Type Theory and Mechanized Reasoning
(Draft)

February 29, 2024

Contents

1 Introduction

Fill in this section.

2 Induction and Recursion

Induction is a technique in the standard mathematical toolkit for showing that a property holds of all objects in an inductively defined collection. There are three kinds of induction we will be interested in: ordinary induction on natural numbers[1] *strong* induction on natural numbers, and *structural* induction on general inductively-defined collections. We will also briefly discuss the connection between induction and recursion (saving a lengthier discussion for later).

2.1 Induction on Natural Numbers

The principle of (ordinary) induction on natural numbers says:

> In order to prove that a property hold of *all* natural numbers, it suffices to prove that it holds of 0 and that, for any natural number i, *if* the property holds of i, then it holds of $i + 1$ as well.

The first condition is called the **base case** and the second condition is called the **inductive step**. Here is a simple (and hopefully familiar) example.

Fact 1

For every natural number n

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Proof. We prove this by induction on n. For the base case, both sides of the equation are 0 when $n = 0$, so the equation holds in this case. For the inductive step, suppose

$$\sum_{i=1}^{j} i = \frac{j(j+1)}{2}$$

[1]The natural numbers form one of the simplest inductively defined sets.

for an arbitrary natural number j. Then

$$\begin{aligned}\sum_{i=1}^{j+1} i &= \left(\sum_{i=1}^{j} i\right) + j + 1 \\ &= \frac{j(j+1)}{2} + j + 1 \\ &= \frac{j(j+1) + 2(j+1)}{2} \\ &= \frac{(j+1)(j+2)}{2}\end{aligned}$$

which demonstrates that the equation holds when $n = j + 1$. □

In the final sequence of equalities of the above proof, the second line holds because we were able to *assume* that

$$\sum_{i=1}^{j} i = \frac{j(j+1)}{2}$$

This assumption is called the **induction hypothesis**; we are hypothesizing that the property holds of an arbitrary natural number. Much of the work of a proof by induction is in determining how to make use of the induction hypothesis (though this is not always the case!).

The principle of induction is often compared to the domino effect. Consider the following thought experiment: we line up an *infinitely long* sequence of dominoes—one on each natural number on the number line—and we tip over the domino at 0. We can "prove" by induction that every domino eventually falls.

Fact 2

For every natural number n, the domino at n eventually falls.

Proof. For the base case, the domino at 0 falls because we tipped it over. For the inductive step, suppose the domino at i eventually fall, for an arbitrary natural number i. Then the domino at $i + 1$ falls directly afterwards, which means that the domino at $i + 1$ eventually falls as well. □

2.2 Strong Induction over Natural Numbers

Some "inductive-looking" arguments can't be proved naturally by ordinary induction. Consider the following example. Recall that a natural number is prime if it has exactly two divisors: 1 and itself.

Fact 3

For every natural number n, if $n \geq 2$, then n can be expressed as a product of prime natural numbers.

Proof. Let's try to prove this by induction on n. The base case holds since $0 < 2$ (the same is also true when $n = 1$). For the inductive step, suppose that i can be expressed as a product of primes, where i is a natural number such that $i \geq 2$. The number $i + 1$ is prime, in which case it is already expressed as a product of primes, or it is composite and there are numbers a and b such that $i + 1 = ab$ and... □

An issue arises here. We'd like to "induct" on the factors of $i + 1$, express a and b as products of primes and multiply those expressions together. But we don't "know" that a and b can be expressed as products of primes, we only know that i can. The inductive hypothesis is not strong enough for this proof to be carried out this way.

The principle of *strong* induction says:

> In order to prove that a property holds of all natural numbers, it suffices to prove that for any natural number i, the property holds of i *if* it holds of *every* natural number j such that $j < i$.

This will clearly give us a strong inductive hypothesis; for our inductive step, rather than assuming that the property holds of i, we get to assume that it holds of *every natural number which is at most* i. I will leave it as an exercise to finish the above using the strong induction principle.

Remark 1. *I'd like to dwell for a moment on the distinction between ordinary induction and strong induction on natural numbers. Given the analogy to the domino effect, both forms of induction may seem obvious. And when we are introduced to both principles, the distinction may seem artificial;* Why would we every choose to use ordinary induction over strong induction? *My hope is that this will become more clear as we look to* formalizing *proofs by induction. In broad strokes, it's easy for a* *language like Agda (the language we will use throughout this course) to "recognize" that ordinary induction is well-founded. In our pen-and-paper proof, we're eliding a detail that may seem obvious to us, but requires a small amount of work in a formal proof, namely that any nontrivial divisor of a natural number n is* less than *n. As we will see, '$i < i + 1$' is* structurally *true, whereas that '$\frac{i}{2} < i$' requires proof; division is a* computation *whereas '$+1$' is a* constructor *(as we will see below).*

2.3 Induction on Inductively-defined Collections

In other course, we might take great care defining inductively-defined collections. Bus since we will ultimately take a computer scientific (specifically, type theoretic) perspective on induction, we will simply take an inductively-defined collection to be an algebraic data type (ADT).[2] If you don't recall exactly what an ADT is, we'll see them very soon in the context of Agda (and it may be helpful to peak at the next chapter). In the meantime, let's consider a simple example: lists. In a language with pointers, we might define a *linked* list as being one of two things:

- a grounding object like `null` representing empty;
- a node with an element and a point to *another linked list*.

In a function language like OCaml, lists are represented in a similar way:

```
type 'a list
  = Nil
  | Cons of 'a * 'a list
```

The "inductive" part of this definition is that lists are defined in terms of themselves. This might seem dubious at first; *How can we refer to lists in the definition of lists, before they're fully defined?*

`Nil` and `Cons` are **constructors** which means they represent values (e.g, they cannot be further "computed"). The reference to `list` in the definition of `list` means `Cons` can "build onto" other lists; we have to have already constructed another list (e.g., `Nil`).[3] Formally speaking, lists are *freely generated* by the constructors `Nil` and `Cons`.

[2]This is clearly not the most formal definition of inductively-defined collections, but it allows us to leverage our understanding of ADTs and functional programming, which tend to be stronger than our intuitions of, say, sets.

[3]This is not strictly true, even in OCaml. It is possible to write `let l = Cons (Nil, l)`, which would be the circular list of empty lists.

Natural numbers have a similar inductive characterization:

- zero is a natural number;
- if n is a natural number, then so is $\mathsf{suc}(n)$.

So there are natural numbers zero, suc(zero), suc(suc(zero)), and so on. We may think of the number of 'suc' constructors appearing in the value as the natural number represented by that value. We can write this in OCaml as an ADT:

```
type nat
  = Zero
  | Suc of nat
```

The constructors `Zero` and `Suc` are pronounced "zero" and "successor", respectively, where `suc(i)` represents the number $i + 1$ (i.e., '+1' is a constructor).

Once we have inductively-defined collections (i.e., ADTs) we can in essence do one of two things:

- define functions on ADTs via **pattern matching**;
- make formal claims about those functions.

The former is what we will take as what we mean when we use the term "recursion", again avoiding the weighty foundational questions about what recursion is, and instead building on our intuitions about writing programs with recursion. For the latter, we will need to be able to induct on the structure of ADTs.

The principle of structural induction on inductively-defined collections, is tricker to state, but is roughly says:

> In order to prove that a property holds of all values of an algebraic data type adt, it suffice to prove that if $\mathsf{Const}(\mathsf{e}_1, \ldots, \mathsf{e}_k)$ (or Const in the case that $k = 0$) is a value of adt and the property holds of all values of adt in $\{\mathsf{e}_1, \ldots, \mathsf{e}_k\}$, then it also holds of $\mathsf{Const}(\mathsf{e}_1, \ldots, \mathsf{e}_k)$.

