose the Omega Test and Cooper's Algorithm as the two integer decision procedures to explore. Michael Norrish depicts in [Norrish, 2003] the state of affairs concerning the implementation of Presburger arithmetic deciding procedures by proof assistants. He then continues describing the Omega Test and Cooper's Algorithm and proposes verified implementations for both of them for the proof assistant HOL. A later talk gives additional details. [Norrish, 2006]

5.3. The Omega Test

The Omega Test was first introduced in [Pugh, 1991]. It adapts Fourier-Motzkin elimination — which acts on real numbers — to integers, and requires the input formula to be put in disjunctive normal form.

This section starts by implementing a normalisation procedure that puts input formulae into their equivalent normal forms. It then takes a leap and implements variable elimination for quantifier-free existential formulae and verifies it sound. Finally, it provides the reader with some usage examples and outlines future work.

This section is significantly based on the material found in [Norrish, 2003] and [Norrish, 2006].

5.3.1. Normalisation

Transforming input formulae into disjunctive normal forms can blow up the size of formulae exponentially, as can clearly be seen whenever a conjunction is normalised over a disjunction:

$$(P \vee Q) \wedge (R \vee S) \equiv (P \wedge R) \vee (P \wedge S) \vee (Q \wedge R) \vee (Q \wedge S)$$

As part of normalisation, universal quantifiers need to be transformed into existential ones resorting on the following equivalence:

$$\forall x.P(x) \equiv \neg\exists x.\neg P(x)$$

Existential quantifiers must be distributed over disjunctions:

$$\exists x.\, P(x) \vee Q(x) \equiv (\exists x.\, P(x)) \vee (\exists x.\, Q(x))$$

Negation needs to be pushed inside conjunctions and disjunctions, and double negation needs to be eliminated:

$$\begin{aligned}\neg(P(x) \wedge Q(x)) &\equiv \neg P(x) \vee \neg Q(x)\\ \neg(P(x) \vee Q(x)) &\equiv \neg P(x) \wedge \neg Q(x)\\ \neg\neg P(x) &\equiv P(x)\end{aligned}$$

Operations on Atoms are evaluated into linear transformations of the form $ax+by+...+cz+k$. As a consequence of limiting the domain to integers, all constraints can be translated into the canonical form $0 \leq ax+by+...+cz+k$. I use a single type to represent them both, and a parameter on that type to keep record of the number of variables within. A vector of the same length contains the coefficients $ax+by+...+cz$, where each coefficient's index is a de Bruijn index indicating the distance in bindings to where the variable was introduced. An additional constant is used to represent k.

```
record Linear (i : ℕ) : Set where
  constructor _::+_
  field
    cs : Vec ℤ i
    k : ℤ
```

Relations are normalised as follows:

$$\begin{aligned}p < q &\equiv 0 \leq q - p + 1\\ p > q &\equiv 0 \leq p - q + 1\\ p \leq q &\equiv 0 \leq q - p\\ p \geq q &\equiv 0 \leq p - q\\ p = q &\equiv 0 \leq q - p \wedge 0 \leq p - q\end{aligned}$$

Divide terms and their negations are special cases. The Omega Test produces them as a byproduct of its main theorem and uses a specific algorithm to eliminate them, as shown later. However, I do not implement such a procedure (discussed later) so I normalise divide terms into constraints by introducing a new existential quantifier:

$$\begin{aligned}n \mid a &\equiv \exists x.\, nx = a\\ n \nmid a &\equiv \exists x.\, \bigvee_{i \in 1...n-1} nx = (a + i)\end{aligned}$$

Taking all into account, the result of normalisation has to be a structure where:

(i) the top layer is a disjunction; (ii) a disjunction only contains conjunctions; and (iii) a conjunction only contains conjunctions, existential quantifiers, negated existential quantifiers, or atoms.

The following tree-like structure contains `Linear`s as leafs and, within each conjunction, distinguishes leafs from further sub-trees containing existential quantifiers.

As with `Formula`s, note that the restriction on the number of available variables is pushed inside the structure — `DNF n` can only contain `Conjunction n` and so forth. The constructors `∃` and `¬∃` make one more variable available to their substructures.

```
mutual
  data Existential (i : ℕ) : Set where
    ¬∃ : Conjunction (suc i) → Existential i
    ∃ : Conjunction (suc i) → Existential i

  record Conjunction (i : ℕ) : Set where
    inductive
    constructor 0≤_∧_E
    field
      constraints : List (Linear i)
      existentials : List (Existential i)

DNF : (i : ℕ) → Set
DNF i = List (Conjunction i)
```

Normalisation proceeds recursively, eliminating universal quantifiers, pushing conjunction and negation inward, normalising implication, evaluating operations on atoms and normalising relations between them. For the exact procedure see the accompanying code.

5.3.2. Elimination

Once normalisation has taken place, the elimination process is ran recursively on quantifier-free sub-formulae. The heart of this is an equivalence theorem that eliminates the variable bound by the innermost existential quantifier:

$$\exists x.P(x) \equiv Q$$

Theorem 1 (Pugh, 1991)**.** *Let $L(x)$ be a conjunction of lower bounds on x, indexed by i, of the form $a_i \leq \alpha_i x$, with α_i positive and non-zero. Similarly, let $U(x)$ be a set of upper bounds on x, indexed by j, of the form $\beta_j x \leq b_j$, with β_j positive and non-zero. Let m be the maximum of all β_js. Then:*

$$(\exists x.L(x) \wedge U(x)) \equiv \left(\bigwedge_{i,j}(\alpha_i - 1)(\beta_j - 1) \leq (\alpha_i b_j - a_i \beta_j)\right) \vee \bigvee_i \bigvee_{k=0}^{\left\lfloor \alpha_i - \frac{\alpha_i}{m} - 1 \right\rfloor} \exists x.(\alpha_i x = a_i + k) \wedge L(x) \wedge U(x)$$

