Implementing a Propositional Logic Theorem Prover in Haskell

Submitted 21st April 2010
in partial fulfillment of the
conditions of the award of the degree
BSc (Jt Hons)

Laurence Edward Day
Student ID: 4061957
School of Mathematical Sciences
University of Nottingham

I hereby declare that this dissertation is all my own work
except as indicated in the text:

Signature: _____

Abstract

We set out to create a program which checks the validity of a theorem specified in propositional logic, written in Haskell. We begin by introducing the syntax and semantics of propositional logic and discussing the concepts of logical inference, soundness and completeness. We discuss the resolution inference rule, the conjunctive normal form for sentences of propositional logic and demonstrate how using resolution to derive contradictions suffices to prove a theorems' correctness.

We then analyse both the algorithm used to implement the resolution rule and the data types which allow Haskell to work with the logic, before describing each of the modules used in a shell-only program implementation and how they interact. We introduce the Glade toolkit and build an interface around the shell implementation and add some features to produce the final version of the theorem prover.

We conclude by discussing the testing undertaken throughout development (performed with both the QuickCheck and Haskell Program Coverage toolkits) and giving two proofs that the soundness and completeness properties hold in propositional logic with respect to resolution refutation.

Contents

Chapter 1

Coding in Context

1.1 Project Motivation

Academia has long recognised that there are dozens of problems within several notable fields which cannot be solved in anything close to a reasonable amount of time by humans, for reasons ranging from having to write out massive amounts of information to not being able to follow the reasoning involved. To this end, several developments have been made in the field of Automated Theorem Proving (ATP) to aid us by performing often unfathomably large-scale computations in our stead.

For example, Herbert Robbins suggested a new set of axioms as a valid basis for boolean algebras in 1933 - whether or not they indeed were, lay as an open problem for over sixty years. However, in 1996 a fully automated proof was found by the EQP program[1] - the first major breakthrough of its kind in pure mathematics. This, alongside the proof by Appel and Haken of the four-colour theorem[2], and Hales' work on the Kepler conjecture[3], go some length towards justifying ATP's usage within mathematics alone.

Given this, we look at the 'simple' matter of proving entailment of a sentence from a set of sentences - in other words, proving that the implicit truth of a set of hypotheses somehow imply a conjecture (or set of conjectures). Naturally, such sentences need to be represented in a form with which a program can meaningfully interact. For this, we employ a logic - specifically propositional logic.

One could argue that first-order logic should be used to represent the sentences (since it can indicate facts, relations and objects as opposed to just facts[4]), however there is an element of undecidability when dealing with first-order logic (discussed in §2.2.5). As a result, we employ propositional logic with the knowledge that we have made a trade-off between representational power and assurance that our computations will terminate.

This isn't to say that programs which employ propositional logic are limited in their capabilities. As a slightly unconventional example, it has been shown that propositional satisfiability (SAT) solvers are of some use when discovering solutions for messages creating collisions with hash functions in cryptanalytic attacks[5]. Also, it has long been known that digital circuit representation is isomorphic to boolean algebra[6] (equivalently propositional logic), allowing for simple verification of the former. Such a simple logic is not to be dismissed out of hand.

If we need further convincing about the usefulness of theorem provers, we need only look at software and hardware verification. With successful applications ranging from compiler verification[7] to generating theorems for use in air traffic management[8] and (costly) lessons to learn from such as the 1994 Pentium FDIV bug[9] (although this did result in some interesting keychains), formally proving system correctness is nowadays essential. This is evidenced somewhat by the sheer number of theorem provers present in both industry and academia.

Looking at the choice of implementation language (paradigm), Haskell (functional) lends itself easily to dealing with problems in the vein of theorem proving for several reasons. Foremost amongst these is the fact that Haskell code tends to be more concise than in languages such as Java or C++ (a feature of the functional paradigm as a whole) - resulting in easier modelling and reasoning, as well as making a final program simpler to prove correct if needed.

Further, Haskell as an (almost) pure language has an advantage in the seperation of pure from effectful code - we can be certain that our functions perform precisely what we demand, without risk of interference from obscure bugs[10]. In this way, we can translate from pseudocode to meaningful functions with minimal effort, worrying about program interaction at a later stage.

With these justifications in mind, we can develop specific objectives to achieve, as discussed in the next section.

1.2 Program Description

Any project, be it software-based or otherwise, requires set goals to measure how 'much' has been implemented. Whilst obvious that we intend to develop a functioning theorem prover, a better measure of success lies in giving a 'feature checklist' of sorts and comparing the end result to this. What follows is just that; a list of features which we would consider 'essential' in the final program.

- The user must be able to provide input with minimal effort - the syntax must be easy to follow.
- The program must be able to convert any input into a form it can operate on with no further user correspondence.
- The program must be robust (must not terminate upon receiving incorrect input - non-alphanumeric characters, etc).
- The program output must be a boolean value, reflecting the validity of the theorem given as input.
- If the user provides no hypotheses, the program must check for contradictions in the conjecture/s provided.
- If the user provides no conjecture, the program must return a positive answer, as all hypotheses are assumed to hold.
- If the user provides neither hypotheses nor conjecture, the program must inform the user of as much.
- The program must pass a number of QuickCheck proposition tests as a measure of program correctness.

Building on the above, there are several peripheral features that - whilst not having a direct impact on the operation of the program - would improve ease of use and faith in the program output. Once the above features are all present within a core implementation, they will be extended as follows:

- The program must output a log of steps (prooflog) taken in an attempt to prove a theorem (regardless of final result), prettyprinted in a way that lets the user follow the reasoning used.
- A user interface must be present (rather than running entirely within shell) to allow easier input and readability of output.
- The user must be able to import a text file containing a proof specification (rather than typing everything in at once), increasing program usability for middle-to-large scale theorems.
- The program must be able to write a prooflog to a text file for later reference - preferably to the same file that imported the specification.
- The Haskell Program Coverage toolkit will be used, ideally demonstrating 80%+ code coverage.

Finally, two proofs are given demonstrating the soundness and completeness of the inference rule - resolution refutation - used by the program.

1.3 Related Work

Given the range of applications, there is significant interest in ATP worldwide. Several competitions run at academic conferences annually to stimulate development of new and efficient ATP methods, the premier of which is the CADE ATP System Competition (CASC), unofficially considered the 'world cup' of such proceedings[11]. Competitions are run within specific subfields of ATP also - the most appropriate to mention here being the SAT Competition.

As would be expected, such competitions produce a number of long-running implementations. The most prominent of these is Vampire[12], having won twenty division titles at CASC at the time of writing. Other such programs include Gandalf and Waldmeister, also CASC division winners. Competitions aside, papers discussing the merits and differences of various programs are commonplace[13].

Unfortunately, the concept of writing theorem provers in Haskell is not groundbreaking. The program Paradox, developed by Claessen and Sörensson at Chalmers, is written in Haskell and a CASC division winner from 2003 through to 2006. The relative simplicity of propositional logic lends itself as an educational aid as well - the framework of a tautology checker is given as an extended example of type and class declaration in Huttons' book[14].

At the undergraduate dissertation stage too, theorem provers in Haskell are explored territory[15]. This work aims to improve upon the efforts made at this level by extending the capabilities of previously developed programs; by providing an easy to understand interface, the ability to import and save both theorem specifications and prooflogs, and rigorous testing elements alongside a handful of suggestions for further work.

Chapter 2

Background Theory

2.1 Propositional Logic 101

Before anything is said about program implementation, we look at the syntax and semantics of propositional logic as a knowledge representation language. If the reader is comfortable with these, skip to §2.2.

2.1.1 Syntax

The 'alphabet' of propositional logic consists of both logical and nonlogical symbols. Sentences are constructed by combining any number of these symbols together (provided they obey the logic semantics).

Non-logical Symbols are predicates of arity zero (we will refer to them as constants), used to represent truths about the world, such as:

P - the four angles of a square are all equal, summing to 360°.

Q - Dublin is the capital of the Republic of Ireland as of 2009.

R - Barack H. Obama is the 44^{th} President of the United States.

Note functions are not considered in propositional logic. This is because they accept terms drawn from the logics' domain as their parameters and return terms within the same. Since the domain of propositional logic is the set of booleans - $\{\text{True, False}\}$, we can safely do away with them.

As an extended example, the constants defined above retain their meaning throughout this subsection.

Logical Connectives are the 'punctuation' of sentences written in a logic, used to combine atomic (single) propositions into complex ones. We introduce the following:

- **Conjunction ($\wedge$)** - binary connective evaluating to True if and only if both of its arguments evaluate to True. In particular, P $\wedge$ Q reads "the four angles of a square ... *and* Dublin is the capital ...".
- **Disjunction ($\vee$)** - binary connective evaluating to True if and only if either of its arguments evaluate to True (and possibly both). In particular, P $\vee$ R reads "the four angles of a square ... *or* Barack H. Obama is the ...".
- **Negation ($\neg$)** - unary connective evaluating to True if its argument evaluates to False and vice-versa. In particular, $\neg$Q reads "Dublin is *not* the capital ...".
- **Implication ($\Rightarrow$)** - binary connective evaluating to True unless its first argument (the antedecent) evaluates to True and its second (the consequent) to False. In particular, Q $\Rightarrow$ R reads "*if* Dublin is the capital ... *then* Barack H. Obama is the ...".
- **Equivalence ($\Leftrightarrow$)** - binary connective evaluating to True if and only if implication holds in both directions. In particular, P $\Leftrightarrow$ Q reads "the four angles of a square ... *if and only if* Dublin is the capital ...".

Having defined our alphabet of symbols, we conclude this subsection by introducing some axioms of logical congruence.