This may be easier to parse in a specific case. The principle of structural induction on lists says:

> In order to show that a property holds of all lists it suffices to show that it holds of the empty list `Nil` and that the property holds of the nonempty list `Cons(x, xs)` given that it holds of `xs`.

Example 1

Consider the standard (non-tail-recursive) definition of list appending written in OCaml:

```
let rec append l r =
  match l with
  | Nil -> r
  | Cons(x, xs) -> Cons(x, append(xs, r))
```

One formal statement we may want to make about `append` is that

$$\texttt{append l Nil} = \texttt{l}$$

for any list `l`. This may seem obvious, but as defined `append l Nil` requires time linear in the size of `l` to terminate, so it stands to reason that after the computation has completed, it would take some effort to verify the property holds of the resultant list.

Proof. For the `Nil` case, `append Nil Nil` = `Nil` (the right-hand argument) by definition. For the `Cons` case,

$$\begin{aligned}\texttt{append Cons(x, xs) Nil} &= \texttt{Cons(x, append(xs, Nil))}\\ &= \texttt{Cons(x, xs)}\end{aligned}$$

where the last equality follows from the inductive hypothesis, i.e., that

$$\texttt{append(xs, Nil)} = \texttt{xs}$$

□

There is quite a bit more we could say about this, but we will leave it for when we start writing proves by induction using Agda. To conclude, I want to tease one point: one beautiful feature of a dependently typed language is that there is no distinction between induction and recursion. Proofs and data take on the same *ontological status* (there is little which is fundamentally different about the number 2 and a proof of, say, Goldbach's conjecture) which means that we will be able to "compute with" proofs. In particular, we will be able to pattern match *within* proofs when we want to prove by induction. This is mediated by what is called the *Curry-Howard Isomorphism*, something which will occupy us for much of the course.

2.4 Further Reading

Fill in this section.

3 A Brief Tour of Agda

Fill in this section.

4 Classical Propositional Logic

When we communicate, we make statements within and about the world. Some of these statements are true or false depending on the state of the world. The truth of the statement "it is raining", for instance, depends on the weather. But some statements seem to be true no matter the context, or even the appreciable content: "it is either raining or not raining" seems irrefutable and, more generally, "either x or not x" seems irrefutable no matter the statement which x stands for.[1] One of the goals of (classical) propositional logic is to model this phenomenon.

In the above example, the word "or" is an example of a *Boolean connective*.[2] Connectives are used to "connect" two or more statements to make a new statement whose truth is contingent on the truth of its constituent parts. Propositional logic may be understood as the study of Boolean connectives: *What are they? How do they work? How many are there? Which ones are necessary? Which ones are useful?*

Since this is the first logic we are studying, we need to lay out the rules of the game. To introduce a logic, we are (typically) required to present two things: the *syntax* and the *semantics*.

- *Syntax* refers to the rules which govern what count as well-formed statements in our logic. We will define syntax formally in the subsequent section, but roughly speaking, it is the concern of syntax to discern between "I ate my lunch in my office" and "I my lunch my office eating" as (in the first case) a statement which has the potential to be meaningful and (in the second case) a sequence of words (or linguistic units) which is nonsense.

- *Semantics* refers to the rules which govern the meaning of well-formed statements. It is the concern of semantics to discern between "I am cold" and "I am cold, and I shiver when I'm cold, but I'm not shivering" as (in the first case) a statement whose truth is contingent on the state of the world and (in the second case) one which cannot be true, no matter the state of the world.

[1]Those remotely familiar with the upcoming material may recognize this as *the law of the excluded middle*, and it may seem more or less refutable depending on your philosophical inclinations.

[2]"Boolean" refers to George Boole, who is considered a founder of modern (algebraic) logic.

Remark 2. *It is difficult to adequately stress how important it is to keep in mind that syntactic objects* have no inherent meaning. *When we take in the statement "I am cold and I have no jacket" (either by reading it or hearing it) we know almost immediately what our interlocutor means. But in the split-second before we've grokked the statement, it's just a sequence of characters (or sounds or linguistic units). We internally parse the statement and imbue the "and" in the statement with meaning so that we understand our interlocutor to be expressing that* both *statements on either side of the "and" are true.*

Perhaps this is a premature warning, but I think it is worth making explicit from the start.

In what follows, we present the syntax and semantics of classical propositional logic. The qualifier "classical" is unimportant now, but is meant to signal that we will eventually consider non-classical logics, e.g., intuitionistic logic. We will then prove a handful of meta-theoretic results about propositional logic, with an eye towards our next topic: SAT solvers.

4.1 Syntax

The basic syntactic objects of propositional logic are *propositional variables*, which are placeholders for unanalyzed atomic (i.e., not compound) sentences. Formally, we fix at the start a set propositional variable symbols.

Definition 1

Let Var *denote a set of* **propositional variable** *symbols* $\{x_1, x_2, x_3, \dots\}$.

We should come to think of propositional variables as akin to ordinary variables in algebra. We can even think of the expression '$2x + 5$' as a syntactic object, which we can give a value once we are given a value of x. We will be agnostic to the ontological status of variables (and syntactic objects in general) but it would not be difficult to find a philosopher to butt heads with about this.

Once we have variables, we can define the notation of well-formed statements, which we will subsequently refer to as *formulas*.

Definition 2

The collection of **propositional formulas***, denoted by* Prop*, is defined inductively as follows.*

- *Every propositional variable is a propositional formula.*

- *If P and Q are propositional formulas, then so are $(\neg P)$ and $(P \vee Q)$ and $(P \wedge Q)$.*

The symbols '$\neg$' and '$\vee$' and '$\wedge$' are pronounced "not" and "or" and "and", respectively. Remember, these formulas are syntactic objects, they have no inherent meaning until we given them meaning by defining our syntax. But that doesn't stop us from trying to model English sentences (with greater or less success) as propositional formulas.

Example 2

It is raining.	x_1
It is either raining or not raining.	$(x_1 \vee (\neg x_1))$
It is cold.	x_2
It it cold but not rainy.	$(x_2 \wedge (\neg x_1))$
I have my umbrella.	x_3
I did not forget my umbrella.	$(\neg(\neg x_3))$
Either it is raining or I have my umbrella.	$((x_1 \wedge (\neg x_3)) \vee ((\neg x_1) \wedge x_3))$

At this point, most standard texts on logic launch into a careful analysis of syntax: *Is a formula a sequence of symbols? Are parentheses part of our syntax? What about whitespace? Does every sequence of symbols represent at most one formula?* We will choose to ignore these questions and use the following principle: a propositional formula is not represented as a sequence of characters, but as a value of an algebraic data type (in line with the previous chapter), e.g., in Agda:

```
Var : Set
Var = Nat

infixr 6 _and_
infixr 5 _or_

data Prop : Set where
  var : Var -> Prop
  not : Prop -> Prop
  _and_ : Prop -> Prop -> Prop
  _or_ : Prop -> Prop -> Prop
```

When we write down a formula in linear fashion (e.g., $(x_1, \vee(x_2 \wedge (\neg x_3))))$ it is shorthand for a value of this ADT. And to simplify the process of writing down a formula in linear fashion, we will follow the principles which are standard in logic and, for students of computer science, may be somewhat familiar: they are the same rules for parsing compound conditionals in most programming languages.

- We will use any reasonable names for propositional variables, e.g., x, y, z, w, etc.
- We will ignore outer parentheses, so $x \wedge y$ is shorthand for $(x \wedge y)$.
- Negation takes highest precedence, so $\neg x \vee y$ is shorthand for $(\neg x) \vee y$.
- Conjunction takes higher precedence than disjunction so $x \wedge y \vee z$ is shorthand for $(x \wedge y) \vee z$.
- Conjunction is *right-associative* so $a \wedge b \wedge c \wedge d$ is shorthand for $a \wedge (b \wedge (c \wedge d))$. Same for disjunction.