Pugh refers to the first disjunct as the *real shadow* and to the remaining as *splinters*. If all α_i or all β_j are 1 — that is, if for every (α,β) pair $\alpha \equiv 1 \vee \beta \equiv 1$ —, the theorem reduces to the *exact shadow*:

$$\exists x.L(x) \wedge U(x) \equiv \bigvee_{i,j} a_i \beta_j \leq \alpha_i b_j$$

My initial intention was to implement and verify the complete theorem. However, I quickly found out about the complexity introduced by splinters. Each splinter introduces a new existential quantifier. This quantifier is then eliminated by the following terminating method based on the Euclidean algorithm for the computation of greatest common divisors:

x is the variable to eliminate

$$\exists y.\exists x. \ldots \wedge ax = by + e \wedge \ldots \tag{5.5}$$

Find the lowest common divisor ℓ of all the coefficients on x and multiply every constraint by an integer n so that its coefficient on x is ℓ

$$\exists y.\exists x. \ldots \wedge \ell x = b' y + e' \wedge \ldots \tag{5.6}$$

Set all coefficients on x to 1 resorting to the equivalence $P(\ell x) \equiv P(x) \wedge \ell | x$.

$$\exists y.\exists x. \ldots \wedge (x = b' y + e') \wedge (\ell \mid x) \wedge \ldots \tag{5.7}$$

Substitute x

$$\exists y. \ldots \wedge \ell \mid b' y + e' \wedge \ldots \tag{5.8}$$

Eliminate the divides term by introducing a new existential

$$\exists y.\exists z. \ldots \wedge \ell z = b' y + e' \wedge \ldots \tag{5.9}$$

Rearrange

$$\exists y.\exists z. \ldots \wedge b' y = \ell z - e' \wedge \ldots \tag{5.10}$$

y is the variable to eliminate

Crucially, Equation 5.8 guarantees the eventual elimination of the divides term, as $b' < \ell$ — and modulus if not. This recursive computation, justified because a transitive relation towards the left on $<$ for natural numbers eventually terminates, is not entirely trivial to model.

Commonly, structural recursion is applied onto terms that have been deconstructed by pattern matching — and thus structures get smaller in "fixed steps". Here, on the other hand, recursion has to be shown to terminate by account of the divides term's coefficient decreasing in steps of variable size.

As for verification, splinters introduce considerable complexity too. Pugh's theorem is of form $\text{LHS} \equiv D_1 \vee D_2$. That shapes the proof, which first shows the soundness of both disjuncts by proving that $D_1 \implies \text{LHS}$ and $D_2 \implies \text{LHS}$ and then its completeness by proving that $\text{LHS} \wedge \neg D_1 \implies D_2$. From these three proof obligations, the last one is the hardest to satisfy.

After some initial exploratory programming, given the complexity they entail, both in terms of implementation and verification, and taking time constraints into account, I decided to discard implementing splinters. Other interactive theorem provers like Coq, HOL or Isabelle, limit the completeness of their implementations too, often just to the real shadow.

This decision left me with two components:

Dark shadow Always applicable. Formulae decided satisfiable after elimination can be shown to be satisfiable before elimination.

Real shadow Only applicable when all α or all β are 1. It preserves both satisfiability and unsatisfiability.

A decision procedure with only these tests is sound but incomplete, and thus has three possible outcomes:

```
data Result : Set where
  satisfiable unsatisfiable undecided : Result
```

Implementing the dark shadow is not involved. With l as the lower bound constraint, u as the upper bound and α, a, β and b as per Pugh:

```
_↓ₚ : ∀ {i} → Pair (suc i) → Linear i
((l , _) , (u , _)) ↓ₚ with head l | ⊖ (tail l) | - (head u) | tail u
... | α | a | β | b = (α ⊛ b) ⊖ (β ⊛ a) ⊖ (# ((α - + 1) ∗ (β - + 1)))
```

The dark shadow reduces to the real shadow when all α_i or all β_j are 1. I use the function `_↓ₚ` for both computations, and then interpret the results accordingly. Unsatisfiability can only be asserted if the real shadow's precondition is met. If it is not, `unsatisfiable` needs to be interpreted as `undecided`. Following, an elimination procedure for quantifier free formulae. `⊨?_[ [] /x]` decides constraints with no variables, as shown in the next section.

```
Ω : ∀ {i} → List (Linear i) → Result
Ω {zero} as with all ⊨?_[ [] /x] as
Ω {zero} as | yes _ = satisfiable
Ω {zero} as | no _ = unsatisfiable
Ω {suc i} as with Ω (as ↓)
Ω {suc i} as | unsatisfiable with all α≡1∨-β≡-1? (pairs as)
Ω {suc i} as | unsatisfiable | yes _ = unsatisfiable
Ω {suc i} as | unsatisfiable | no _ = undecided
Ω {suc i} as | r                    = r
```

5.3.3. Verification

This subsection verifies the soundness of the incomplete elimination procedure on quantifier-free formulae presented above. The exact specification follows:

```
Ω−Sound : ∀ {i} (as : List (Linear i)) → Set
Ω−Sound as with Ω as
Ω−Sound as | undecided     = ⊤
Ω−Sound as | satisfiable   = ⊨ as
Ω−Sound as | unsatisfiable = ⊨ as → ⊥
```

No proof is required if the procedure is incapable of deciding the input; an environment satisfying the input is required if the input is decided satisfiable; a function showing the inadequacy of any given environment is required if the input is decided unsatisfiable. The goal is to satisfy this predicate for any conjunction of constraints. (The meaning of `⊨` is explained below.)

Preamble

Although their definitions are available in the source code accompanying this report, my aim is to provide the reader with an intuition of the meaning of some of the different symbols used in this subsection.

`Env i` An environment with `i` variables, usually named `ρ`.

`LowerBound`, `Irrelevant`, `UpperBound` Predicates on a linear's innermost variable's coefficient c. They state $0<c$, $0=c$ and $0>c$ respectively.