Symmetry:
$$A \wedge B \equiv B \wedge A$$
$$A \vee B \equiv B \vee A$$
$$A \Leftrightarrow B \equiv B \Leftrightarrow A$$

Double Negation:
$$A \equiv \neg\neg A$$

Implication:
$$A \Rightarrow B \equiv \neg A \vee B$$

Equivalence:
$$A \Leftrightarrow B \equiv (A \Rightarrow B \wedge B \Rightarrow A)$$

Distribution:
$$A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$$
$$(A \wedge B) \vee C \equiv (A \vee C) \wedge (B \vee C)$$

De Morgan:
$$\neg(A \wedge B) \equiv (\neg A \vee \neg B)$$
$$\neg(A \vee B) \equiv (\neg A \wedge \neg B)$$

2.1.2 Semantics

We now look at the meanings assigned to sentences of propositional logic. The notation and ideas in this section are adapted from the text by Brachman and Levesque[16] and tailored to propositional logic.

As they stand, we have no way to interpret sentences written in propositional logic. Placing ourselves in the role of a theorem prover; if a user types in the sentence (A $\vee$ B) $\Rightarrow$ C, we have no way of knowing what meaning the user gives the three constants A, B and C, nor can we tell what this sentence evaluates to.

More formally, the truth value of a sentence is determined by how we interpret these constants - we need an *interpretation mapping* between the set of defined propositional constants Σ and the booleans, which we denote θ: $\Sigma \mapsto \{\text{True, False}\}$. Using this mapping, we can derive the truth value of complex propositions through the application of some fairly trivial operator semantics. In particular we can define a mapping between sentences and truth values, denoted $\theta\prime$, by induction:

Examples: $\theta\prime(\text{A}) \mapsto \text{True} \Leftrightarrow \theta(\text{A}) \mapsto \text{True}$.
$\theta\prime(\text{A} \wedge \text{B}) \mapsto \text{True} \Leftrightarrow \theta\prime(\text{A}) \mapsto \text{True} \wedge \theta\prime(\text{B}) \mapsto \text{True}$.

The behaviour of $\theta\prime$ for the other logical connectives is as expected.

However - so far as the logic is concerned, the choice of assignments made by θ when applied to Σ is irrelevant: we don't know (or care) which interpretations are more appropriate than others. It is worth noting that if we define n to be the cardinality of Σ, $|\Sigma|$ - there are 2^n possible permutations of interpretation mappings.

Introducing a final piece of notation: the binary operator $\models$ ("models") is used as follows: we read the expression $\theta \models \alpha$ as "$\theta\prime(\alpha)$ evaluates to True for this particular interpretation mapping θ". With this, we can introduce our definitions of *entailment* and what it means for a sentence to be *unsatisfiable*.

Assuming (possibly complex) propositions α and β we claim the following, abusing notation as we go -

$$\begin{aligned}
\alpha \models \beta &\Leftrightarrow \forall\theta, (\theta \models \alpha \Rightarrow \theta \models \beta)\ [\dagger] \\
&\Leftrightarrow \forall\theta, (\theta \models \alpha \Rightarrow \beta) \\
&\Leftrightarrow \nexists\theta, (\theta \models \neg(\alpha \Rightarrow \beta)) \\
&\Leftrightarrow \nexists\theta, (\theta \models \alpha \wedge \neg\beta) \\
&\Leftrightarrow \forall\theta, (\theta\prime(\alpha \wedge \neg\beta) \mapsto \text{False})\ [\star]
\end{aligned}$$

This claim gives us both definitions at once. [†] tells us that if *any* interpretation mapping θ giving rise to $\theta \models \alpha$ also means that $\theta \models \beta$, we can conclude that β is a logical consequence of α - or that α *entails* β. [$\star$] adds to this by saying that α entails β if and only if for *all* interpretations θ, $\theta\prime$ cannot evaluate the sentence $\alpha \wedge \neg\beta$ to True - this is shorthand for saying that $\alpha \wedge \neg\beta$ is *unsatisfiable*. This makes sense once one accepts that a logical argument that $\alpha \Rightarrow \beta$ is invalid if α is true and β is not.

Finally, it is immediate that if our α above is a *set* of sentences $\{\alpha_1, \ldots, \alpha_k\}$ for some finite k, we can reformulate the set as a single sentence by chaining each α_i to the rest of the set by conjunction. In this way, we can refine our definition of entailment for a *knowledge base* (KB) α as:

$$\{\alpha_1, \ldots, \alpha_k\} \models \beta \Leftrightarrow \forall\theta, (\theta \models (\alpha_1 \wedge \cdots \wedge \alpha_k) \Rightarrow \beta)$$

Our logical notation and definitions now in place, we look at the process of deriving new information from old, or inference.

2.2 Logical Inference

The crux of reasoning, automated or human, is coming to conclusions based on confirmed knowledge. For example, if you wake up in the morning and see that it's raining heavily, you (hopefully) don't cheerfully walk outside without an umbrella or some other form of cover. However - computers, not being able to reason in the same way we do, must follow strictly defined rules to obtain any new information.

The simplest of these *inference rules*, one that we subconsciously use all the time, is the rule of *modus ponens*. This rule states that if we know that if $\alpha \Rightarrow \beta$ and α are both true, we can infer β (this is simple to check with a truth table). In particular, to borrow from that ancient syllogism - if all men are mortal and Socrates is a man, we know that Socrates is mortal.

There are several rules relating to each logical connective discussed in §2.1.1 - and we now focus on their application.

2.2.1 Soundness and Completeness

First, we must note that inference rules work in conjunction with a formal language (in this case propositional logic) to form what is referred to as a *logical system*. These must be both sound and complete with respect to their inference rules for their users to have any

faith in their conclusions.

Soundness, also known as **truth-preserving**, is the property held by a logical system if its inference rules only derive new information that is valid under the formal languages' semantics. More precisely, if we can derive a sentence β through (finite) application of inference rules to a knowledge base α (which we will abbreviate $\alpha \vdash \beta$, pronounced "α *syntactically entails* β"), it must be the case that $\alpha \models \beta$.

[Note we have introduced some duality in our usage of the word 'entails' - from now on we read $\alpha \models \beta$ as "α *semantically entails* β".]

Completeness, the converse of soundness, completeness is the property held if *any* sentence which is semantically entailed by a knowledge base α can be derived by (finite) application of the inference rules of a logical system - formally represented by $\alpha \models \beta \Rightarrow \alpha \vdash \beta$.

We can conclude that in a logical system possessing both properties, $\alpha \vdash \beta \Leftrightarrow \alpha \models \beta$ and our two definitions of entailment are equivalent. We take advantage of this when constructing proofs in the program to follow.

We must now choose which inference rules to associate with propositional logic, bearing in mind that such a set of rules (and the resulting logical system) must be sound, complete, easy to reason with and implementable as a computer program. Somewhat surprisingly, Robinson posited the existence of a single inference rule which could fill this role - *resolution*[17].

2.2.2 Mechanics of Resolution

The premise of resolution is deceptively simple - when dealing with two disjunctions containing a complementary pair, a disjunction of all constants except for the pair in question can be inferred. More formally - and by way of example - assuming disjunctions of two constants:

$$\frac{A \vee B \qquad\qquad \neg B \vee C}{A \vee C}$$

This principle easily generalises to disjunctions of an arbitrary number (larger than zero) of constants -

$$\frac{A_1 \vee \ldots \vee A_i \vee \ldots \vee A_k \qquad B_1 \vee \ldots \vee B_j \vee \ldots B_p}{A_1 \vee \ldots \vee A_{i-1} \vee A_{i+1} \vee \ldots \vee A_k \vee B_1 \vee \ldots \vee B_{j-1} \vee B_{j+1} \vee \ldots \vee B_p}$$

- where $A_i = \neg B_j$ and vice-versa. In the case of k or p = 1, note the solitary constant D in the disjunction can be rewritten as D $\vee \perp$, where $\perp$ ('bottom') represents the proposition which is *always* false (the classic example is 1 = 2), and logical equivalence is preserved.

For those concerned, worry not - the dangling question of whether or not this rule *does* present a sound and complete logical system when paired with propositional logic is addressed in §4.3.

Of course, restrictions must be imposed on the structure of the sentences we are dealing with in order to apply resolution in a meaningful way - the rule says nothing about any logical connectives other than disjunction and negation. These restrictions are enforced by rewriting our sentences in Conjunctive Normal Form (CNF).

2.2.3 Conjunctive Normal Form

The author believes that it was Hilbert and Ackermann to first show that any sentence written in propositional logic can be transformed into a logically equivalent conjunction of disjunctions, with each disjunct either a constant or the negation of the same[18]. The algorithm they follow to do this is as follows:

1. *Implication Elimination* - transform any proposition of the form A $\Rightarrow$ B to $\neg A \vee B$ and any proposition of the form A $\Leftrightarrow$ B to $(\neg A \vee B) \wedge (\neg B \vee A)$.
2. *De Morgan's Laws* - push negations of parenthesised sentences inwards by transforming any proposition of the form $\neg$(A $\vee$ B) to $\neg A \wedge \neg B$ and any proposition of the form $\neg$(A $\wedge$ B) to $\neg A \vee \neg B$.
3. *Double Negation Elimination* - transform any proposition of the form $\neg\neg$A to A.
4. *Algebraic Distribution* - distribute conjunctions over disjunctions by transforming any proposition of the form A $\vee$ (B $\wedge$ C) to (A $\vee$ B) $\wedge$ (A $\vee$ C) and any proposition of the form (A $\wedge$ B) $\vee$ C to (A $\vee$ C) $\wedge$ (B $\vee$ C).
5. *Repetition Elimination* - transform any proposition of the form (A $\vee \cdots \vee A \vee \cdots \vee$ B) to (A $\vee \cdots \vee$ B).