The hope is that, given a sequence of symbols (i.e., a formula written in linear fashion) we can determine its associated value in Prop, and if we can't we ask the person who wrote it to clarify (rather that pinning the blame on the syntax itself).

Example 3

The following are the same formulas as in Example 2, but expressed in this nicer shorthand, as well as in Agda.

r	`r = var 1`
$r \vee \neg r$	`r or not r`
c	`c = var 2`
$c \wedge \neg r$	`c and not r`
u	`u = var 3`
$\neg\neg u$	`not (not u)`
$(r \wedge \neg u) \vee (\neg r \wedge u)$	`(r and not u) or (not r and u)`

4.2 Semantics

Our next task is to determine the meaning of well-formed formulas. At the risk of becoming overly-philosophical, this requires briefly discussing the meaning of "meaning". For classical logic, meaning tends to be *truth*. We will consider other notions of meaning (e.g., for intuitionistic logic) but it suffices for now to take our task to be determining the truth or falsity of a given propositional formula.

Naturally, there is more to it than this. Consider the formula x (this is a formula because all variables are formulas). Whether or not x is true depends on the value given to x. That is, the truth depends on the state of the world (or a *state of affairs* in the Pears/McGuinness translation of Wittgenstein's *Tractatus*).

Definition 3

A **valuation** *is a function from* Var *to the set* $\{0, 1\}$.

The value 1 represents truth and the value 0 represents falsity (this will come to be convenient). If instead we use `Bool` in Agda, we can write this definition as a type synonym:

```
Val : Set
Val = Var -> Bool
```

If we think of propositional variables as standing for the possible atomic statements that can be made of the world, then a valuation expresses how the world is, what is and is not the case.

Given a valuation v, we can determine the truth-value of x, it's just $v(x)$; the valuation tells use whether or not x is the case. What's more, we can extend a valuation to apply to *any* formula. Hopefully this is seems reasonable: if we know the truth or falsity of every atomic statement, we should be able to determine the truth or falsity of every compound statement.

Definition 4

Given a valuation v, **evaluation** *with respect to v, written $\overline{v} : \mathsf{Prop} \to \{0, 1\}$, is defined recursively as follows.*

$$\overline{v}(x) = v(x) \text{ where } x \in \mathsf{Var}$$

$$\overline{v}(\neg P) = \begin{cases} 1 & \overline{v}(P) = 0 \\ 0 & \textit{otherwise} \end{cases}$$

$$\overline{v}(P \vee Q) = \begin{cases} 1 & \overline{v}(P) = 1 \textit{ or } \overline{v}(Q) = 1 \\ 0 & \textit{otherwise} \end{cases}$$

$$\overline{v}(P \wedge Q) = \begin{cases} 1 & \overline{v}(P) = 1 \textit{ and } \overline{v}(Q) = 1 \\ 0 & \textit{otherwise} \end{cases}$$

This definition may look as though it's not doing anything. We define the truth value of disjunction using the word "or" and that of conjunction using the word "and". But this is the point. We're "transferring" from syntax (the symbol '$\wedge$') to semantics (the word "or"). In Agda:

```
eval : Val -> Prop -> Bool
eval v (var x) = v x
eval v (not p) = notb (eval v p)
eval v (p and q) = eval v p && eval v q
eval v (p or q) = eval v p OR eval v q
```

Remark 3. *The evaluation function is performing exactly what a calculator does with an arithmetic expression: given '*$2+3$*' it "change" the '+' symbol into the* `plus` *operation and applies it to the arguments* 2 *and* 3*. We've just built a propositional truth-value calculator.*

4.3 Meta-Theory

Once we have a logic, we're often not so interested in reasoning within the logic (though we will come back to this when we start using SAT solvers) but rather, what can be said "about" reasoning within the logic (this is where the 'meta' part of meta-theory comes from). In order to reason about propositional logic, we need a couple important semantic notions.

Definition 5

A formula ϕ is **valid** *(or is a* **tautology***) if $\overline{v}(\phi) = 1$ for* every *valuation v. A formula ϕ is* **satisfiable** *if $\overline{v}(\phi) = 1$ for* some *valuation v.*

Validity corresponds to the phenomenon noted at the opening of the chapter, that there are statements which seems unassailable, no matter the context. Such statements over propositional variables which stand for their constituent parts (e.g., either x or not x) are exactly the valid propositional formulas.

Example 4

In the syntactic shorthand we introduced at the end of the last section, the example statement given in the introduction of this chapter is $x \vee \neg x$. This is an example of a tautology. To demonstrate this, suppose that v is an arbitrary valuation. If $v(x) = 1$, then by definition $\overline{v}(x \vee \neg x) = 1$. If $v(x) = 0$, then by definition $\overline{v}(\neg x) = 1$, which implies $\overline{v}(x \vee \neg x) = 1$. Therefore, no matter the valuation (and in particular, no matter how the valuation assigns truth or falsity to x) the formula $x \vee \neg x$ is made true.

The formula $(r \wedge \neg u) \vee (\neg r \wedge u)$ is not valid. Let v be a valuation such that $v(r) = 1$ and $v(u) = 1$; it may assign truth values to all other variables as it pleases. Since $v(u) = 1$, it follows that $\overline{v}(\neg u) = 0$ and, therefore, $\overline{v}(r \wedge \neg u) = 0$. Similarly, we may demonstrate that $\overline{v}(\neg r \wedge u) = 0$. Therefore, $\overline{v}((r \wedge \neg u) \vee (\neg r \wedge u)) = 0$ (as both constituent parts of the disjunction are false).

This formula *is* satisfiable. Let v be a valuation such that $v(r) = 1$ and $v(u) = 0$. Then $\overline{v}(r \wedge \neg u) = 1$, which then makes the entire disjunction true.

Remark 4. *At this point take advantage of the fact that truth values are represented by numbers. This allows us to write our evaluation function in terms of function on numbers:*

$$\begin{aligned}
\overline{v}(\neg P) &= 1 - \overline{v}(P) \\
\overline{v}(P \vee Q) &= \max(\overline{v}(P), \overline{v}(Q)) = \left\lceil \frac{\overline{v}(P) + \overline{v}(Q)}{2} \right\rceil \\
\overline{v}(P \wedge Q) &= \min(\overline{v}(P), \overline{v}(Q)) = \overline{v}(P) * \overline{v}(Q)
\end{aligned}$$

The argument that $x \vee \neg x$ is a tautology reduces to recognizing that $\max(1 - x, x) = 1$ *if $x = 0$ or $x = 1$. For another example, if we want to demonstrate that $x \wedge \neg x$ is* unsatisfiable, *we simply have to recognize that*

$$\overline{v}(x \wedge \neg x) = v(x) * (1 - v(x)) = 0$$

no matter the value of $v(x)$.

Validity and satisfiability are dual notions (in the way that "for every" and "for some" are dual notions). This is demonstrated nicely by the following fact.

Fact 4

A formula ϕ is valid if and only if $\neg\phi$ is unsatisfiable.

I will leave it to the reader to convince themselves of the fact, it follows immediately from the definitions. This allows us to reduce the problem of determining if a formula is valid to determining if its negation is unsatisfiable (e.g., by a SAT solver).

A majority of formulas are not valid,[3] and ultimately, absolute validity is not as useful as validity *relative* to a collection of assumptions. The statement "I am cold" is not a tautology, it depends on the state of the world (e.g., my ability to thermo-regulate). But *assuming* I am cold and that I shiver when I'm cold, it should then follow that I am shivering. This is the notion of entailment.