`Pair i` A pair of lower bound and upper bound constraints, usually named `lu`.

```
Pair : (i : ℕ) → Set
Pair i = Σ (Linear i) LowerBound × Σ (Linear i) UpperBound
```

`[ ρ /x] a` The integer result of substituting the variables in `a` with the values in the environment `ρ` and evaluating.

`⊨[ ρ /x] a` The foundation stone of verification: the interpretation of the value `a` as a type after substitution.

```
⊨[_/x] : ∀ {i} → Env i → Linear i → Set
⊨[ ρ /x] a = + 0 ≤ ([ ρ /x] a)
```

`⊨? a [ ρ /x]` A function deciding whether the interpretation of `a` after substitution is inhabited.

`⊨ as` An environment satisfying every `a` in `as` after substitution.

```
⊨ : ∀ {i} → List (Linear i) → Set
⊨ {i} as = Σ (Env i) λ ρ → ∀[ as ] ⊨[ ρ /x]
```

Variations $\ldots_{\mathrm{p}}$ For convenience. The function is applied to a pair of lower bound and upper bound constraints.

Variations $\ldots_{\mathrm{i}}$ For convenience. The function is applied to an irrelevant constraint.

`∀[ xs ] P` The proof that `P x` for every `x` in `xs`.

`∃[ xs ] P` The proof that `P x` for some `x` in `xs`.

Dark shadow

The goal is to prove that the elimination performed by the dark shadow preserves satisfiability: whenever a formula is satisfiable after applying dark shadow elimination to it, it must be shown to be satisfiable before elimination too.

$$\bigwedge_{i,j}(\alpha_i - 1)(\beta_i - 1) \leq \alpha_i b_j - a_i \beta_j \implies \exists x.L(x) \wedge U(x)$$

The original proof proceeds by induction on every $L(x) \times U(x)$ pair, where the proof obligation is fulfilled resorting to a proof by contradiction:

$$\neg(\exists x.a \leq \alpha x \wedge \beta x \leq b) \implies \neg(\alpha - 1)(\beta - 1) \leq \alpha b - a\beta$$

However, P cannot be generally concluded from $\neg P \to \bot$ in constructive mathematics: the first requires a witness $p{:}P$ that the later does not provide. Nevertheless, a proof by contradiction still has its use. If the elements to test for P can be limited to a finite set, a proof by contradiction — showing that it cannot be that P is false for every element — can be used to build a terminating search function that is guaranteed to find an element satisfying P.

Below I present such a generalised search function, searching within a finite list for elements satisfying a decidable predicate.

```
search : {A : Set} {P : A → Set} (P? : Decidable P) (as : List A)
       → (All (¬_ ∘ P) as → ⊥)
       → Σ A P

search P? [] raa = ⊥-elim (raa [])
search P? (a :: as) raa with P? a
search P? (a :: as) raa | yes p = a , p
search P? (a :: as) raa | no ¬p = search P? as (λ ¬pas → raa (¬p :: ¬pas))
```

In this case, the search is for some x that satisfies a conjunction of constraints of form $a \leq \alpha x \wedge \beta x \leq b$, with α and β positive and non-zero. For every constraint, x must be bound between

$\lfloor \frac{a}{\alpha} \rfloor$ and $\lceil \frac{b}{\beta} \rceil$; the conjunction of all constraints must be bound between the highest lower bound and the lowest upper bound.

```
start : ∀ {i} → Env i → List (Σ (Linear (suc i)) LowerBound) → ℤ
start ρ ls = List.foldr _⊔_ (+ 0) (map (a/α ρ) ls)

stop : ∀ {i} → Env i → List (Σ (Linear (suc i)) UpperBound) → ℤ
stop ρ us = List.foldr _⊓_ (+ 0) (map (b/β ρ) us)

search-space : ∀ {i} → Env i → List (Pair (suc i)) → List ℤ
search-space ρ lus with start ρ (map proj₁ lus)
search-space ρ lus | Δ₀ with stop ρ (map proj₂ lus) - Δ₀
search-space ρ lus | Δ₀ | + n = List.applyUpTo (λ i → + i + Δ₀) n
search-space ρ lus | Δ₀ | -[1+ n ] = []
```

The proof outlined by Norrish could be used as a guarantee of the success of the search. However, while Norrish's proof by contradiction is on individual pairs of constraints…

```
⊨norrish : ∀ {i} (ρ : Env i) (xs : List ℤ) (lu : Pair (suc i))
         → ¬ ∃[ xs ] (λ x → ⊨[ x :: ρ /x]ₚ lu)
         → ¬ ⊨[ ρ /x] (lu ↓ₚ)
```

…the search function demands a proof by contradiction on the entire conjunction of constraint pairs.

```
by-contradiction : ∀ {i} (ρ : Env i) (xs : List ℤ) (lus : List (Pair (suc i)))
                 → ∀[ map _↓ₚ lus ] ⊨[ ρ /x]
                 → ¬ ∀[ xs ] (λ x → ¬ ∀[ lus ] ⊨[ x :: ρ /x]ₚ)
```