The result is a sentence consisting of disjunctions 'punctuated' by conjunctions, where each disjunct is possibly negated. In particular:

$$\text{A} \Leftrightarrow \neg\text{B} \equiv (\neg\text{A} \vee \neg\text{B}) \wedge (\text{B} \vee \text{A})$$

Our sentences now in CNF, we introduce more definitions and notation. Treating conjunctions as seperators, our disjunctions are now referred to as *clauses*, turning a CNF sentence into a *clausal list*. In particular:

$$(\neg A \vee \neg B) \wedge (B \vee A) \mapsto [[\neg A, \neg B], [B, A]].$$

Our sentences are now suitable for application of the resolution inference rule. Further, this style of representing sentences is inherently suited to Haskell using lists of lists as our data structure, provided we interpret disjunctions as seperating items *within* a clause.

2.2.4 How Does Resolution (Refutation) 'Prove' Anything?

At this point we must admit that we are telling a white lie by saying that the program uses 'resolution'. Rather it uses '*resolution refutation*', by using the resolution rule in the process referred to in §2.1.2. In other words, we apply the resolution rule as described, but apply it to the conjunction of the hypotheses and the negated conjecture/s, ultimately looking for a derivation of $\bot$. We interchange the terms resolution refutation and resolution from now on.

We look at how we can use resolution to prove entailment of a sentence β from a knowledge base α. As discussed in §2.2.1, the two definitions of entailment are equivalent under a sound and complete logical system (which we assume for now, but prove later). Using this, we adopt the *proof by contradiction* method:

Assuming (for contradiction) that $\alpha \not\models \beta$, we attempt to prove $\forall\theta$, $(\theta \models \neg(A \Rightarrow B))$, or equivalently $\forall\theta$, $(\theta \models A \wedge \neg B)$. However, the interpretation mapping θ is unnecessary, as the resolution rule explicitly states that we are can only infer sentences which exclude *complements* of a constant, and so the value of any given constant under an interpretation is irrelevant.

We apply the resolution rule to the clausal list KB consisting of the CNF form of the sentence $\alpha \wedge \neg B$, adding any sentences inferred by the resolution rule (called *resolvents*) to the KB and repeating. This process ends in one of two ways:

1. **Exhaustion**: no new resolvents can be inferred from the KB which are not the empty clause [] (justified below). From this, we can assume that no contradictions exist in the clausal list of the original KB, and our assumption that $\alpha \not\models \beta$ holds.
2. **Contradiction**: the resolution rule is applied to a complementary pair; in particular [K] and [$\neg$K] for some K. Rewriting these clauses in the way described in §2.2.2 gives us:

$$\frac{\bot \vee K \qquad\qquad \neg K \vee \bot}{\bot \vee \bot}$$

This inference of $\bot$ indicates a contradiction (otherwise we suggest that falsity is part of our KB, and all bets are off). As a result, our assumption that $\alpha \not\models \beta$ must be incorrect, hence $\alpha \models \beta$ (which is what we wanted to show), and the proof is complete.

2.2.5 Why Didn't We Use First-Order Logic?

The decision to choose propositional logic over first-order logic can be explained with a single example (the idea for which is from the text by Brachman and Levesque[16]) - in brief, quantifiers and substitution introduce non-termination. Assuming that the reader is comfortable with the syntax of first-order logic, consider the following rule -

$$\forall\ a, b\ .\ AncestorOf(childOf(a), b) \Rightarrow AncestorOf(a, b)$$

In other words, if a person (say A) is an ancestor of another person B, then so is their parent, C. As a (silly) instance of how using this rule in practice could throw up problems, consider what would happen if we attempted to prove that someone is their own ancestor using resolution refutation (given a KB which is already converted into CNF) -

[¬AncestorOf(childOf(a), b), AncestorOf(a, b)], [¬AncestorOf(johnDoe, johnDoe)]

- where johnDoe is some arbitrary term for which the AncesterOf function is defined. Applying the resolution rule in the only place we can (substituting both variables a and b with the term johnDoe) we end up with the following KB -

[¬AncestorOf(childOf(a), b), AncestorOf(a, b)], [¬AncestorOf(johnDoe, johnDoe)],
[¬AncestorOf(childOf(johnDoe), johnDoe)]

On the next pass, we would resolve the just-generated resolvent with the first clause (as no other resolvents are possible), substituting the variables a with childOf(johnDoe) and b with johnDoe. It should now be clear that we never reach a result - rather, we generate an infinite KB in our attempt to find one. This intractability makes for unpleasant hackery in the program code to counteract it - such as returning an "Unknown" result after a number of steps exponential in the input - which whilst initially considered for the program to follow, was ultimately dropped.

If interested, a particularly informal-yet-informative explanation of first-order intractability in the context of Turing machines has been written by Bezhanishvili and Moss[19].

Having discussed the form of a proof using resolution refutation in propositional logic, we now look at the algorithm used in our Haskell program to implement it.

Chapter 3

Haskell Implementation

3.1 Resolution Algorithm

In this section we describe the algorithm implementing the resolution inference rule of 2.2.2 in comparatively high-level terms - we avoid describing the underlying mechanics of Haskell in all but the most necessary of cases.

We assume that the conjunction of our set of hypotheses (the KB) α and the negation of our (possibly complex) conjecture β has been converted into CNF, and also that this CNF sentence is represented as a clausal list, with conjunctions seperating items of the outermost list (the clauses) and disjunctions the items of the innermost lists (the constants).

As a running example, we prove the tautology $((A \vee B) \wedge (A \Rightarrow C) \wedge (B \Rightarrow C)) \Rightarrow C$ by deriving a contradiction from the clausal list [[A, B], [¬A, C], [¬B, C], [¬C]].

1. The data types used are to record progress are initialised. We keep track of:
    - The clause currently being used to generate resolvents (the *resolution clause*) - initialised to [] (the *empty list*).
    - A list of the resolution clauses which have already been considered - initialised to [].
    - The *remainder* of the clausal list - initialised to the full clausal list.
    - A boolean value indicating whether or not a contradiction exists in the (original) KB - initialised to False.
2. The first clause from the remainder is set as the resolution clause - in this case [A, B]. We add the resolution clause to the list of already-considered clauses and delete it from the remainder, leaving [[¬A, C], [¬B, C], [¬C]]. Alternately, if the

remainder is the empty list before the resolution clause is set, the algorithm terminates under the exhaustion condition.

3. The first constant of the resolution clause (the *resolution constant*) is taken as the constant we generate resolvents from - A at this point. This constant is temporarily deleted from the resolution clause, leaving [B]. (*Note this temporary resolution clause can be [].*)

4. The resolution constant is negated (if it is already negated, double negation elimination is applied), giving ¬A, and the remainder is searched for all clauses containing this negation. In all clauses where it is found - here the single clause [¬A, C] - the negation is deleted and the temporary resolution clause is appended, leaving the resolvents - here the single resolvent [C, B].

5. The resolvents are filtered against the clauses in the remainder and those resolution clauses already considered - if any of the resolvents are a permutation of any of these clauses, they represent no new information and as such are excluded. Those remaining resolvents (if any) are added to the remainder - [C, B] is not a permutation of any of [¬A, C], [¬B, C], [¬C] or [A, B] and so is added.

6. At this point, if [] was one of the resolvents added to the remainder, we have derived our contradiction. If this is the case, the contradiction boolean is set to True and the algorithm terminates under the contradiction condition. Otherwise, there are two scenarios:

   - New resolvents were inferred - the resolution constant is appended to the end of the resolution clause, here giving us [B, A], and the algorithm goes back to step 3. *However*, if all constants of the resolution clause have been considered, the algorithm moves to the scenario below.
   - No new resolvents were inferred - the algorithm goes back to step 2.

Continuing this algorithm, the example given runs from start to finish as follows:

Proof: $((A \vee B) \wedge (A \Rightarrow C) \wedge (B \Rightarrow C)) \Rightarrow C$			
Res. Constant	**Res. Clause**	**Remainder**	**Resolvents**
A	[A, B]	[[¬A, C], [¬B, C], [¬C]]	[C, B]
B	[B, A]	[[¬A, C], [¬B, C], [¬C], [C, B]]	[C, A]
¬A	[¬A, C]	[[¬B, C], [¬C], [C, B], [C, A]]	[C]
C	[C, ¬A]	[[¬B, C], [¬C], [C, B], [C, A], [C]]	[¬A]
¬B	[¬B, C]	[[¬C], [C, B], [C, A], [C], [¬A]]	
¬C	[¬C]	[[C, B], [C, A], [C], [¬A]]	[B], [A], []

After inferring [], the algorithm terminates and the proof by contradiction is complete.

All background theory in place, we now look at the without-frills core implementation of the program.

3.2 Core Implementation

We begin by analysing the Haskell data structures and functions used to represent and access sentences as well as the progress of the resolution algorithm. We then consider the general purpose of each module of the program before looking at how these modules fit together to produce an operational (but basic) theorem prover. After this, we extend the core implementation into the final version with an interface and conclude by discussing some of the bugs which arose during development.

3.2.1 Data Structures Used

Firstly, we consider the data structure which represents propositional logic sentences and discuss the notation introduced to make it easier to both read and work with.

```
data Sentence =
  Sentence `Conj` Sentence
  | Sentence `Disj` Sentence
  | Sentence `Impl` Sentence
  | Sentence `Equi` Sentence
  | Neg Sentence
  | Lit String
  | Brack Sentence
  deriving (Eq, Show)
```

This recursive algebraic data type is near enough a direct representation of how we would write a sentence in propositional logic by hand (with the exceptions of Lit String and Brack Sentence, used to assign names to constants and draw attention to explicit parentheses respectively). Note that for readability we also introduce some operator synonyms, namely:

```
(/\), (\/), (==>), (<=>) :: Sentence -> Sentence -> Sentence
(/\)  = Conj
(\/)  = Disj
(==>) = Impl
(<=>) = Equi
```

This allows us to represent (A $\wedge$ B) $\Rightarrow$ C as Brack (Lit "A" /\ Lit "B") ==> Lit "C", for example. We also introduce a fixity hierarchy (based on precedence) on the binary connectives to eliminate excess parentheses - namely (from strongest to weakest binding) $\neg, \vee, \Leftrightarrow, \Rightarrow, \wedge$. Finally, we force left-associativity of conjunction and disjunction, and right-associativity of implication. (eg. A $\vee$ *B* $\vee$ *C* is shorthand for (A $\vee$ B) $\vee$ C).