Definition 6

A set of formulas Γ **entails** *the formula ϕ, written $\Gamma \models \phi$, if every valuation which satisfies every formula in Γ also satisfies ϕ.*

Notation 1. *For two formulas ϕ and ψ, we will write '$\phi \models \psi$' instead of '$\{\phi\} \models \psi$'. We will write $\Gamma \not\models \phi$ if Γ does not entail ϕ.*

Example 5

As a simple example, $P \land Q \models P$, as any valuation v such that $\overline{v}(P \land Q) = 1$ must make $\overline{v}(P) = 1$ by definition (alternatively, if $\overline{v}(P) * \overline{v}(Q) = 1$, then it must be that $\overline{v}(P) = 1$).

As a more complicated example, $P \land Q \models \neg(\neg P \lor \neg Q)$. Let v be a valuation such that $\overline{v}(P \land Q) = 1$, so $\overline{v}(P) = 1$ and $\overline{v}(Q) = 1$. Using our alternative definition of evaluation, we just need to recognized that

$$\overline{v}(\neg(\neg P \lor \neg Q)) = 1 - \max(1 - \overline{v}(P), 1 - \overline{v}(Q)) = 1$$

That this entailment holds is to say that, if P and Q are true, then it is impossible that neither P nor Q is true. Hopefully this rings true to your intuitions about natural language.

To show that an entailment does not hold, we have to exhibit a valuation which makes the left-hand-side true and the right-hand-side false. For example, we have $x \lor y \not\models x \land y$ since we can consider any valuation v such that $v(x) = 1$ and $v(y) = 0$. If one of two atomic statements is true, it does not necessarily mean that both are true.

The last semantic notion we consider for now will, again, feel familiar to students of computer science.

Definition 7

The formulas ϕ and ψ are **logically equivalent***, written $\phi \equiv \psi$, if $\overline{v}(\phi) = \overline{v}(\psi)$ for every valuation v. Equivalently $\phi \models \psi$ and $\psi \models \phi$.*

[3]Despite this, determining which formulas are valid is a computationally difficult problem.

Formulas are logically equivalent if they "look the same" from the perspective of evaluation. Logically equivalence captures the experience we may have as programmers when we realize we can rewrite the condition of a conditional statement in a simpler way. When we replace a Boolean expression with something else and the behavior of the program does not change, we are replacing it with something which, as a formula, is logically equivalent.

Example 6

We may write code in Python like:

```
if (a or b) or (not b and c):
    # DO SOMETHING
```

only to realize that we could have written:

```
if (a or b) or c:
    # DO SOMETHING
```

because, if the first disjunction evaluates to `False`, then it must be that the expression `(not b)` evaluates to `True`.

Example 7

In a different course (e.g., one more philosophically or historically oriented) we may spend more time looking a number of important *named* logical equivalences. For our purposes, we will focus on three. The first is *double negation elimination*:

$$P \equiv \neg\neg P$$

which, in our numeric definitions of evaluation, amounts to the fact that

$$\overline{v}(P) = 1 - (1 - \overline{v}(P))$$

for any choice of valuation v. The next two are *De Morgan's Laws:*

$$P \land Q \equiv \neg(\neg P \lor \neg Q)$$
$$P \lor Q \equiv \neg(\neg P \land \neg Q)$$

or as equations:

$$\overline{v}(P) * \overline{v}(Q) = 1 - \max(1 - \overline{v}(P), 1 - \overline{v}(Q))$$
$$\max(\overline{v}(P), \overline{v}(Q)) = 1 - (1 - \overline{v}(P)) * (1 - \overline{v}(Q))$$

for any choice of valuation v. I will leave it to the reader to convince themselves that the above equations hold (remember, evaluation can only yield the value 0 or 1, these equations do not hold otherwise).

As you may expect, logical equivalence is an equivalence relation[4] Perhaps more importantly, it is *compatible* with the connectives in our logic.

Fact 5

For any propositional formulas P, P', Q, Q', if $P \equiv P'$ and $Q \equiv Q'$, then

$$\neg P \equiv \neg P'$$
$$P \vee Q \equiv P' \vee Q'$$
$$P \wedge Q \equiv P' \wedge Q'$$

This means we can replace and *sub-formula* with something which is logically equivalent and it will remain logically equivalent.

Example 8

When proving logical equivalence, it is common to "pick up" double-negations in the application of other equivalences:

$$\neg(\neg P \vee Q) \equiv \neg\neg P \wedge \neg Q \equiv P \wedge \neg Q$$

The first equivalence is an application of De Morgan's law. The second is an application of double-negation elimination *on the first conjunct* of $\neg\neg P \wedge Q$.

4.4 Functional Completeness

We now aim to prove a nontrivial meta-theoretic result about propositional logic. As a warmup, if you've seen any amount of logic—for example, in a

[4]This means for any formulas P, Q, and R, we have (1) $P \equiv P$ and (2) $P \equiv Q$ implies $Q \equiv P$ and (3) $P \equiv Q$ and $Q \equiv R$ implies $Q \equiv R$.

discrete mathematics course—you may be thinking: *What about implication? Isn't 'P implies Q' a fundamental form of reasoning? Haven't we already used implication in our example of propositional statements in natural language?*

Implication does not appear explicitly in our logic because it don't need to.[5] That is to say, any statement we could write with an implication symbol '$\to$' is *logically equivalent* to a statement we can already write in our logic.

Remark 5. *This is not to say that there wouldn't be a benefit to explicitly including implication. If we, for instance, cared about the* size *of formulas (measured as number of connectives) in logic, including implication would make a difference. But if we only care about formulas* up to *logical equivalence, then implication can be defined* within *the logic.*

As a thought experiment, suppose that we included the following case in our definition of propositional formulas (Definition 2):

- If P and Q are propositional formulas, then so is $(P \to Q)$.

This requires us to include a case in our definition of evaluation (Definition 4). When should '$P \to Q$' be considered true? Hopefully we can intuit that it should be false if P is true and Q is false.

But what about if P is false? Well, to say "when pigs fly" (or any other idiom to this effect) is to say that *if* pigs fly then x is certainly true, where x an be any (usually ridiculous) statement. So when P is false, it doesn't matter whether or not Q is true, the whole implication is true. This means including the following case to Definition 4:

$$\overline{v}(P \to Q) = \begin{cases} 0 & \overline{v}(P) = 1 \text{ and } \overline{v}(Q) = 0 \\ 1 & \text{otherwise} \end{cases}$$

We should already be suspicious of this inclusion because the truth-value of an implication seems to depend only on conjunction. Within our logic augmented with implication, we have the following fact.

Fact 6

$P \to Q \equiv \neg(P \land \neg Q) \equiv \neg P \lor Q$ for any formulas P and Q.

[5] I chose the collection of connectives in our presentation in part because they should are the most familiar to programmers.

Proof. Let v be a valuation such that $\overline{v}(P \to Q) = 1$. This implies $\overline{v}(P) \neq 1$ or $\overline{v}(Q) \neq 0$. If $\overline{v}(P) = 0$, then $\overline{v}(\neg P) = 1$ and so $\overline{v}(\neg P \vee Q) = 1$. If $\overline{v}(Q) = 1$, then $\overline{v}(\neg P \vee Q) = 1$ as well. This shows $P \to Q \models \neg P \vee Q$.

The proof that $\neg P \vee Q \models P \to Q$ is similar, and the proof that $P \to Q \equiv \neg(P \wedge \neg Q)$ is similar as well. □

We will now freely write formulas like '$P \to Q$' as shorthand for '$\neg P \vee Q$'. In Agda, this is akin to writing implication as a *function* instead of a *constructor*.

```
implies : Prop -> Prop -> Prop
implies : p q = not (p and q)
```

We *could* take this further: De Morgan's laws tell us that

$$P \vee Q \equiv \neg(\neg P \wedge \neg Q) \quad \text{and} \quad P \wedge Q \equiv \neg(\neg P \vee \neg Q)$$

which means we only really need disjunction or conjunction but not both. In Agda, for example:

```
or' : Prop -> Prop -> Prop
or' p q = not (not p and not q)

and' : Prop -> Prop -> Prop
and' p q = not (not p or not q)
```

This warmup is meant to preempt a more general question: *Are there any logical connectives we could add to our logic that would make it more expressive? Or is every connective we could add logically equivalent to a formula we could write using '$\neg$' and '$\vee$'?* This is the question of *functional completeness*.