Nevertheless, the premise that must be proven false (informally, $\forall x.\neg\forall lu.\vDash_x lu$) is equivalent to the form $\exists lu.\neg\exists x.\vDash_x lu$ — where every l is paired with every u. This later form is suitable to be fed into Norrish's proof by contradiction, which for any lu expects $\neg\exists x.\vDash_x lu$. The difference is that Norrish's proof is used only once. Note that the unsolved postulate is the justification offered by Norrish for his initial induction. The proof is a one-way implication, but bi-implication can be shown.

```
postulate ∀lus∃xs⇒∃xs∀lus : ∀[ lus ] (λ lu → ∃[ xs ] (λ x → ⊨[ x :: ρ /x]ₚ lu))
                           → ∃[ xs ] (λ x → ∀[ lus ] ⊨[ x :: ρ /x]ₚ)

∀xs¬∀lus⇒∃lus¬∃xs : ∀[ xs ] (λ x → ¬ ∀[ lus ] ⊨[ x :: ρ /x]ₚ)
                  → ∃[ lus ] (λ lu → ¬ ∃[ xs ] λ x → ⊨[ x :: ρ /x]ₚ lu)

∀xs¬∀lus⇒∃lus¬∃xs = begin
  ∀[ xs ] (λ x → ¬ ∀[ lus ] ⊨[ x :: ρ /x]ₚ)
    ~⟨ AllProp.All¬⇒¬Any ⟩
```

```
¬ ∃[ xs ] (λ x → ∀[ lus ] ⊨[ x :: ρ /x]ₚ)
  ~⟨ (λ ¬∃xs∀lus ∀lus∃xs → ¬∃xs∀lus (∀lus∃xs⇒∃xs∀lus ∀lus∃xs)) ⟩
¬ ∀[ lus ] (λ lu → ∃[ xs ] λ x → ⊨[ x :: ρ /x]ₚ lu)
  ~⟨ AllProp.¬All⇒Any¬ (λ lu → any (λ x → ⊨? lu [ x :: ρ /x]ₚ) xs) lus ⟩
∃[ lus ] (λ lu → ¬ ∃[ xs ] λ x → ⊨[ x :: ρ /x]ₚ lu)
  ■
where open ⇒−Reasoning
```

Finally, the lu pair for which $\neg\exists x.\vDash_x lu$ is found and used to derive a constradiction using Norrish's proof.

```
¬∃lus¬∃xs : (lus : List (Pair (suc i)))
           → ∀[ map _↓ₚ lus ] ⊨[ ρ /x]
           → ∃[ lus ] (λ lu → ¬ ∃[ xs ] λ x → ⊨[ x :: ρ /x]ₚ lu)
           → ⊥

¬∃lus¬∃xs [] [] ()
¬∃lus¬∃xs (lu :: lus) (⊨lu↓ :: ⊨lus↓) (here ¬∃xs) = ⊨norrish ρ xs lu ¬∃xs ⊨lu↓
¬∃lus¬∃xs (lu :: lus) (⊨lu↓ :: ⊨lus↓) (there ∃lus¬∃xs) = ¬∃lus¬∃xs lus ⊨lus↓ ∃lus¬∃xs
```

Put together, this satisfies the proof by contradiction:

```
by−contradiction : ∀ {i} (ρ : Env i) (xs : List ℤ) (lus : List (Pair (suc i)))
                   → ∀[ map _↓ₚ lus ] ⊨[ ρ /x]
                   → ¬ ∀[ xs ] (λ x → ¬ ∀[ lus ] ⊨[ x :: ρ /x]ₚ)

by−contradiction {i} ρ xs lus ⊨lus↓ ∀xs¬∀lus =
  ¬∃lus¬∃xs lus ⊨lus↓ (∀xs¬∀lus⇒∃lus¬∃xs ∀xs¬∀lus)
```

The proof by contradiction is then used to guarantee the success of the search for x:

```
⊨↑ₚ : ∀ {i} (ρ : Env i) (lus : List (Pair (suc i)))
      → ∀[ map _↓ₚ lus ] ⊨[ ρ /x]
      → Σ ℤ λ x → ∀[ lus ] ⊨[ x :: ρ /x]ₚ

⊨↑ₚ ρ lus ⊨lus↓ with search−space ρ lus
⊨↑ₚ ρ lus ⊨lus↓ | xs = search (λ x → all ⊨?_[ x :: ρ /x]ₚ lus ) xs (by−contradiction ρ xs lus ⊨lus↓)
```

Norrish's proof

Below, I briefly reproduce Norrish's proof of soundness for the dark shadow. For any pair of lower bound and upper bound constraints, it has to be shown that:

$$(\alpha - 1)(\beta - 1) \leq \alpha b - a\beta \implies (\exists x.a \leq \alpha x \wedge \beta x \leq b)$$

To prove it, assume the opposite. Then, there is no multiple of $\alpha\beta$ between $a\beta$ and αb:

$$\neg\exists x.a\beta \leq \alpha\beta x \leq \alpha b$$

As both $0<\alpha$ and $0<\beta$, the other assumption implies that $a\beta \leq \alpha b$. Take i to be the greatest multiple of $\alpha\beta$ less than $a\beta$. Then

$$\alpha\beta i < a\beta \leq \alpha b < \alpha\beta(i+1)$$

Because $0<\alpha\beta(i+1)-\alpha b$, conclude $1\leq\beta(i+1)$, and thus $\alpha\leq\alpha\beta(i+1)-\alpha b$. Similarly, $\beta\leq a\beta-\alpha\beta i$. Infer $\alpha+\beta\leq\alpha\beta+a\beta-\alpha b$, or (re-arranging), $\alpha b-a\beta<\alpha\beta-\alpha-\beta+1$, which contradicts the first assumption.

I do not intend to reproduce here the entire proof as written in Agda. In fact, time constraints and the low priority I assigned to filling in the details made me keep some sub-goals as unfinished postulates. Instead, I show how the main goal is split into smaller sub-goals and how those are later put back together. I also give an example of a finished sub-goal proof to show the reader what it looks like.

I use a parametrised module for all proofs that involve a particular lower bound and upper bound pair. I *open* the constituents of the supplied pairs so that I can refer to them more comfortably from within types.

```
module Norrish {i : ℕ} (ρ : Env i) (lu : Pair (suc i)) where
  l = proj₁ lu
  u = proj₂ lu
  α = head (proj₁ l)
  −a = tail (proj₁ l)
  a = ⊖ −a
  0<α = proj₂ l
  −β = head (proj₁ u)
  β = − −β
  b = tail (proj₁ u)
  0>−β = proj₂ u
  0<β : + 0 < β
  0<β with −β | 0>−β
  0<β | +_ _ | +≤+ ()
  0<β | −[1+_] n | z = +≤+ (Nat.s≤s Nat.z≤n)
  n = a/α ρ l
```

I define the form of some sub-goals separately, so that I can later refer to them from within multiple types too.

```
  aβ≤αb : Linear i
  aβ≤αb = ((α ⊛ b) ⊖ (β ⊛ a))
```