Focusing on the data structure used to track the progress of the resolution algorithm, we employ a *record* allowing us to both access and change the contents of its fields using its accessor helper functions. The book *Real World Haskell*[20] by O'Sullivan, Stewart and Goerzen has a particularly easy-to-understand introduction to them.

```
data Resolution = Resolution
  { resolclause :: ResolClause
  , pastclauses :: CNFSentence
  , remainder   :: CNFSentence
  , contra      :: Bool
  }
```

Note each field specified in the record represents one of the four data structures we keep track of in the algorithm. A few type synonyms are in place for readability - CNFSentence is [[Sentence]] and ResolClause is [Sentence].

One of the major negative points of Haskell is the difficulty it has when dealing with mutable state. This leads to a paradox of sorts - how can we implement an algorithm which frequently changes a number of data types when the restrictions of the language prevent us from doing so? The answer lies with *monads*, specifically the *State* monad. Whilst not focusing directly on the State monad (the underlying principles between all monads are similar), Peyton-Jones provides a far more lucid explanation of monad mechanics[21] whilst explaining monadic I/O than can be hoped to be given here.

Below is the Haskell library code instantiating State as a member of the Monad typeclass and its data type declaration.

```
instance Monad (State s) where
                    return a = State $ \s -> (a, s)
                    (State x) >>= f = State $ \s -> let (v, s') = x s in runState (f v) s'
newtype State s a = State { runState :: (s -> (a, s)) }
```

In a sentence, the data type *State s a* mutates the parameter of type *s*, and returns a value of type *a*. We set our parameter *s* to be the Resolution record data type defined above, and

since we are not concerned with a return type, set the parameter *a* to the empty type (). The result is a data type which can pass around as a function parameter capable of mutating the fields of its record, and our 'paradox' is solved.

We introduce one final type synonym, namely ResolveM in place of State Resolution. The result is a data type ResolveM () (we shall call this the *resolution monad* for simplicity, it's not a new monad by any means) which we declare as the data type of the function initiating the resolution algorithm, namely *resolve*.

Data types introduced, we look at the role of each of the modules used in the program.

3.2.2 Module Descriptions

For each module we discuss both their general purpose and a few of their most important functions.

Main.hs - this module deals with the I/O of the program, requesting user input in the form of making choices between menu options and entering hypotheses and conjectures, returning as output a notification of if a theorem is valid or not.

For example, below is the function *outputproof* taking the set of hypotheses and negated conjecture as arguments, applying the resolution algorithm (the *propresolution* function) and returning an answer based on the value of the contradiction boolean the algorithm tracks.

```
outputproof :: Sentence → Sentence → IO ()
outputproof hyp con = do putStrLn ("\nThe knowledge base we are working with is: \n\n"
  ++ hypshown)
  putStrLn ("\nOur negated conjecture is: \n\n" ++ conshown)
  case contra res of True → putStrLn "\nThe hypotheses DO satisfy the conjecture!\n"
                     _ → putStrLn "\nThe hypotheses DO NOT satisfy the conjecture!\n"
  putStr "Return to main? [y/n]:\n\nChoice: "
  choice ← getLine
  case choice of "Y" → main
                 "y" → main
                 "N" → putStrLn "\nQuitting...\n"
                 "n" → putStrLn "\nQuitting...\n"
                 _ → putStrLn "\nInvalid input, terminating...\n"
  where res = propresolution (hyp `Conj` con)
    hypshown = replace (replace ((replace (showSent (convert hyp))) "(" "") ")" "") " /\\ " "\n"
    conshown = (replace ((replace (showSent (convert con))) "(" "") ")" "")
```

The other functions in the module involve menu navigation and passing example tau-

tologies to the *outputproof* function.

Parse.hs - this module was produced by the Happy parser generator[22] and acts as the translator from user input to a form usable by the program. A file specifying an unambiguous grammar for the Sentence data type in Backus-Naur Form (BNF)[23] was provided (see §7.1) and the resulting module contains, for example, a function translating from strings to a list of tokens (the *lexer* function) and a function translating from a list of tokens to the Sentence data type according to the grammar (the *parser* function), amongst others.

Below is the code for the lexer and parser functions. The lexer formed part of the abovementioned grammar file, whereas the parser was automatically generated (and leaves a lot to be desired aesthetically!). The composition of these functions *parse'* is the function called wherever needed throughout the rest of the program.

```
lexer :: String → [Token]
lexer []                          = []
lexer ('\n' : cs)                 = []
lexer ('\r' : cs)                 = []
lexer (c : cs)
   | isSpace c                    = lexer cs
   | isAlpha c                    = lexLit (c : cs)
lexer ('(' : cs)                  = TokenOB : (lexer cs)
lexer (')' : cs)                  = TokenCB : (lexer cs)
lexer ('!' : cs)                  = TokenNeg : (lexer cs)
lexer ('\\' : '/' : cs)           = TokenDisj : (lexer cs)
lexer ('/' : '\\' : cs)           = TokenConj : (lexer cs)
lexer ('=' : '=' : '>' : cs)      = TokenImpl : (lexer cs)
lexer ('<' : '=' : '>' : cs)      = TokenEqui : (lexer cs)
lexer (',' : cs)                  = TokenCmma : (lexer cs)
lexer (_ : cs)                    = lexer cs
lexLit cs = case (span isAlpha cs) of
              (l, rest) → (TokenLit l) : lexer rest
```

```
parse tks = happyRunIdentity happySomeParser where
  happySomeParser =
    happyThen (happyParse action_0 tks) (λx → case x of
      {HappyAbsSyn4 z → happyReturn z;
      _other → notHappyAtAll})
```

The other functions of the module deal with the middle steps of the translation between tokens and the Sentence data type.

ConvertToCNF.hs - this module deals with the transformation of a propositional logic sentence from that given as user input into CNF. A number of functions defined here are composed to form a *convert* function, defined as:

```
convert :: Sentence → Sentence
convert = combinerep ∘ doubleneg ∘ distribute ∘ (fixpoint demorgan) ∘ impelim
```

The functions comprising the convert function each deal with a specific part of the transformation as described in §2.2.3. For example, below is the code for implication elimination and the application of De Morgan's laws.

```
impelim :: Sentence → Sentence
impelim (Conj a b)  = Conj (impelim a) (impelim b)
impelim (Disj a b)  = Disj (impelim a) (impelim b)
impelim (Impl a b)  = Disj (Neg (impelim a)) (impelim b)
impelim (Equi a b)  = Conj (Disj (impelim (Neg a)) (impelim b)) (Disj (impelim (Neg b)) (impelim a))
impelim (Neg a)     = Neg (impelim a)
impelim (Lit a)     = (Lit a)
impelim (Brack a)   = impelim a
```

```
demorgan :: Sentence → Sentence
demorgan (Conj a b) = Conj (demorgan a) (demorgan b)
demorgan (Disj a b) = Disj (demorgan a) (demorgan b)
demorgan (Neg a)    = case a of
                        (Brack d) → demorgan (Neg d)
                        (Conj b c) → Disj (Neg (demorgan b)) (Neg (demorgan c))
                        (Disj y z) → Conj (Neg (demorgan y)) (Neg (demorgan z))
                        _ → Neg (demorgan a)
demorgan (Lit a)    = (Lit a)
demorgan (Brack a)  = demorgan a
```

Other functions defined in this module convert a CNF sentence into a clausal list for the resolution algorithm, and prettyprint sentences for output.

Resolution.hs - this module implements the resolution algorithm of §3.1. To this end, a number of functions are defined to generate/filter resolvents, negate the resolution clause, check for contradictions and so on. For example, on the next page is the code for the function generating resolvents from a negated resolution constant.

```
addResolvents :: Sentence → ResolveM ()
addResolvents s = do rs ← get
  let resolve ss = if (elem s ss)
    then ((delete s ss) ++ resolclause rs)
    else ss
    resolvents = map resolve (remainder rs)
  put (rs {remainder = ((remainder rs) ++
    (filterRepeats (remainder rs)
    (filterRepeats (pastclauses rs) (leaveNew resolvents (remainder rs))))))})
  return ()
```

Note the locally defined function *resolve* returns a resolvent if the negated constant is present in the clause it is mapped to, and just the clause otherwise. These untouched clauses, representing no new information, are filtered out by the *leaveNew* function - the remaining resolvents, possibly permutations of elements of the remainder or previously-considered clauses, are further filtered by the *filterRepeats* function.

Tautologies.hs - this module serves only to store String representations of the hypotheses and conjectures for number of example tautologies called upon by the Main module. Examples of such tautologies are:

```
  -- Constructive Dilemma
cdilhyp :: String
cdilhyp = "A ==> B, C ==> D, A \\/ C"
cdilcon :: String
cdilcon = "B \\/ D"
```

The module descriptions themselves act as a rough guide to understanding the workings of the program. In the next subsection, we consider how all of the modules fit together to produce the core implementation.

3.2.3 Fitting It All Together

We go through this section by showing the possible options a user can make at each stage in a program run, accompanied by screenshots where helpful.

Immediately after having compiled the Main module and calling the *main* function, we are faced with the following:

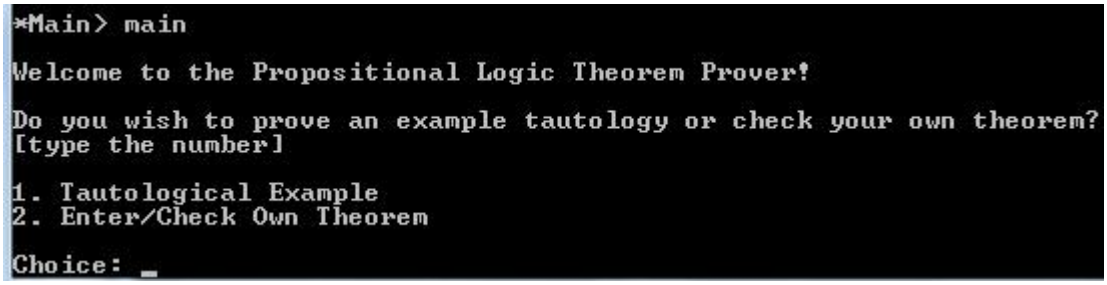

Figure 3.1: The theorem prover immediately after calling *main*.