We first need to think more carefully about what a connective is. Take, for instance, a more complex connective, a ternary (three argument) connective for if-then-else reasoning.

Example 9

The formula $P \mathbin{?} Q : R$ represents the statement 'if P then Q else R'. We can add this to our logic and expand our evaluation function to include:

$$\overline{v}(P \mathbin{?} Q : R) = \begin{cases} \overline{v}(Q) & \overline{v}(P) = 1 \\ \overline{v}(R) & \text{otherwise} \end{cases}$$

It is not difficult to imagine that more complicated connectives would warrant more complicated truth-value calculations. But in any case, the kind of calculation is the same: we're applying a *function* to the truth-values of the constituent parts of the statement.

Definition 8

*A n-**variate** Boolean function is a function from $\{0, 1\}^n$ to $\{0, 1\}$.*

In general we can think of an arbitrary Boolean connective as a Boolean function, one which tells us how to calculate the truth-value of the new compound statement it represents in terms of the truth-values of its constituent parts. In Agda, we can use dependent types to create a type synonym for Curried Boolean function:

```
BoolFun : (n : Nat) -> Set
BoolFun zero = Bool
BoolFun (suc n) = Bool -> BoolFun n
```

Example 10

We can write the Boolean function used for evaluating statements of the form $P \mathbin{?} Q : R$ in Agda:

```
if-then-else : BoolFun 3
if-then-else true q r = q
if-then-else false q r = r
```

We can now rephrase the question of functional completeness in terms of Boolean functions.

Definition 9

*An n-variate Boolean function is **represented** by the formula ϕ is*

$$\overline{v}(\phi) = F(v(x_1), \ldots, v(x_n))$$

*for all valuations v. A set of connectives $\mathcal{C}$ is **functionally complete** if every Boolean function (on any number of arguments) is represented by a formula using the connectives in C.*

Notation 2. *A formula ϕ over the variables $x_1, \ldots, x_n$ also defines a Boolean function. We write $F_\phi : \{0, 1\}^n \to n$ for the Boolean function given by*

$$F_\phi(b_1, \ldots, b_n) = \overline{v_{b_1,\ldots,b_n}}(\phi)$$

where $v_{b_1,\ldots,b_n}(x_i) = b_i$ for $i \in \{1, \ldots, n\}$, and $v_{b_1,\ldots,b_n}(x) = 0$ otherwise. When we refer to such a function for a single connective, we will just write the connective, e.g., we will write $F_\wedge$ instead of $F_{x_1 \wedge x_2}$.

Example 11

(Exclusive Disjunction) The proposition $P \oplus Q$ represents the statement "either P or Q but not both" or "exactly one of P and Q holds". The Boolean function for exclusive disjunction is

$$F(x_1, x_2) = x_1 + x_2 \pmod 2$$

and this function is represented by the formula

$$(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2)$$

We can prove this by working through all possible values of x_1 and x_2 (we saw two cases in Example 4).

Theorem 1

$\{\neg, \vee, \wedge\}$ *is a functionally complete set of connectives.*

We'll come back to the proof of this theorem in the next section. For now, we can use this theorem to bootstrap the proofs of functional completeness for other sets of connectives.

Fact 7

If $\mathcal{C}$ is a complete set of connectives and for every connective $\Box \in \mathcal{C}$, the function $F_\Box$ is represented by a formula using the connectives in $\mathcal{D}$, then $\mathcal{D}$ is also functionally complete.

Corollary 1

$\{\neg, \vee\}$ *and* $\{\neg, \wedge\}$ *are functionally complete sets of connectives.*

Proof. The function $F_\neg$ is represented by a $\neg x_1$ and, by De Morgan's law, the function $F_\wedge$ is represented by $\neg(\neg x_1 \vee \neg x_2)$, so $\{\neg, \vee\}$ is functionally complete. Similarly for $\{\neg, \wedge\}$. $\Box$

4.5 Conjunctive Normal Form

We conclude the chapter by discussing normal forms, which will lead to a proof of Theorem 1 above. In the next chapter, we will cover algorithms for determining if a formula is satisfiable. In this algorithmic setting, we want more control over the "shape" of the formula given as input. Logically equivalent formulas which have nice properties tend to be called *normal forms*. This is akin to *Jordan normal forms*, which are representations of matrices that are in some sense equivalent (they represent the same linear operator) and in some sense have nicer properties (they're upper triangular, their diagonal elements are eigenvalues, etc.).

Definition 10

A **literal** *is a propositional variable or its negation (e.g., x or $\neg y$). A* **clause** *is a disjunction of literals (e.g., $x \vee \neg y$). A* **conjunctive normal form (CNF)** *formula is a conjunction of clauses (e.g., $(x \vee \neg y) \wedge (y \vee z)$).*

Notation 3. *A literal is said to be* **positive** *if it is just a variable, and* **negative** *if it is the negation of a variable. It will be convenient to write x^0 for the negative literal $\neg x$ and x^1 for the positive literal x. In particular, it is convenient that x^{1-a} is the literal with opposite polarity, and that x^a is satisfied by a valuation v if $v(x) = a$.*

We will use '$\bigwedge$' and '$\bigvee$' notation (similar to '$\prod$' and '$\sum$' notation for products and sums) to represent large clauses and CNF formulas, e.g.,

$$\bigvee_{i=1}^{k} l_i \qquad \textit{and} \qquad \bigwedge_{i=1}^{m} C_i$$

One nice feature of CNF formulas is that the question of their satisfiability has a simple combinatorial framing.

Fact 8

A CNF formula is satisfiable if and only if there is a valuation which satisfies at least one *literal in* every one *of its clauses.*

Another is that they have simple representation in code:

```
Lit : Set
Lit = Var & Bool
```

```
Cls : Set
Cls = List Lit

CNF : Set
CNF = List Cls

eval-lit : Val -> Lit -> Bool
eval-lit v (x , b) = v x ==b b

eval-cls : Val -> Cls -> Bool
eval-cls v = anyL (eval-lit v)

eval-cnf : Val -> CNF -> Bool
eval-cnf v = allL (eval-cls v)
```

And (part of the point) up to logical equivalence, they are sufficient for proposition logic.

Theorem 2

Every proposition formula is logically equivalent to a CNF formula.

We will also delay the proof of this theorem, and take a minor detour through another standard normal form. A **disjunctive normal form (DNF)** formula is the same as a CNF formula with conjunction and disjunction swapped; that is, a DNF formula is a disjunction of conjunctions of literals. DNF formulas will not be our focus, but they are a useful intermediate representation on our way to proving Theorem 2.

Theorem 3

Every Boolean function is represented by a DNF formula.

Proof. The power of DNF formulas comes from their ability to simulate *truth tables*. In essence, a truth table of a Boolean function is the entire list of its input-output behavior.

Our first observation: conjunctions represent functions which have exactly one input whose output is 1. Let $\mathbf{1}_{b_1,\dots,b_n} : \{0,1\}^n \to \{0,1\}$ denote the Boolean function where

$$\mathbf{1}_{b_1,\dots,b_n}(x_1,\dots,x_n) = \begin{cases} 1 & x_i = b_i \text{ for } i \in \{1,\dots,n\} \\ 0 & \text{otherwise} \end{cases}$$

I will leave it as an exercise to verify that $\mathbf{1}_{b_1,\dots,b_n}$ is represented by the formula

$$x_1^{b_1} \land \cdots \land x_n^{b_n}$$

If conjunctions represent functions with a single input whose output is 1, then a *disjunction* of conjunctions should represent a function which has output 1 on *any of the inputs* which make one of its disjuncts output 1. For example, $F_\oplus(x_1, x_2) = 1$ exactly when $x_1 = 1$ and $x_2 = 0$ or $x_1 = 0$ and $x_2 = 1$. Hence, we get the representative formula

$$(x_1^1 \land x_2^0) \lor (x_1^0 \land x_2^1)$$

For an arbitrary function F, we can "collect together" the inputs which have output 1 into a single formula.