Next, a proof for one of the sub-goals, where I show that $(\alpha-1)(\beta-1)\leq\alpha b-a\beta$ implies $a\beta\leq\alpha b$ when both $0<\alpha$ and $0<\beta$. Observations that are a common requirement to multiple sub-goals have been abstracted away into lemmas.

```
⊨βa≤αb : ⊨[ ρ /x] (aβ≤αb ⊖ (# [α−1][β−1])) → ⊨[ ρ /x] aβ≤αb
⊨βa≤αb ⊨ds = begin
  + 0
    ≤⟨ ⊨ds ⟩
  [ ρ /x] (aβ≤αb ⊖ (# [α−1][β−1]))
    ≡⟨ cong (λ ● → [ ρ /x] (aβ≤αb ⊕ ●)) (⊖#n≡#−n [α−1][β−1]) ⟩
  [ ρ /x] (aβ≤αb ⊕ (# (− [α−1][β−1])))
    ≡⟨ [ ρ /x]−⊕ _ _ ⟩
  [ ρ /x] aβ≤αb + [ ρ /x] (# (− [α−1][β−1]))
    ≡⟨ cong (λ ● → [ ρ /x] aβ≤αb + ●) ([ ρ /x]−# _) ⟩
  [ ρ /x] aβ≤αb − [α−1][β−1]
    ≤⟨ 0≤n→m−n≤m _ _ ⊨0≤[α−1][β−1] ⟩
  [ ρ /x] aβ≤αb
    ■
  where open ≤−Reasoning
```

Putting the remaining sub-goals together, I supply Norrish's proof:

```
⊨norrish : ∀ {i} (ρ : Env i) (xs : List ℤ) (lu : Pair (suc i))
         → ¬ ∃[ xs ] (λ x → ⊨[ x :: ρ /x]ₚ lu)
         → ¬ ⊨[ ρ /x] (lu ↓ₚ)


⊨norrish ρ xs lu ⊭xs ⊨lu↓ =
  let ps = ⊨αβn<aβ≤αb<αβ[n+1] (⊨βa≤αb ⊨lu↓ ) ⊭xs
    in ⊭[α−1][β−1]≤αb−aβ (⊨α≤αβ[n+1]−αb ps) (⊨β≤aβ−αβn ps) ⊨lu↓
  where open Norrish ρ lu
```

Real shadow

The dark shadow preserves satisfiability. It must be shown that whenever the real shadow is applied, unsatisfiability is preserved too. Where all α_i or all β_i are 1:

$$\neg \bigwedge_{i,j} (\alpha_i - 1)(\beta_j - 1) \leq \alpha_i b_j - a\beta_j \implies \neg \exists x. L(x) \wedge U(x)$$

That is, given arguments $\bigwedge_{i,j}(\alpha_i-1)(\beta_j-1)\leq\alpha_i b_j-a\beta_j \implies \bot$ and $\exists x. L(x)\wedge U(x)$, the latter must be transformed into an argument suitable to the former. Using induction, the proof obligation can be reduced to a predicate on lower bound and upper bound pairs.

$$(\exists x. a \leq \alpha x \wedge \beta x \leq b) \implies (\alpha - 1)(\beta - 1) \leq \alpha b - a\beta$$

After the conjuncts on the LHS of the implication are appropriately multiplied, $a\beta\leq\alpha b$ by transitivity of $\leq$. The proof concludes as $(\alpha-1)(\beta-1)$ reduces to 0 when either α or β are 1. Below, such a proof written in Agda.

```
⊨real−shadow : (x : ℤ) → (α ≡ + 1 ⊎ −β ≡ − + 1)
                  → ⊨[ x :: ρ /x] (α x+ −a)
                  → ⊨[ x :: ρ /x] (−β x+ b)
                  → ⊨[ ρ /x] (aβ≤αb ⊖ (# [α−1][β−1]))
⊨real−shadow x α≡1∨−β≡−1 a≤αx βx≤b = begin
  + 0
    ≤⟨ ⊨[ ρ /x]−trans (β ⊛ a) (# (α ∗ β ∗ x)) (α ⊛ b)
         (a≤αx⇒aβ≤αβx x a≤αx ) (βx≤b⇒αβx≤αb x βx≤b) ⟩
  [ ρ /x] aβ≤αb
    ≡⟨ sym (IntProp.+−identity^r _) ⟩
  [ ρ /x] aβ≤αb + (+ 0)
    ≡⟨ cong (λ ● → [ ρ /x] aβ≤αb + ●) (sym ([ ρ /x]−# (+ 0))) ⟩
  [ ρ /x] aβ≤αb + [ ρ /x] (# (+ 0))
    ≡⟨ sym ([ ρ /x]−⊕ aβ≤αb (# (+ 0))) ⟩
  [ ρ /x] (aβ≤αb ⊕ (# (+ 0)))
    ≡⟨ cong (λ ● → [ ρ /x] (aβ≤αb ⊕ ●)) (sym (⊖#n≡#−n (+ 0))) ⟩
  [ ρ /x] (aβ≤αb ⊖ (# (+ 0)))
    ≡⟨ cong (λ ● → [ ρ /x] (aβ≤αb ⊖ (# ●))) (sym (α≡1∨−β≡−1→[α−1][β−1]≡0 α≡1∨−β≡−1)) ⟩
  [ ρ /x] (aβ≤αb ⊖ (# [α−1][β−1]))
    ■
  where open ≤−Reasoning
```

Delivering soundness

Next, I prove the soundness of the elimination procedure for normalised formulae of the following form:

$$\exists x.\exists x_1.\ldots\exists x_n.\ 0 \leq A[x, x_1, \ldots, x_n] \wedge 0 \leq B[x, x_1, \ldots, x_n]$$

The elimination process has to be shown to preserve both unsatisfiability and satisfiability. I do not reproduce these proofs here, they are rather bulky. Instead, I comment on their logic, although I recommend reading their code alongside.

```
Ω−sound : ∀ {i} (as : List (Linear i)) → Ω−Sound as
Ω−sound as with Ω as    | inspect Ω as
Ω−sound as | undecided | _ = tt
Ω−sound as | unsatisfiable | >[ eq ]< = unsat as eq
Ω−sound as | satisfiable | >[ eq ]< = sat as eq
```

Both proofs are recursively built. If an input is decided unsatisfiable (goal `⊨as→⊥`), a proof of unsatisfiability after elimination (`⊨as↓→⊥`) is obtained recursively. Then satisfiability before elimination is assumed (`⊨as`) and satisfiability after elimination derived (`⊨as↓`) through the use of `⊨real−shadow`. From there, a contradiction is obtained.