We are presented with two options. Choosing the first will call the *listTautologies* function from the Main module, whilst the second calls the *ownTheorem* function. The former prompts the user to select a tautology to prove (the components of which are specified in the Tautologies module), whilst the latter requests custom hypotheses and conjecture/s to be entered seperately.

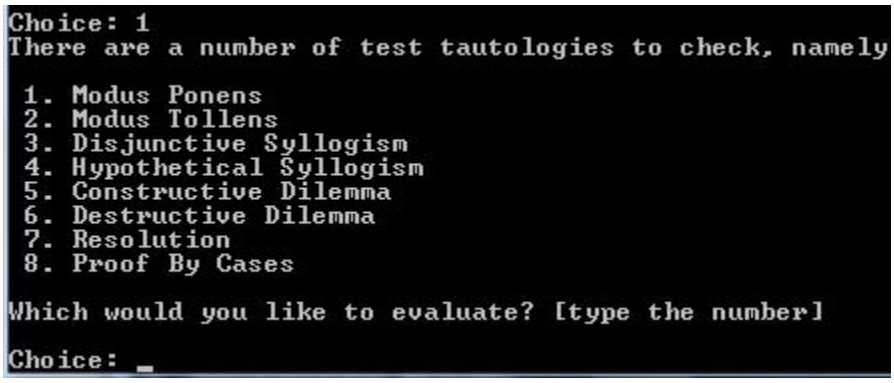

Figure 3.2: *listTautologies* displaying the programs' example list, called from *main*.

Regardless of the choice made, the strings representing the chosen/defined theorem hypotheses and conjecture are transformed into the Sentence data type through two seperate calls to the *parse'* function of the Parse module. Note the conjecture is negated before it is parsed by simply placing "!(" before its string representation and ")" after it.

The resulting sentences are passed to the *outputproof* function of Main. The sentences are transformed into CNF by the *convert* function of ConvertToCNF and prettyprinted by removing parentheses and (for the hypotheses) replacing conjunctions with new line symbols.

The original sentences are joined together by conjunction and passed to the *propreso-*

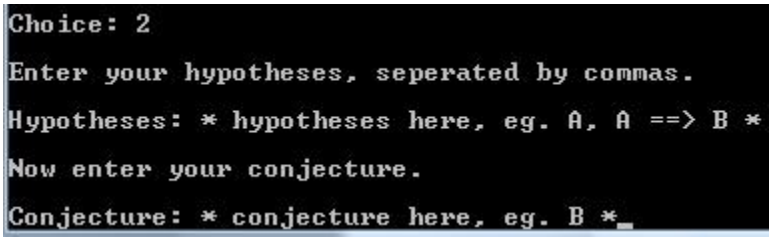

Figure 3.3: *ownTheorem* requesting details of a user-defined tautology, called from *main*.

lution function of the Resolution module. This function first applies *convert* to the conjunction it received as a parameter, before transforming it into a clausal list and initialising the fields of the Resolution record to be used by the resolution algorithm.

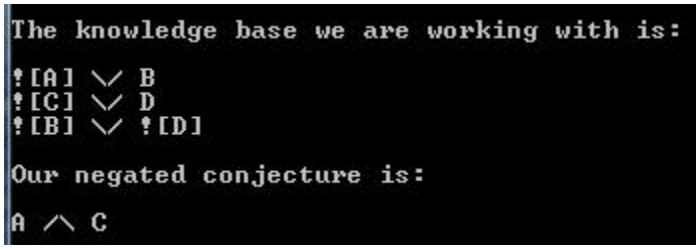

Figure 3.4: The prettyprinted CNF form of a theorems' hypotheses and conjecture.

The initialised record and and the *resolve* function are then passed to the *execState* function as parameters, handing over program control to the resolution algorithm until it terminates. The post-algorithm record is the result of the *propresolution* function, and as such, back in the Main module the *outputproof* function calls the records' accessor function *contra* to assess whether or not a contradiction has been derived.

The value that *contra* returns decides the answer that *outputproof* displays, True implying that the theorem in question is valid, as justified in §3.1. At this point the program offers the user the choice to return to *main* or to terminate the program.

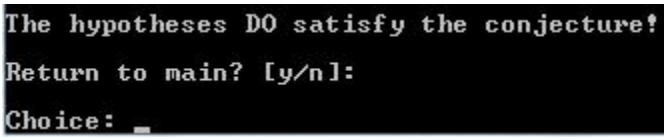

Figure 3.5: Confirmation of a theorem's validity and a request to continue or terminate, from *outputproof*.

All executions of the core implementation follow a structure more or less identical to the one given above, barring cases where the program terminates unexpectedly (currently

parse' throws an error if the input is of the wrong form) or the user makes invalid menu choices.

Whilst this implementation of the theorem prover effectively does what it says on the tin, it's not very user-friendly - as two specific examples, if the user types in any part of a theorem incorrectly the entire process must be restarted, and there isn't much appeal in typing a lengthy theorem specification into a shell window with no way of saving the result . To this end, we strip down the implementation just described to the basics and wrap a proper graphical interface around it.

3.3 Graphical Interface Implementation

Somewhat surprisingly given its reputation as a shell-only language, GUIs for Haskell programs aren't difficult to put together - a number of tools and packages exist which allow one to chain Haskell code to a front-end, such as wxHaskell and Gtk2Hs. Of these, we discuss the latter and the Glade development tool, both of which were used in this work. We then investigate the capabilities that the new interface offers the theorem prover and conclude the section by demonstrating some of the newly implemented 'damage control' for error cases.

3.3.1 Introducing Gtk2Hs/Glade

GTK+ is a toolkit used to create graphical user interfaces which over the years has had bindings to over a dozen different languages written for it - including C++, Python, Ruby and most importantly Haskell itself. The Haskell binding is creatively called Gtk2Hs, and features support for GTK+'s Glade visual user interface builder.

The Glade toolkit allows the user to quickly put together an interface using a wide choice of widgets and control options (see Figure 3.6) - the output consists of an XML file which can be passed to an GTK+ object named GtkBuilder. This accepts as input a text description of an interface and instantiates the objects within - the Gtk2Hs library then allows for Haskell code to be associated with these widgets in a manner very similar to imperative programming.

The new interface opens new doors in terms of what the program can do - namely being able to delete hypotheses or conjectures without restarting the program and displaying a log of steps taken by the resolution algorithm as a 'proof log' of sorts. We discuss these in more detail in the next section.

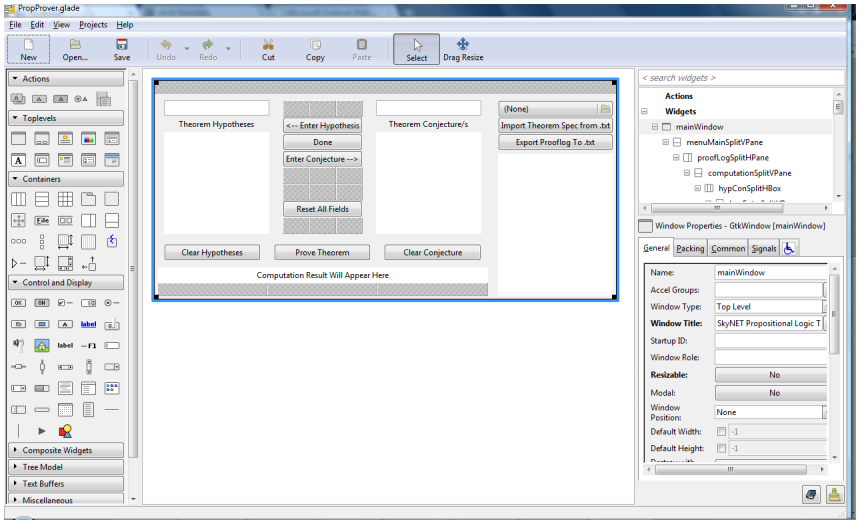

Figure 3.6: The Glade interface development tool, working on the file used for the theorem prover's interface.

3.3.2 Functionality Improvements

Temporarily shelving the obvious fact that dealing with a program that has a graphical user interface is vastly preferable to one which runs entirely in a console even if only for aesthetic reasons, the multiple-output-widget nature of a GUI allows for far more information to be available at any given time.

Further, the options available to a user for manipulating this information 'increase' - clicking a button is easier than typing in a command, and code to handle data within widgets (for example, when clearing hypotheses) is far easier to understand than that which loops back and forth within Haskell.

With that said, we look at what extra features we now have available to us in the next section.

Resetting Sections

In the core implementation, if the user inadvertently entered the hypotheses or conjectures incorrectly - by adding an extra parenthesis for example - the program would have to return a parse error notification, since there was no way to correct the mistake short of restarting the main loop (this isn't strictly true - in the core implementation there was no failsafe for incorrect input and the entire shell would terminate unexpectedly, but this was fixed before the GUI was implemented, see §3.3.4).

By way of dealing with this, the GUI allows the user to input data one clause at a time, into seperate list-boxes. The requirement that hypotheses are entered first also falls away - conjectures are allowed to be added before any hypotheses, the user can alternate between entering hypotheses and conjectures and so on. Taking advantage of this - if the user accidentally enters an incorrect clause (be it due to incorrect punctuation or otherwise) there is the option of erasing either one or both of the list-boxes and re-entering the specification. The reset/clear buttons can clearly be seen underneath the list-boxes in Figure 3.7.