The **support** of a Boolean function F, written $\mathsf{supp}(F)$, is the collection of all inputs $(b_1, \dots, b_n)$ such that $F(b_1, \dots, b_n) = 1$. The formula

$$\bigvee_{(b_1,\dots,b_n)\in\mathsf{supp}(F)} \left(\bigwedge_{i=1}^{n} x_i^{b_i} \right)$$

is a DNF formula which is true exactly when one of its disjuncts is true, which occurs exactly when the values assigned to $x_1, \dots, x_n$ are in the support of F (i.e., when F outputs 1). Therefore, this formulas represents F. □

Note that this immediately implies Theorem 1 since DNF formulas are written with the connectives in $\{\neg, \land, \lor\}$. We also can proof Theorem 2 as a corollary by a neat trick.

Proof. (of Theorem 2) Let ϕ be an arbitrary propositional formula. By double negation elimination, we have $\phi \equiv \neg\neg\phi$. Since $F_{\neg\phi}$ is represented by a DNF formula ψ (by Theorem 3) we also have $\neg\phi \equiv \psi$. Now the trick: the *negation* of a DNF formula can readily be shown to be logically equivalent to a CNF formula. This requires a generalization of De Morgan's laws:

$$\neg \bigwedge_{i=1}^{m} P_i \equiv \bigvee_{i=1}^{m} \neg P_i \qquad \neg \bigvee_{i=1}^{m} P_i \equiv \bigwedge_{i=1}^{m} \neg P_i$$

With this, we can "push" the negation in $\neg\psi$ through the disjunction and conjunctions, swapping them in the process, and ending with a CNF formula.

$$\begin{aligned}
\neg\psi &\equiv \neg \bigwedge_{i=1}^{m} \left(\bigvee_{j=1}^{k_i} y_j^{a_j} \right) \\
&\equiv \bigvee_{i=1}^{m} \neg \left(\bigvee_{j=1}^{k_i} y_j^{a_j} \right) \\
&\equiv \bigvee_{i=1}^{m} \left(\bigwedge_{j=1}^{k_j} \neg y_j^{a_j} \right) \\
&\equiv \bigvee_{i=1}^{m} \left(\bigwedge_{j=1}^{k_i} y_j^{1-a_j} \right)
\end{aligned}$$

Finally we have that $\phi \equiv \neg\neg\phi \equiv \neg\psi$ and $\neg\psi$ was shown to be logically equivalent to a CNF formula, so ϕ is also logically equivalent to a CNF formula. □

4.6 Further Reading

Fill in this section.

5 SAT Solvers

The CNF satisfiability problem (CNF-SAT or just SAT for short) is the computational decision problem of determining if a given CNF formula is satisfiable. SAT is a fundamental problem in complexity theory because a vast number of decision problems reduce to it (e.g., there is a polynomial-time transformation T from graphs to formulas such that a graph G is 3-colorable if and only if $T(G)$ is satisfiable). This is to say, if we could solve SAT efficiently, we could solve this vast number of decision problems efficiently. This is encapsulated in the *Cook-Levin Theorem.*[1]

Theorem 4

SAT is NP-complete.

As this is not a complexity theory course, we won't dwell on the this. It suffices to say that SAT is believed to be a computationally intractable problem. And yet, there is a community of computer scientists and engineers constructing algorithms—and, perhaps more importantly, implementing algorithms—for SAT. These software tools are collectively called *SAT solvers* and, despite the supposed difficulty of SAT, they are used for industrial and research applications to determine the satisfiability of CNF formulas with millions of variables and clauses.

Remark 6. *An important caveat to this story-line is that NP-completeness is about* worst-case *complexity. This does not rule out the possibility that SAT is tractable for specific subsets of CNF formulas. It is generally believed that the formulas which appear in practice are relatively easy to solve, and it is an active area of research to better understand the* structure *of CNF formulas generated by industry.*

We will consider a simple algorithm for SAT called the *Davis-Putnam-Logemann-Loveland procedure* (DPLL for short) named after its discoverers. DPLL is a backtracking algorithm, similar to the kind used to solve games, and despite its simplicity, it is the basis of many state-of-the-art SAT solvers. We begin by building up some language for *partial assignments*, which we refer to as *assignments*, below.

[1]Students at Boston University are in a unique position to appreciate this result given that Leonid Levin is (as of writing this) still a faculty member of the computer science department.

5.1 Restriction

We defined evaluation with respect to a valuation, which provided truth-values for *all* propositional variables. When we define DPLL, we're going to build a *partial* valuation one literal at a time (this is how we're going to "branch-and-bound".

Definition 11

An **assignment** *is a set of literals in which no literal appears both positively and negatively.*

We can think of an assignment as a valuation in which we do not commit to a truth-value for every variable. In Agda, we could use either of the two representations:

```
PVal : Set
PVal = Var -> Maybe Bool

Asn : Set
Asn = List Lit

ans-to-pval : Asn -> PVal
ans-to-pval [] v = Nothing
ans-to-pval ((x , b) :: ls) v with x == v
... | true = Just b
... | false = ans-to-pval ls v
```

Given an assignment α which assigns values to every variable appearing in a formula ϕ, we can determine the truth-value of ϕ. If there is a variable in ϕ which is not assigned by α, it may still be possible to determine the truth-value of ϕ. Take, for example, the assignment $\{x^1\}$ and the formula $x^1 \vee y^0$. Since x is assigned to be true, it doesn't matter what value y is assigned to be, the truth-value of ϕ is 1.

But it may be that we cannot determine the truth-value of ϕ. For the assignment $\{x^0\}$ and formula $x^1 \vee y^0$, the truth-value depends on the value given to y. And we can express that dependence as another CNF formula: y^0. If α' satisfies y^0, then $\alpha \cup \alpha'$ satisfies $x^1 \vee y^0$, e.g., since $\{y^0\}$ satisfies y^0, the assignment $\{x^0, y^0\}$ satisfies $x^1 \vee y^0$.

This is the notion of restriction. Given an assignment, we can either determine the truth value of a CNF formula, or we can find a new (smaller) formula which is equisatisfiable.

Definition 12

A transformation T : Prop $\to$ Prop **preserves satisfiability** *given that for any formula ϕ, if ϕ is satisfiable then so is ϕ. We say that ϕ and $T(\phi)$ are* **equisatisfiable** *(or that T is an equisatisfying transformation) if ϕ is satisfiable if and only if $T(\phi)$ is satisfiable.*

Equisatisfiability will play a greater role later in this section. It is important to note that if two formulas are equisatisfiable, they are not necessarily logically equivalent. If T is the transformation $\phi \mapsto \phi \wedge x$, where x is a variable does not appear in ϕ, then ϕ and $T(\phi)$ are equisatisfiable, but not logically equivalent.