If an input is decided satisfiable (goal ⊨as), a proof of satisfiability after elimination (⊨as↓) is obtained recursively. Then $\models\uparrow_p$ adds a new x to the environment, and returns a proof (⊨as) that by doing so, satisfiability is preserved.

Clearly, irrelevant constraints — those where the variable to eliminate has coefficient 0 — do not have their meaning altered by elimination. Similarly, the meaning of no constraint changes after a variable with coefficient 0 is prepended to it. Given that the Omega Test requires constraints to be split into lower bounds and upper bounds, I handle irrelevant constraints outside of the it.

partition partitions a list of constraints into three sub-lists: lower bound constraints, irrelevant constraints and upper bound constraints. Functions pairs and irrels then use such output to generate a cartesian product of all lower bound and upper bound constraints, and a list of irrelevant constraints, respectively. These functions carefully avoid pattern matching on the output of partition. If they were to pattern match, Agda would no longer relate the results of pairs and irrels to the original list.

The function entangle mixes together proofs on pairs of bounds and proofs on irrelevant constraints into proofs on their original list. untangle_i and untangle_p do the reverse. If the satisfiability predicate were to be defined on sets instead of on lists, these functions would become unnecessary.

5.3.4. Results and usage

To demonstrate an example usage of the presented elimination procedure, I slightly augment the syntax of the normalised input formulae handled by the soundness proof and include negated existentials.

```
data NormalForm (i : ℕ) : Set where
  ∃  : NormalForm (suc i) → NormalForm i
  ¬∃ : NormalForm (suc i) → NormalForm i
  st : List (Linear i) → NormalForm i
```

Accepting universal quantifiers implies accepting full-blown normalised Presburger formulae (because $\forall x.P(x) \equiv \neg\exists x.\neg P(x)$, which results in disjunctions if $P(x)$ contains conjunctions). Accepting full-blown normalised Presburger formulae as presented in § 5.3.1 increases the complexity of the proof of soundness. I restrict the syntax of unnormalised expressions accordingly and mark the handling of full-blown Presburger formulae for future work.

The elimination procedure negates its output on every negated existential and ultimately relies on the previously defined Ω.

```
Ω⇓ : ∀ {i} → NormalForm i → Result
Ω⇓ (∃ nf) = Ω⇓ nf
Ω⇓ (¬∃ nf) with Ω⇓ nf
Ω⇓ (¬∃ nf) | satisfiable = unsatisfiable
Ω⇓ (¬∃ nf) | unsatisfiable = satisfiable
Ω⇓ (¬∃ nf) | undecided = undecided
Ω⇓ (st as) = Ω as
```

Soundness is defined for such elimination procedure and proven in a way similar to the proof in § 5.3.3. This time, the two functions proving satisfiability and unsatisfiability recurse on each other every time an existential quantifier is negated.

Following, a set of example usages. The terms inside of `{!…!}` are not accepted by the type-checker:

```
-- Shortcuts
x : ∀ {i} → Atom (suc (suc i))
x = var′ (suc zero)

y : ∀ {i} → Atom (suc i)
y = var′ zero

-- Some theorems typecheck

ex₁ : Σ ℤ λ x → Σ ℤ λ y → + 0 < x × x + + 1 < + 2 ∗ y × y < + 4
ex₁ = solve (∃′ ∃′ : ((num′ (+ 0) [ <′ ] x)
                    ∧′ (x +′ (num′ (+ 1))) [ <′ ] ((+ 2) ∗′ y)
                    ∧′ y [ <′ ] (num′ (+ 4))))

ex₂ : ¬ Σ ℤ λ x → Σ ℤ λ y → y > + 0 × x − y ≥ x
ex₂ = solve (¬′ ∃′ ∃′ : y [ >′ ] num′ (+ 0) ∧′ (x −′ y) [ ≥′ ] x)

-- Predicates proven false do not typecheck

ex₃ : Σ ℤ λ y → y < y
ex₃ = {!solve(∃′:(y[<′]y))!}

-- The negation of predicates proven false do typecheck

¬ex₃ : ¬ Σ ℤ λ y → y < y
¬ex₃ = solve (¬′ ∃′ : (y [ <′ ] y))

-- The decision procedure is sound but incomplete
-- Sometimes, neither a predicate nor its negation typecheck

ex₄ : Σ ℤ λ x → Σ ℤ λ y → + 2 ∗ x ≡ + 2 ∗ y + + 1
ex₄ = {!solve(∃′∃′:(((+2)∗′x)[=′](((+2)∗′y)+′num′(+1))))!}

¬ex₄ : ¬ Σ ℤ λ x → Σ ℤ λ y → + 2 ∗ x ≡ + 2 ∗ y + + 1
¬ex₄ = {!solve(¬′∃′∃′:(((+2)∗′x)[=′](((+2)∗′y)+′num′(+1))))!}
```

5.3.5. Future work

Although the present development is in my view satisfactory, there is ample room for further work. Below is a to-do list, ordered by priority, of what my work after submission is likely to be.

Removal of postulates Currently several base lemmas remain postulates and so do several lemmas used by Norrish. Fulfilling these is a priority and — I am confident — a very realistic goal. Once completed, Agda can compile the program in safe mode: after that the correctness of the program is proven beyond any reasonable doubt.