Proof Logs

In the core implementation, the resolution record did not keep track of the steps taken by the algorithm to act as a proof log of sorts - mainly because displaying the proof log for a particularly lengthy proof in shell would both look particularly unappealing and also possibly exceed the maximum amount of text displayable within the console.

Thankfully this isn't a problem when using a GUI, as we can display as much text as we like in a textbox equipped with a scrollbar - which is the purpose of the box on the right-hand side of the interface (see Figure 3.7). We can now display a proof log for even the most complex of theorems with our only 'concern' being the resources needed to prettyprint it.

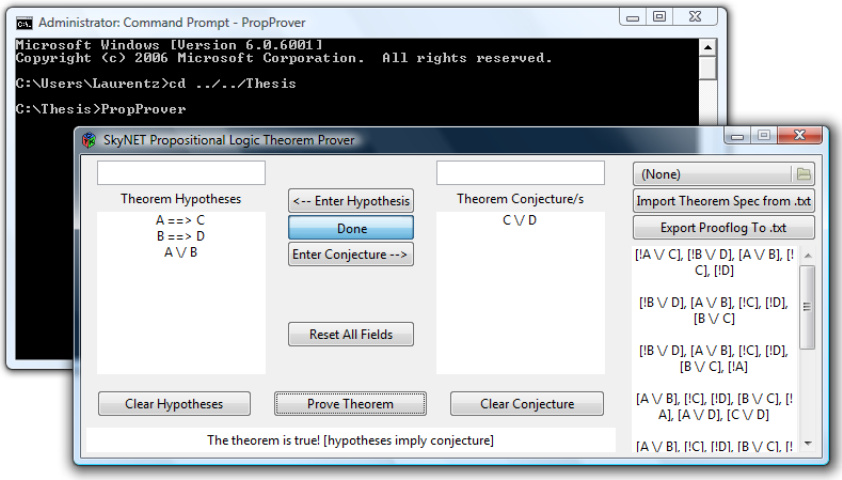

Figure 3.7: Confirming the constructive dilemma tautology, proof log appearing in the right-hand textbox.

To do this, the resolution record (recall §3.1) was extended by adding a field *prooflog* of

type [CNFSentence] initialised to the singleton list containing the original input to the algorithm. This field is updated with the remainder each time a resolvent is generated or a new resolution clause is selected. When the algorithm terminates, the log will have at its head either a clause containing a complementary pair (usually as the first and last singletons) or the empty list - indicating that either a contradiction was found or the algorithm ran out of input respectively.

This list of remainders is prettyprinted and displayed in the proof log textbox every time the algorithm is executed (whenever the *Prove Theorem* button is pressed) - whether the theorem is true or not. This allows a user to follow, step for step, the reasoning used and either refactor their theorem or draw conclusions as needed.

This proof log can be exported for future reference if required, further if a theorem specification was imported from a text file the proof log can be appended to the end of said file. This is discussed in the next section.

3.3.3 Importing and Exporting Theorems/Proofs

Being able to enter a theorem specification piecemeal has its usefulness - however, implicit in using a theorem prover is that the theorem in question is too complex to deal with manually. This may well be due to length - there isn't anything uncommon with a theorem specification of dozens of clauses. Alternately, if a user was attempting to formulate a similarly complex theorem themselves, it is entirely possible that a lot of changes will need to be made - clauses refactored, added and so on. In either case, the prospect of having to enter a theorem of such a size one clause at a time quickly becomes daunting enough to warrant not bothering to use the program in the first place.

To counteract this - we allow a theorem to be specified in a text file, and use some simple parsing to import the clauses specified in such a file directly into their appropriate listboxes, allowing the theorem to be proved immediately. Needless to say, this saves considerable time, since clauses need only be defined once in the file and then edited and removed at will.

To import a proof specification, the user clicks on the *Import Theorem Spec From .txt* button on the right-hand side of the interface and a file chooser dialog appears (see Figure 3.8). Note that there are restrictions imposed on the format of the file in order for the file to be accepted and the theorem imported - namely:

- Each clause of the theorem must begin on a new line.

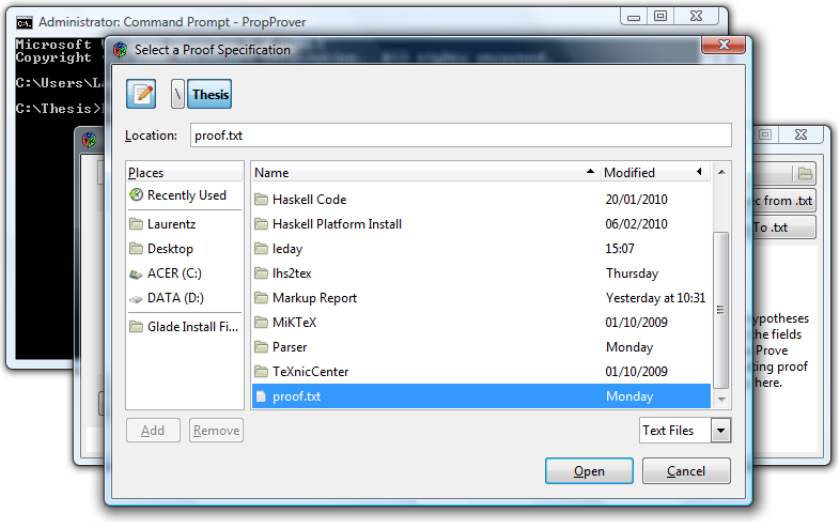

Figure 3.8: The file chooser dialog for importing proof specifications from text files.

- The hypotheses and conjectures must be defined seperately, with a '%' character on the lines above and below both sets.
- The character '%' is reserved otherwise, and the '*' character is not to be used.

Apart from these, the user is free to place comments before/after/in between the hypotheses and conjecture blocks as they see fit. Provided the above are all met, the clauses of the theorem will appear in their appropriate place, alongside a status message "*Loaded file from ...*" followed by the file path. Otherwise, a status message will appear saying "*Not a proof specification file, or too few/many % tokens present [4 required].*"

Once a theorem has been proved (or disproved) and a proof log has been generated, the user can append it to the text file currently selected by the file chooser dialog by pressing the *Export Prooflog To .txt* button on the right hand side of the interface.

The reason that '*' is a reserved character in proof specification files is that it is used in the header of the exported proof log (see Figure 3.9), acting as a flag for whether or not it has already been appended to the file in question - if this check wasn't present, the user would be able to append the log as many times as they liked. As the interface stands, if the user attempts to export a proof log to a file which has already had one appended to it, the status message "*A proof log is already associated with this file!*" is displayed.

If no file is specified in the file chooser dialog and the export button is pressed - re-

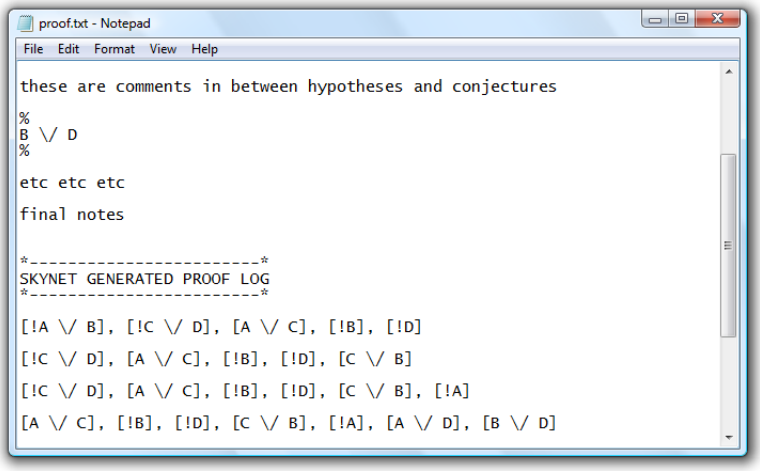

Figure 3.9: An example of a proof specification file with the proof log appended.

gardless of whether or not a proof log is present - the status message "*No file selected!*" is displayed.

3.3.4 Parsing Error Detection

In the core implementation, entering any part of a theorem specification incorrectly would cause the program to terminate immediately, giving a parseError as defined by the Happy grammar file generating the Parse module. Whilst the new interface goes some way towards removing this problem (by allowing a user to reset a part of a theorem specification), the user might still slip up somewhere without realising - classically, by putting too few/many parentheses in a clause. To help minimise errors such as these from occurring, the following checks were implemented -

- **The equalBrackets function**: before any other checks are done, the parentheses of a theorem specification are checked to ensure that all open brackets are closed or closing brackets are not where they do not need to be. To do this, the string representation of a theorem specification is passed to the *equalBrackets* function, accepting an integer and a string and returning an integer.

  The integer argument serves only to store the intermediate state of the function output. If this integer ever becomes negative, the function immediately terminates - for reasons which should be clear on the next page. Otherwise, each character of the string is examined in turn. There are four input cases:

- **Input is []** - return 0.
- **Input is '(':str** - return the result of calling (equalBrackets (n+1) str) + 1.
- **Input is ')':str** - return the result of calling (equalBrackets (n-1) str) - 1.
- **Input is _:str** - return the result of calling (equalBrackets n str).

It follows that if the result of equalBrackets is anything other than zero, there is a parenthesis mismatch and the result of parsing the string must be a ParseError (defined below). Otherwise, the string can be fed to the lexer function as before.

- **Additional Sentence constructors**: the constructors of the Sentence datatype were extended to include Null and ParseError. Correspondingly, the return type of the lexer function was changed to *Maybe [Token]* and extended by returning the (new) NullT token when given the empty list as input, and propagating errors by case matching on every recursive call to the lexer function.

  The result of the lexer is fed to a modified version of the parse function (recall the *parse'* function used in the core implementation was the composition of the lexer and parse functions). This new version case matches on the input, returning ParseError if fed Nothing or Null if fed Just [NullT]. Otherwise, the first n - 1 elements of the list of tokens are extracted from the Just constructor and passed to testValid, described below.