Definition 13

The **restriction** *of a CNF formula ϕ by an assignment α, written $\overline{\alpha}(\phi)$, is defined as follows.*

$$\overline{\alpha}(x^a) = \begin{cases} 1 & x^a \in \alpha \\ 0 & x^{1-a} \in \alpha \\ x^a & \textit{otherwise} \end{cases}$$

$$\overline{\alpha}\left(\bigvee_{i=1}^{k} l_i\right) = \begin{cases} 1 & \overline{\alpha}(l_i) = 1 \textit{ for some } i \in \{1, \dots, k\} \\ 0 & \overline{\alpha}(l_i) = 0 \textit{ for all } i \in \{1, \dots, k\} \\ \left(\bigvee_{i=1}^{k} \overline{\alpha}(l_i)\right) \setminus \{0\} & \textit{otherwise} \end{cases}$$

$$\overline{\alpha}\left(\bigwedge_{i=1}^{m} C_i\right) = \begin{cases} 1 & \overline{\alpha}(C_i) = 1 \textit{ for all } i \in \{1, \dots, m\} \\ 0 & \overline{\alpha}(l_i) = 0 \textit{ for some } i \in \{1, \dots, m\} \\ (\bigwedge_{i=1}^{m} \overline{\alpha}(C_i)) \setminus \{1\} & \textit{otherwise} \end{cases}$$

Notation 4. *In the case that $\alpha = \{x^a\}$, we will write $\phi|_{x=a}$ for the restriction $\overline{\alpha}(\phi)$.*

Example 12

The following is a simple example of restriction by a single variable assignment.

$$\left((x^0 \vee y^1 \vee z^1) \wedge x^1 \wedge (x^0 \vee z^0)\right)|_{x=1} = (y^1 \vee z^1) \wedge z^0$$

This is a more complicated to implement in Agda because we need to work with `Either`, but it follows the same pattern:

```
r-lit : Asn -> Lit -> Either Lit Bool
r-lit [] l = left l
r-lit ((x , b) :: ls) (y , c) with x == y
... | true =  right (b ==b c)
... | false = r-lit ls (y , c)

r-cls : Asn -> Cls -> Either Cls Bool
r-cls a [] = right false
r-cls a (l :: ls) with r-lit a l
... | left l = mapE (_::_ l) (r-cls a ls)
... | right true = right true
... | right false = r-cls a ls

restrict : Asn -> CNF -> Either CNF Bool
restrict a [] = right true
restrict a (c :: cs) with r-cls a c
... | left c = mapE (_::_ c) (restrict a cs)
... | right true = restrict a cs
... | right false = right false

restrict1 : Lit -> CNF -> Either CNF Bool
restrict1 l f = restrict (l :: []) f
```

5.2 DPLL

With restriction, DPLL is simple to implement. It's based on the following observation:

> For any variable x, the formula ϕ is satisfiable if and only if it is satisfiable by a valuation v such that $v(x) = 1$ or by a valuation v such that $v(x) = 0$.

Or, in the language of restrictions:

> ϕ is satisfiable if and only if $\phi|_{x=0}$ is satisfiable or $\phi|_{x=1}$ is satisfiable.

As a warm-up, let's write a naive version of DPLL.

```
{-# TERMINATING #-}
naive-dpll : CNF -> Bool
naive-dpll f = naive-dpll' (left f) where mutual
  naive-dpll' : Either CNF Bool -> Bool
  naive-dpll' (right b) = b
  naive-dpll' (left f) = go f

  go : CNF -> Bool
  go [] = true
  go ([] :: _) = false
  go (((x , b) :: ls) :: cs)
    with naive-dpll' (restrict1 (x , b) cs)
  go (((x , b) :: ls) :: cs)
    | true = true
  go (((x , b) :: ls) :: cs)
    | false = naive-dpll' (restrict1 (x , notb b) cs)
```

The function `naive-dpll'` is a version of `naive-dpll` which applies to something of type `Either CNF Bool`, the output type of `restrict1`. Besides this, the go function exactly implements the logic described above: we pattern match on our input CNF until we find a variable to branch on, and then we branch on it. If the input formula is empty, then we've satisfied every clause, and so the formula is satisfiable. If there is an empty clause, then the current assignment is said to have found a **conflict**, and we have to backtrack.

Remark 7. *Note the use of the terminating pragma. It's clear to us that this algorithm terminates because restriction can only make a formula smaller. But it is not clear to Agda since the new formula is not* structurally *smaller. This hearkens back to our discussion of the difference between ordinary and induction in Remark 1.*

One role of contemporary SAT-solver research is to improve this simple algorithm. There are two heuristics which have historically been associated with DPLL.

- *Unit Propagation.* If a clause with a single literal x^a appears in ϕ, then we can immediately branch on x, and we don't have to check $\phi|_{x=1-a}$..

- *Pure Literal Elimination.* if a literal x^a appears in ϕ but x^{1-a} does not, then we can immediately branch on x and we don't to check $\phi|_{x=1-a}$.

We now present an implementation of DPLL in Agda with unit propagation, and we leave it as an exercise to add pure literal elimination to the implementation.

```
has-unit : CNF -> Maybe Lit
has-unit [] = Nothing
has-unit ((l :: []) :: cs) = Just l
has-unit (_ :: cs) = has-unit cs
```

```
{-# TERMINATING #-}
dpll : CNF -> Bool
dpll f = dpll' (left f) where mutual
  dpll' : Either CNF Bool -> Bool
  dpll' (right b) = b
  dpll' (left f) = go f

  go : CNF -> Bool
  go [] = true
  go ([] :: _) = false
  go (((x , b) :: ls) :: cs)
    with has-unit (((x , b) :: ls) :: cs)
  go (((x , b) :: ls) :: cs)
    | Just l = dpll' (restrict1 l (((x , b) :: ls) :: cs))
  go (((x , b) :: ls) :: cs)
    | Nothing
    with dpll' (restrict1 (x , b) cs)
  go (((x , b) :: ls) :: cs)
    | Nothing
    | true = true
  go (((x , b) :: ls) :: cs)
    | Nothing
    | false = dpll' (restrict1 (x , notb b) (ls :: cs))
```

And that's it, that's DPLL (it's sometimes surprising that this algorithm is still relevant today). A couple last remarks.

- One detail we are eliding quite heavily is the choice of variable to branch on, in the case there are no unit clauses or pure literals. We've use a trivial heuristic above (i.e., we branch on the first variable we see) but the choice of branching variable can be incredibly important.
- To say that many contemporary SAT solvers are based on DPLL is true, but it is a heuristic built on top of DPLL called *conflict-driven clause learning* (CDCL) that has made contemporary solvers so successful. This heuristic "learns" new clauses when there is a conflict (i.e., when there is an empty clause in the restriction of the input formula under the current assignment) in that it adds a new (hopefully small) clause to the original formula that the solver can use to guide its future exploration of the search-space of possible assignments.

See ?? for more details about both of these ideas.

5.3 CNF Encoding

We shift our focus now to the *use* of SAT solvers. First, we need to be able to encode computational problems as CNF formulas. This turns out to be tricky (and something which SMT solvers, discussed later, are meant to address) as CNF formulas are not as convenient for expressibility as they are for computation.

In the previous chapter we proved that every propositional formula is logically equivalent to a CNF formula. The issue is that the procedure we used to prove this may create *very large* CNF formulas, potentially exponential in the size of the original formula (where size is measure by the number of connectives). This leaves two questions: *What formulas* can *be represented by small CNF formulas? And what do we do when a formula* cannot *be represented by a small CNF formula?*

We will primarily focus on the first question. The second question is solved in part by Gregory Tseitin.

Theorem 5 (Tseitin)

There is an equisatisfying transformation T : $\mathsf{Prop} \to \mathsf{Prop}$ such that, for any propositional formula ϕ, the formula $T(\phi)$ is a CNF formula whose size is polynomial in the size of ϕ.

Since we're interested in formulas up to satisfiability, it suffices to consider a CNF formula which is equisatisfiable. For this transformation (typically called the *Tseitin transformation*) it is also possible to derive a satisfying assignment for the original formula from a satisfying assignment of the transformed formula.