Evaluation of normal forms Evaluation is currently defined for a subset of input formulae. Evaluating full-blown formulae requires a more complex proof of the correctness of normalisation and a more complex evaluation procedure. Still, this complexity would not impregnate the "inner" layer of the proof of soundness, and the implementation of Norrish's proof would remain unchanged.

Verification of the normalisation process Because I had not enough time to evaluate normal forms, I did not attempt to verify normalisation correct either. This is likely to be mildly labourious.

Quoting Building input formulae automatically out of users' goals is a major usability improvement.

Submission to the standard library Submitting my development for inclusion in Agda's standard library would be the culmination of this work. There is no guarantee this will happen and, if it does, I expect it to entail considerable communication and adaptation work.

Implementation and verification of splinters Most proof assistants provide incomplete Presburger solvers that do not make use of splinters. Given the complexity of implementing and verifying them, this as an entirely optional goal.

5.4. Cooper's Algorithm

During my initial research phase, I briefly considered Cooper's Algorithm as a candidate for a verified Presburger arithmetic solver. First introduced in [Cooper, 1972], [Norrish, 2003] and [Chaieb and Nipkow, 2003] provide comprehensible reviews and discuss implementation details.

The main elimination theorem handles both disjunctions and conjunctions, and thus there is no need to normalise input formulae into DNF or CNF, but negation needs to be pushed inside. Once a quantifier-free expression is selected for variable elimination, the lowest common multiplier ℓ of all coefficients on x needs to be computed, all constraints multiplied appropriately so that their coefficients on x is $\pm\ell$ and finally, all coefficients on x divided by ℓ in accordance to the following equivalence:

$$P(\ell x) \equiv P(x) \wedge \ell | x$$

Implementing the main elimination step is straightforward as well. The main theorem operates on divides terms too, and there is therefore no need to eliminate them.

As with the Omega Test, elimination occurs into an equivalent disjunction, which leaves three goals to be verified — $D_1 \implies \text{LHS}$, $D_2 \implies \text{LHS}$ and $\text{LHS} \wedge \neg D_2 \implies D_1$. However, unlike with the Omega Test, no shortcut can be applied to decide a formula unsatisfiable; partly verifying the theorem results in an incomplete procedure only capable of announcing the satisfiability of a quantifier-free formula. Verifying the whole theorem is considerably more complex than verifying the totality of the Omega Test, and I therefore discarded the more efficient Cooper's Algorithm in favour of the simpler Omega Test.

6. Verification and validation

Dependent types facilitate definitions up to a great level of precision. These types are used to accurately model specifications. A formal specification is then considered to have been met if a term inhabiting its corresponding type is supplied. No amount of anecdotal evidence (testing) can obtain the grade of verification attained by these machine-checked formal proofs. These machine-proofs are much stronger evidence than human pair reviews. § 2.1 introduces the correspondence between logic and computation.

For the exact details on the verification of the software developed for this report, I refer the reader to the corresponding sections § 3.2, § 4.1 and § 5.3.3 and to the accompanying source code. The brevity of this chapter is a consequence of the central role that verification plays through the entire work.

Due to time constraints, some propositions in § 5 remain postulates and therefore circumvent all verification. However, these are all relatively simple lemmas or have been proven correct in [Norrish, 2003], and I am confident that with sufficient time I could satisfy them. If these postulates are accepted as truthful, the rest of the developed software follows as verified.

This report and the source code that comes with it are literate Agda files. They must be type-checked to automatically have their syntax highlighted. This occurs as part of the build process every time this report is built.

7. Overall evaluation

This project set out to research the construction of evidence providing problem solvers in Agda. I selected three problems to explore: equations on monoids, for which I provide a completely verified solver; equations on commutative rings, for which I gain insight from an already existing solution; and Presburger arithmetic, for which I build a solution for the first time implemented in Agda.

The solutions in § 3 and § 4 are final — they fully settle their respective problems. The solution for § 5 is not, hence I outline future work in § 5.3.5. Nevertheless, in § 5.3.4 I provide a limited interface through which the user can already benefit from my work.

There exists no reasonable doubt about the correctness of the solver for monoids, the solver for commutative rings or, obviating postulates, the solver for Presburger arithmetic. The postulated claims are all reasonable to make and have, in many cases, already been proven by others.

The research work involved in this project has been considerable — particularly during the problem selection phase. Although two deliverables were produced, this project was primarily a research project. A plethora of little discoveries had to be made and often, progress was rather slow and irregular. As my supervisor well put it, my learning process was by implosion: I started with a multitude of ill-defined concepts and vague ideas and no sense of their relevance. They were gradually refined and made more precise.

In the course of this project, and sometimes indirectly, I learned bits and pieces about abstract algebra, type theory, category theory and logic. I now better understand what it is to solve a problem constructively; how proofs of correctness are structured; how implementation and verification relate; how Agda's pattern matching and unification works; and what dependent types have to bring to the table. Finally, the experience of interpreting and formally reproducing a scientific paper has been invaluable.

7.1. Organisation

I used my blog mostly to sketch new ideas and take notes, not so much for planning. For that, I relied on my whiteboard — the source of all organisation in my living. Once I started writing this report, I used the LaTeXpackage **todonotes** to keep track of tasks within the report itself. Tasks are highlighted as big orange notes on the sides of pages, and are impossible to miss during reviews. I used `grep` to list all these tasks.

Below is a brief breakdown of this project's timeline:

2017-10

- understand what an evidence providing problem solver is
- start absorbing the literature
- start the blog

2017-11

- solve equations on monoids
- start reading about solving equations on commutative rings

2017-12

- examine the existing solution for solving commutative rings
- start reading about solving Presburger arithmetic

2018-01

- start working on solving Presburger arithmetic

2018-02

- decide what decision algorithm to use for Presburger arithmetic
- write the background chapter
- polish the monoid solver
- write the chapter on solving monoids

2018-03

- solve Presburger arithmetic
- write the chapter on solving Presburger arithmetic
- write the chapter on solving commutative rings

8. Summary and conclusions

This report explores the construction of verified problem solvers for three distinct problem domains. § 3 provides a detailed description of a solver for equations on monoids; § 4 examines an existing solution for equations on commutative rings and draws parallelisms to the solution for monoids; § 5 is an ongoing attempt to define a solver for Presburger arithmetic in Agda.