- **The testValid function**: it stands to reason that tokens can only appear in certain sequences, otherwise the theorem specification in question is fundamentally incorrect. For example, two conjunction tokens cannot follow each other without at least one literal token between them, and a closing parentheses cannot immediately follow an opening one.

  To catch out errors such as these, a function *testValid* was implemented as an intermediate function in the parsing process, accepting a list of tokens and returning a boolean value. This function pattern matches on the head of the input - if the list of tokens following it is empty then the function returns either True or False as appropriate - otherwise the second token is pattern matched. If this second token is a token permitted to follow the first (ie. an opening parenthesis can be followed by a negation symbol, a literal or another opening parenthesis) then the function is recursively called on the tail of the original input, else False is returned.

  In this way, if testValid returns True, we can be confident that the list of tokens in question represents a semantically correct theorem specification (if not, we return a

ParseError). From this, we can hand the list of tokens to the Happy-defined functions to parse them just as in the core implementation.

Thanks to the above, the interface now allows us to pattern match on the result of parsing the hypotheses and conjecture/s independently - the resolution algorithm is only executed if neither result is Null or ParseError. Otherwise, the program reacts as follows -

- **Either/both sections return ParseError** - a status message is displayed indicating whether the parse error is present in the hypotheses, conjecture/s or both.
- **Both sections return Null** - a status message indicating that no theorem specification has been provided is displayed.
- **The hypotheses return Null** - the program searches for contradictions in the conjecture clauses.
- **The conjecture/s return Null** - the theorem is automatically assumed to be True, as we assume all hypotheses, even if contradictory.

We have now covered all improvements to the core implementation, and are presented with the theorem prover in its final form. We conclude this chapter by looking at some of the bugs which appeared during program development.

3.4 Bugs Encountered

A number of nontrivial issues arose whilst building the theorem prover as it stands - all of which have been resolved. Due to the heavily recursive nature of some of the conversion functions and the resolution algorithm itself, some mistakes were subtle and easy to miss, only being revealed after testing (to be discussed in §4.1). Others still were oversights resulting from the implemented data types not being able to handle all input cases.

Recursive Errors in Clausal List Generation - the function in the ConvertToCNF module creating clausal lists out of sentences in CNF, *listcnf*, had a recursion error in that if there were more than two disjuncts in a disjunction, the resulting list would only seperate the first disjunct from the others, leaving the others untouched (eg. A $\vee$ B $\vee$ C would convert to [A $\vee$ B, C]). This error was easily fixed by recursively applying *listcnf* to both sides of a disjunction if the first disjunct is again a disjunction, and flattening the resulting list.

What follows are the code snippets of both versions of *listcnf* which deal with disjunctions:

```
listcnf :: Sentence → CNFSentence
listcnf (Disj a b) = [[a, b]]
```

Figure 3.10: Incorrect version of *listcnf*'s disjunction case, seperating only the first two disjuncts.

```
listcnf :: Sentence → CNFSentence
listcnf (Disj a b) = case a of
  (Disj c d) → [group (listcnf a ++ listcnf b)]
  _ → [[a, b]]
  where group xss = [x | xs ← xss, x ← xs]
```

Figure 3.11: Corrected version, checking the structure of the first disjunct and recursively applying itself if it is a disjunction, otherwise acting as before.

Recursive Errors in Algebraic Distribution - in the same vein as the previous error, when converting a sentence from the form given by the user into CNF, the *distribute* function of the ConvertToCNF module was distributing conjunctions over disjunctions incorrectly and being recursively called where there was no need for it to be. The function was entirely rewritten as a result, what follows is the corrected code snippet dealing with distribution when the literal is the first disjunct:

```
distribute :: Sentence → Sentence
distribute (Disj y z) = case y of
                        (Lit a) → case z of
                        (Conj b c) → distribute ((y `Disj` b) `Conj` (y `Disj` c))
                        (Disj b c) → y `Disj` (distribute z)
                        _ → y `Disj` z
```

Figure 3.12: Corrected *distribute* code snippet for disjunction in the case A $\vee$ (B $\wedge$ C).

Empty Hypotheses/Conjecture Strings - this problem was discussed and dealt with in §3.3.4. It is mentioned again here as it posed a significant problem for the core implementation, but can be ignored at this point.

Proof Log Prettyprint Errors - whilst simple, the prettyprinter for the proof log was not perfect when first implemented. It was possible that bracketing a sentence incorrectly in a theorem specification would, whilst still indicating a correct theorem - which the program *would* be able to infer - cause the prettyprinter to arrive at undefined cases.

For an example, note that disjunction is defined to be left-associative; if a (correct) theorem specification was entered containing a clause [$A \vee (B \vee C)$] (ie. we force right-

associativity) then the prettyprinting would reach an exception.

If this happened, the entire program would terminate unexpectedly. However, since the theorem prover is capable of doing its job regardless, it was decided that the proof log need not be perfect every time. To correct the error then, instead of terminating the program the proof log now contains the message "****Parsing error when producing proof log. Remove unnecessary parentheses****" wherever the prettyprinter attempted to deal with the cause of the error (this is simply a default case for any input other than a possibly negated literal). By deleting parentheses as requested and/or refactoring the theorem specification if necessary, the correct proof log will be given.

The above are all examples of issues which have been overcome throughout the creation of the program as is submitted. In §5.2 a number of potential *improvements* to the program are suggested for those who wish to potentially extend this work. In the next section we discuss the mechanisms used to catch the errors just discussed.

Chapter 4

Auxiliary Work - Testing/Proofs

4.1 QuickCheck Testing

As with all non-trivial programs, an element of testing needs to be present for a user to have confidence in the results given - by the very nature of a theorem prover, this involves checking a large number of varying input cases. However, as anyone having even rudimentary experience with automated theorem proving can tell you, some theorems can get very lengthy and complicated, very quickly. Generating valid test cases for theorems consisting of any more than ten or so clauses by hand soon becomes a tedious and unreasonable process, and so we hand the task over to QuickCheck.[27]

4.1.1 How Does QuickCheck Help?

QuickCheck is, quite simply, a flexible program tester. A user specifies a number of propositions which must be satisfied by a program and QuickCheck generates test data using its own library functions. This test data is then run against the program in question and any input that fails a proposition test can be flagged for inspection, giving rise to a (pun unavoidable) quick way to check that functions do what they are supposed to.

Using QuickCheck brought a number of issues with the core implementation of this work to light, mostly along the lines of errors with algebraic distribution and infinite recursion in obscure branches of code - discussed in §3.4 - if any remain in the submitted program they are so far-flung as to be nearly negligible.

Throughout this section we discuss how QuickCheck is capable of generating test data to run against the program given the nature of the Sentence datatype and analyse the propositions being tested.

4.1.2 The Sentence Generator

The Sentence datatype used throughout this work is best visualised as a tree with between zero and two branches at each node. Clearly, an instance of such a tree has potential to be infinite, with each branch length independent of any other - this raises the question of whether an automated checker is capable of creating sufficiently 'complicated' Sentences to make testing non-trivial.

QuickCheck produces test input by creating instances of datatypes which are members of the Arbitrary typeclass - simply defined as those datatypes for which arbitrary elements can be generated. Sentence is fortunately such a datatype thanks to QuickCheck's library functions; below is the instantiation of Sentence as a member of Arbitrary.

```
158 instance Arbitrary Sentence where
159   arbitrary     = sengen
160   coarbitrary _ = variant 0
161
162 sengen :: Gen Sentence
163 sengen = sized sengen'
164
165 sengen' :: Int -> Gen Sentence
166 sengen' 0 = liftM Lit arbitrary
167 sengen' n
168       | n > 0 = frequency [(1, liftM2 Conj subsen subsen),
169                            (1, liftM2 Disj subsen subsen),
170                            (1, liftM2 Impl subsen subsen),
171                            (1, liftM2 Equi subsen subsen),
172                            (2, liftM Neg subsen),
173                            (1, liftM Brack subsen),
174                            (2, liftM Lit arbitrary)]
175         where subsen = sengen' (n `div` 2)
```

Figure 4.1: The QuickCheck Sentence generator.

QuickCheck begins producing comparatively small test cases which gradually increase as time progresses. The above code utilises two combinator functions, *sized* and *frequency*. The first ensures that the Sentence generator is dependent on an (increasing) size parameter, whilst the second forces the generator to make a choice between the constructors of Sentence, with negation and literals occurring twice as often as the others (tipping the balance towards a terminating tree), calling the generator recursively when needed with the size parameter halved, terminating on a literal as we would expect.

In this way, QuickCheck creates instances of the Sentence datatype which are not only sound in their construction and independent and identically distributed in each branch, but with the potential to be as large as the user requires by changing the number of tests QuickCheck runs on each call (time and space permitting). On the downside, these instances very rarely denote a genuine theorem, and so filters must be put in place - discussed in the next subsection.

4.1.3 Test Propositions

As mentioned previously, QuickCheck runs test input against propositions specified by the user - by default a proposition 'passes' if a hundred tests return as True (the number of tests is editable). Conversely, if any input returns False, QuickCheck will display it to the user for debugging purposes.

Nine propositions were tested in this work, each one representing a tautology of propositional logic - a failure in any of these indicates an issue with either the conversion from given input to CNF or the resolution algorithm itself. They are as follows -

- **Single Negation:** $A \neq \neg A \; \forall \; A$.
- **Double Negation:** $A = \neg\neg A \; \forall \; A$.
- **Law of Excluded Middle:** $A \vee \neg A \Rightarrow \top \; \forall \; A$.
- **Falsum Principle:** $A \wedge \neg A \Rightarrow \bot \; \forall \; A$.
- **Law of Contraposition:** $(A \Rightarrow B) \Leftrightarrow (\neg B \Rightarrow \neg A) \; \forall \; A, B$.
- **Reductio Ad Absurdum:** $[(\neg A \Rightarrow B) \wedge (\neg A \Rightarrow \neg B)] \Rightarrow A \; \forall \; A, B$.
- **De Morgan's Law:** $\neg(A \wedge B) \Leftrightarrow (\neg A \vee \neg B) \; \forall \; A, B$.
- **Principle of Syllogism:** $[(A \Rightarrow B) \wedge (B \Rightarrow C)] \Rightarrow (A \Rightarrow C) \; \forall \; A, B, C$.
- **Proof By Cases:** $[(A \vee B) \wedge (A \Rightarrow C) \wedge (B \Rightarrow C)] \Rightarrow C \; \forall \; A, B, C$.