Remark 8. *That the transformed formula is a "small" CNF formula does not imply it is easy to solve. And there may be "better" CNF encodings depending on domain specific considerations. But it* does *mean there won't be a bottleneck at encoding.*

Having a general transformation like the one above can be very useful, but there are fundamental propositional statements which we can encode directly. We'll consider just the statements that will be necessary for the example in the subsequent section.

- $\mathsf{one}_{\geq}(l_1, \dots, l_k)$ stands for "at least one of the literals $l_1, \dots, l_k$ is true". This is an easy one to represent as CNF, its disjunction:

$$\mathsf{one}_{\geq}(l_1, \dots, l_k) \triangleq l_1 \lor \dots l_k$$

- $\mathsf{one}_{\leq}(l_1, \dots, l_k)$ stands for "at most one of the literals $l_1, \dots, l_k$ is true". This one is not as easy, but we can rephrase the above statement as: "it is not possible for two distinct literals l_i and l_j to be true". Using De Morgan's law, we can write this as a CNF formula:

$$\mathsf{one}_{\leq}(l_1, \dots, l_k) \triangleq \bigwedge_{1 \leq i < j \leq k} \neg(l_i \wedge l_j) \equiv \bigwedge_{1 \leq i < j \leq k} \neg l_i \vee \neg l_j$$

- $\mathsf{one}(l_1, \dots, l_k)$ stands for "exactly one of the literals $l_1, \dots, l_k$ is true". With the above two CNF encodings, this one is easy:

$$\mathsf{one}(l_1, \dots, l_k) \triangleq \mathsf{one}_{\leq}(l_1, \dots, l_k) \wedge \mathsf{one}_{\geq}(l_1, \dots, l_k)$$

There are a number of CNF encodings used in practice, and part of the "art" of using SAT solvers is writing good encodings for the given application.

We make one final observation for this section: the formula $\mathsf{one}_{\geq}(l_1, \dots, l_k)$ introduces a quadratic number of clauses, which may be intractable for very large k. Using a trick similar to the one used by Tseitin in the proof of Theorem 5, we can get that down to a linear number of clauses.

Fact 9

The CNF formula $\mathsf{one}_{\geq}(l_1, \dots, l_k)$ *is equisatisfiable with*

$$\begin{aligned} &\mathsf{one}_{\geq}(l_1, z_1) \\ \wedge\ &\mathsf{one}_{\geq}(\neg z_1, l_2, z_2) \\ \wedge\ &\dots \\ \wedge\ &\mathsf{one}_{\geq}(\neg z_{k-2}, l_{k-1}, z_{k-1}) \\ \wedge\ &\mathsf{one}_{\geq}(\neg z_{k-1}, l_k) \end{aligned}$$

were $z_1, \dots, z_{k-1}$ *are new variables.*

It's not important that you immediately grok this fact, this is just to say that the quadractic size blow-up is not inherent (this trick can also be used to give a linear size (equisatisfiable) encoding of $l_1 \oplus \cdots \oplus l_k$, which would require exponentially many clauses otherwise).

5.4 Example: Sudoku

To demonstrate the use of SAT solvers, we will use a Python interface called PySAT. This is so we can side-step concerns about how to invoke different solvers, read their outputs, format their inputs, etc. With PySAT, we have just a handful of methods we need to get started.

- `pysat.solvers.Solver` is the class for working with SAT solvers. `Solver()` instantiates a new solver.
- Given a solver `s`, the method `s.add_clause(c)` adds a clause to the CNF formula that will be solved by `s`. Clauses are represented as lists of nonzero integers, where positive and negative literals are represented as positive and negative numbers, respectively.
- The method `s.append_formula(f)` adds a collection of clauses to the underlying formula. The formula `f` is given as a list of clauses.
- The method `s.solve()` determines the satisfiability of the formula represented by the clauses added to `s`. It returns `True` if it is satisfiable and `False` otherwise.
- The method `s.get_model()` can be called after `s.solve()` to get a satisfying assignment (represented as a list of nonnegative integers) if the list is satisfiable. It returns `None` otherwise.

The following is a simple example based on one in the PySAT documentation:

```python
from pysat.solvers import Solver

s = Solver()
s.add_clause([-1, 2])
s.add_clause([-2, 3])
print(s.solve())
print(s.get_model())

# prints:
# True
# [-1, -2, -3]
```

The formula given to the solver in this example may be written as

$$(\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3)$$

and the satisfying assignment returned by the solver is $\{\neg x_1, \neg x_2, \neg x_3\}$.

We now consider a classic exercise in the application of SAT solvers: building a Sudoku solver. The assiduous reader may wish to attempt this before reading what follows.

(Pause for effect...)

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Figure 5.1: Example of a Sudoku puzzle

If you haven't had a chance to solve a Sudoku puzzle, it is a puzzle in which the goal is to fill a 9×9 grid with digits from 1 to 9, subject to the following restrictions:

- Each row must contain exactly one of the digits from 1 to 9.
- Each column must contain exactly one of the digits from 1 to 9.
- Each 3×3 box outlined by dark lines (see Figure 5.1) must contain exactly one of the digits from 1 to 9.

What makes the puzzle interesting is that some numbers have already been filled in, and your solution must be consistent with them.[2]

In order to represent this puzzle as a CNF formula we first have to determine the underlying variables. Since propositional formulas can only be true or false, we cannot have variables which represent the *value* at every position, be we *can* have a variable for whether or not a position has a certain value:

$$x_{i,j,k} \text{ stands for “position } (i,j) \text{ has the value } k\text{”}$$

But we must enforce that every position has exactly one value:

$$\mathsf{valid} \triangleq \bigwedge_{i=1}^{9} \bigwedge_{j=1}^{9} \mathsf{one}(x_{i,j,1}, \ldots, x_{i,j,9})$$

We then need to express that our rows, columns, and boxes have exactly one digit from 1 to 9. I will only present the formulas for rows and columns, the

[2]It is also generally accepted that a Sudoku puzzle has exactly one solution, but won't make use of this assumption.

formula for boxes (denoted by boxes below) is more complicated to write down, though conceptually very similar.

$$\mathsf{rows} \triangleq \bigwedge_{i=1}^{9} \bigwedge_{k=1}^{9} \mathsf{one}(x_{i,1,k}, \dots, x_{i,9,k})$$
$$\mathsf{columns} \triangleq \bigwedge_{j=1}^{9} \bigwedge_{k=1}^{9} \mathsf{one}(x_{1,j,k}, \dots, x_{9,j,k})$$

We can combine the formulas discussed so far into a single formula which expresses the conditions of a Sudoku puzzle:

$$\mathsf{sudoku} \triangleq \mathsf{valid} \wedge \mathsf{rows} \wedge \mathsf{columns} \wedge \mathsf{boxes}$$

We can finally include clauses which express that some positions have already filled in. Let B be a partial function from positions to digits from 1 to 9 where $B(i,j) = k$ if k is filled in at the start of the puzzle at position (i,j). For example $B(3,2) = 9$ for the puzzle in Figure 5.1. We define the CNF formula which is a conjunction of unit clauses, one for each position already filled:

$$\mathsf{board}(B) \triangleq \bigwedge_{(i,j,k) \text{ s.t } B(i,j)=k} x_{i,j,k}$$

If we add this to our sudoku formula, it will kick off some unit-propagations, which will simplify the formula and enforce the values at the given positions.

Fact 10

If sudoku $\wedge$ board(B) *has a satisfying assignment, then it is possible to derive a solution to the Sudoku puzzle represented by* B.

See the course repository for a full solution. Besides the code for pretty-printing and converting a satisfying assignment into a solution to the given Sudoku puzzle, its just 20 or so lines of code. And we didn't do anything algorithmic, that was done by the SAT solver. We just had to express what (not how) we wanted the SAT solver to solve.

5.5 Further Reading

Fill in this section.

6 Propositional Proofs

7 Theories and Models

8 The Lambda Calculus