Time constraints limited my work on the Presburger arithmetic solver. Much is still to be done, but the contributed work encompasses the heart of it — and provides an interface for a subset of Presburger formulae to be proven. Future work is outlined in § 5.3.5.

Perhaps most importantly, this project has been of invaluable educational significance for me — more on this in 7. I will likely aspire to find further entertainment within the field.

Bibliography

[Boutin, 1997] Boutin, S. (1997).
Using reflection to build efficient and certified decision procedures.
In *International Symposium on Theoretical Aspects of Computer Software*, pages 515–529. Springer.

[Bove et al., 2009] Bove, A., Dybjer, P., and Norell, U. (2009).
A brief overview of Agda – a functional language with dependent types.
In *International Conference on Theorem Proving in Higher Order Logics*, pages 73–78. Springer.

[Chaieb and Nipkow, 2003] Chaieb, A. and Nipkow, T. (2003).
Generic proof synthesis for presburger arithmetic.
Technical report, Technical report, Technische Universität München.

[Cooper, 1972] Cooper, D. C. (1972).
Theorem proving in arithmetic without multiplication.
Machine intelligence, 7(91-99):300.

[Curry, 1934] Curry, H. B. (1934).
Functionality in combinatory logic.
Proceedings of the National Academy of Sciences, 20(11):584–590.

[de Bruijn, 1972] de Bruijn, N. G. (1972).
Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem.
In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. Elsevier.

[Fischer and Rabin, 1974] Fischer, M. J. and Rabin, M. O. (1974).
Super-exponential complexity of presburger arithmetic.
Technical report, Massachusetts Inst of Tech Cambridge Project MAC.

[Grégoire and Mahboubi, 2005] Grégoire, B. and Mahboubi, A. (2005).
Proving equalities in a commutative ring done right in coq.
In *International Conference on Theorem Proving in Higher Order Logics*, pages 98–113. Springer.

[Howard, 1980] Howard, W. A. (1980).
The formulae-as-types notion of construction. hindley, jr, & seldin, jp (eds), to hb curry: Essays on combinatory logic, lambda calculus and formalism.

[Janičić et al., 1997] Janičić, P., Green, I., and Bundy, A. (1997).
A comparison of decision procedures in Presburger arithmetic.
University of Edinburgh, Department of Artificial Intelligence.

[Kokke and Swierstra, 2015] Kokke, P. and Swierstra, W. (2015).
Auto in agda.
In *International Conference on Mathematics of Program Construction*, pages 276–301. Springer.

[McBride and McKinna, 2004] McBride, C. and McKinna, J. (2004).
The view from the left.
Journal of functional programming, 14(1):69–111.

[Norell and Chapman, 2009] Norell, U. and Chapman, J. (2009).
Dependently typed programming in agda.
In *Advanced Functional Programming. LNCS 5832*, pages 230–266. Springer.

[Norrish, 2003] Norrish, M. (2003).
Complete integer decision procedures as derived rules in hol.
In *International Conference on Theorem Proving in Higher Order Logics*, pages 71–86. Springer.

[Norrish, 2006] Norrish, M. (2006).
Deciding presburger arithmetic.
http://ssll.cecs.anu.edu.au/files/slides/norrish.pdf.

[Oury and Swierstra, 2008] Oury, N. and Swierstra, W. (2008).
The power of pi.
ACM Sigplan Notices, 43(9):39–50.

[Presburger, 1929] Presburger, M. (1929).
Uber die vollstandigkeiteines gewissen systems der arithmetik ganzer zahlen, in welchen die addition als einzige operation hervortritt.
In *Comptes-Rendus du ler Congres des Mathematiciens des Pays Slavs*.

[Pugh, 1991] Pugh, W. (1991).
The omega test: a fast and practical integer programming algorithm for dependence analysis.
In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 4–13. ACM.

[Russinoff, 2017] Russinoff, D. M. (2017).
Polynomial terms and sparse horner normal form.
http://www.russinoff.com/papers/shnf.pdf.

[Sørensen and Urzyczyn, 2006] Sørensen, M. H. and Urzyczyn, P. (2006).
Lectures on the Curry-Howard Isomorphism (Studies in logic and the foundations of mathematics, 0049-237X; v. 149).
Elsevier Science Limited.

[Stansifer, 1984] Stansifer, R. (1984).
Presburger's article on integer arithmetic: Remarks and translation.
Technical report, Cornell University.

[van der Walt, 2012] van der Walt, P. (2012).
Reflection in agda.
Master's thesis.

[van der Walt and Swierstra, 2012] van der Walt, P. and Swierstra, W. (2012).
Engineering proof by reflection in agda.
In *Symposium on Implementation and Application of Functional Languages*, pages 157–173. Springer.

A. Program listing

The sources necessary to build this report — including the source files that contain all the quoted code listings — can be found in the `report` directory at https://github.com/umazalakain/fyp. All programs have been written for Agda 2.5.3.

Running `make modules` inside the `report` directory will compile all Agda code present in this report and any modules it depends on. The only required external library is Agda's standard library, obtainable from https://github.com/agda/agda-stdlib. (Installation instructions can be found at https://agda.readthedocs.io/en/v2.5.3/tools/package-system.html#example-using-the-standard-library.) Running `make` inside the `report` directory first compiles all source code and then outputs the report as PDF.

Dependent types accurately describe the behaviour of functions, often more so than documentation. The interesting parts of the source code are already commented in this work and therefore — in the interest of not repeating oneself and judging dependent types self-documenting enough — there has been **no significant documentation added to the source code** itself.

During the course of this project I fixed a small bug making spacing of compiled literate Agda programs inconsistent and unpleasant to the eye. The commit `8b83da6` can be cherry-picked from Agda's master branch.