Note that the sentences A, B and C above have no parameters on their construction - all generated test cases should satisfy the above. However, it was mentioned previously that all branches of a generated sentence are independent and identically distributed - this allows us to enforce restrictions necessary to test certain propositions. To illustrate this example, Figure 4.2 is the proposition representing De Morgan's Law.

```
38 propDeMorgan xs = (case xs of (Neg (x `Conj` y)) -> True
39                               _                  -> False) ==>
40                    case xs of
41                    (Neg (x `Conj` y)) -> not (contra (propresolution ((convert xs)
42                                                                       `Equi`
43                                                                       ((convert (Neg x)) `Disj` (convert (Neg y))))))
44              where types = xs::Sentence
45
```

Figure 4.2: De Morgan's Law represented as a QuickCheck proposition.

It reads - given a generated Sentence *xs*, if it is the negation of a conjunction then we test it - otherwise we force QuickCheck to generate a new one and try again. To test the proposition, we apply the resolution algorithm to the Sentence constructed by chaining *xs* to the disjunction of the negations of its original conjuncts with the equivalence constructor. If there is no contradiction detected by the algorithm (picked up by *not(contra...)*) then the proposition holds and the test passes for that particular *xs*.

For the most part, QuickCheck's output consists of either a message "*OK, passed n tests.*" or returning the input which caused the proposition to fail. However, in propositions where some test cases are rejected, it is possible that QuickCheck completes the number of tests that it was asked to perform but less than that number made it past the filter/s in place. In this case the message returned is "*Arguments exhausted after m tests.*" meaning that m of the n generated Sentences were of the correct form, of which all of them passed.

Below is a figure illustrating the output from testing four of the nine propositions.

```
*Main'> test propFalsum
OK, passed 100 tests.
*Main'> test propDeMorgan
Arguments exhausted after 27 tests.
*Main'> test propLEM
OK, passed 100 tests.
*Main'> test propContraposition
OK, passed 100 tests.
*Main'>
```

Figure 4.3: QuickCheck informing the user that four propositions passed testing, however only 27 valid cases were generated for propDeMorgan.

The set of testable propositions in this work is easily extendable by adding to the QCPropositions.hs file of the source code. We now look at the other half of testing considered - examining code coverage.

4.2 Haskell Program Coverage (HPC)

HPC is a toolkit bundled with the Glasgow Haskell Compiler used to record the proportion of code that is actually executed when interacting with a program, allowing for the user to pinpoint redundant code or specific cases where obscure errors may arise[28]. As a result, we can keep the code needed for full functionality to a minimum, making for easier reading and future modification.

4.2.1 What Does HPC Provide?

To use HPC, one compiles the program source into an executable (using *ghc -fhpc Example.hs –make*) and then runs the binary. By putting the program through its paces, HPC marks off code which has been executed by updating a .tix 'tick file' - an array corresponding to each line of code in the program. When the program has been tested to a 'suitable' degree, a markup report is generated (with *hpc markup Example.exe*) consisting of a general overview of the percentage of each module which has been executed as well as a copy of the source code for each module, with certain lines highlighted in colours varying by reason. Below follow screenshots of the overview and an example of code markup -

Top Level Definitions		Alternatives		Expressions	
%	covered / total	%	covered / total	%	covered / total
100%	1/1	88%	38/43	97%	552/567
100%	17/17	90%	40/44	94%	258/274
90%	9/10	71%	47/66	71%	219/307
76%	51/67	59%	134/225	72%	416/576
82%	78/95	68%	259/378	83%	1445/1724

Figure 4.4: An example of the summary report generated by HPC - each modules' statistics are displayed seperately along with a mean average. Note - 'Alternatives' refers to pattern matches on input and case statements, and 'Expressions' are checked at every level of a program.

```
47                                  False -> widgetGrabFocus hypEntry
48
49          onClicked hypButton $ do
50                theoremEntered <- toggleButtonGetActive doneButton
51                case theoremEntered of
52                     True  -> entrySetText statusText "Cannot enter new hypotheses
53                     False -> do
54                              hypothesis <- get hypEntry entryText
55                              case hypothesis of ""  -> putStrLn "No hypothesis ent
56                                                 _   -> do
57                                                        hypContents <- textViewGetBu
```

Figure 4.5: An example of code markup. Yellow highlights indicate that this code has not been executed yet when testing. Other colours used are green for boolean expressions which always evaluate to true, and red for false.

With this markup report as an aid, we can now start improving our code.

4.2.2 How Does HPC Help?

The code colour markup is a particularly helpful resource - since it actively informs the user what code is left to check to cover all bases it serves as a guide to hunting down bugs. By directing the program in such a way that 'coloured code' must be executed, the tester can guarantee that they won't be caught off guard by a hitherto dormant function throwing a wrench in the works. Further, if a crash occurs, HPC effectively pinpoints the precise line where the errors arise.

When moving the program from the core shell implementation to the Glade GUI a number of modules were removed due to redundancy - for example QuickCheck, QCPropositions and Tautology were removed since all algorithmic testing had already been performed on the shell implementation, and the PropProver module which replaced the old Main has no way of accessing the propositions defined in Tautology.

However, some functions which were defined in the modules which *remained* became redundant - for example a number of functions in the ConvertToCNF module dealt with prettyprinting to a console. By reading the markup report, these functions were uncovered and either updated to interact with the new GUI or removed completely.

Further, HPC helped reveal that some of the case branches of functions involved in the conversion process into CNF were redundant, since previously applied functions eliminated the possibility of certain Sentence structures arising. Whilst one could argue that having a total function is more ideal from a coverage standpoint, having less code to read through is a bonus from a debugging point of view and so the functions have been 'streamlined', so to speak.

A copy of the coverage overview is included in the Appendix (§7.2) and the code markup is included in the electronic submission of this work for interest. Bear in mind that since the Parse module was automatically generated and the code logic is as good as unreadable, testing it was a matter of entering as much input as possible with no real guarantee that all code would be executed. As a result, its coverage percentage was expected to be significantly lower than the other modules.

4.3 Suitability of Resolution Refutation

This section concludes with proofs of the logical soundness and completeness of resolution refutation, justifying its usage throughout. We do this through proving the stability and/or existence of *models* for sets of clauses, and thus abstract away from the details of any specific theorem.

These proofs require basic familiarity with the terminology introduced by Robinson[17] - however definitions of terms used in this section are given below.

A **ground clause/constant** is simply a synonym for a clause/constant respectively. The word 'ground' indicates the presence of no variables, however this must be the case in propositional logic by the arity of the predicates.

A **model** is a set of ground constants containing no complementary constants (for example, A and $\neg$A). A model Γ is a model for a set of ground clauses S if:

$$\forall\, C \subseteq S;\ \exists\, c \in C \text{ such that } c \in \Gamma.$$

4.3.1 Proof of Soundness

To prove the soundness of a logical system paired with resolution refutation, we must prove that any model that models an initial set of assumptions also models any possible resolvents from that set.

Theorem I

Given a set of ground clauses S, any model Γ of S is also a model of $Res^*(S)$, where $Res^*(S)$ is the union of S and the set of all possible resolvents derivable from S.

Proof

Let A and B be two clauses in $S \cup Res(S)$ which can generate a resolvent, where Res(S) is the set of all resolvents obtained thus far. Without loss of generality we can assume that they are both of the form $A = \alpha \vee \gamma$ and $B = \neg\gamma \vee \beta$, where either α or β or both can be []. We must show that Γ is also a model of $C = \alpha \vee \beta$.

We proceed by case analysis on γ.

Case I, $\gamma \in \Gamma$ - since Γ models A, and $\neg\gamma \notin \Gamma$ (else Γ would contain a complementary pair and thus not be a model), it must be the case that $\exists\, c \in \beta$ such that $c \in \Gamma$, hence Γ models C.

Case II, $\neg\gamma \in \Gamma$ - in a symmetric argument to the above, Γ models A, $\gamma \notin \Gamma$ and it must be the case that $\exists$ c $\in \alpha$ such that c $\in \Gamma$, hence Γ models C.

Case III, γ, $\neg\gamma \notin \Gamma$ - this is trivial, the result is immediate.

In any case, the result follows. ■

4.3.2 Proof of Completeness

Proving the completeness property is a trickier exercise. We show that if a set of ground clauses S is satisfiable then it has a model, in turn proving that if S is unsatisfiable then this contradiction can be derived using resolution. Credit for the approach to the proof goes to Waldmann[26].

Theorem II

Given a set of ground clauses S, if S semantically entails $\perp$ (S $\models \perp$) then $\perp$ can be derived using resolution (S $\vdash \perp$). Equivalently, if $\perp \notin$ Res*(S) then S has a model.

Proof

To begin, we introduce the concept of a lexical ordering $\leq$ on the ground literals in S with two basic properties -

1. Given any two literals, the literal denoted by the character closer to the end of the alphabet has precedence, eg. A $\leq$ C, E $\leq$ K. (in reality, the level of precedence assigned to each literal is a choice for the reader to make, provided they are consistent).
2. Given a complementary pair of literals (ie. A and $\neg$A) then A $\leq \neg$A.

This ordering can be easily extended to an ordering on ground clauses as follows: given any two clauses α and β, take the literal with the highest precedence from each (the *leading literals*, call them A and B) and compare the two. If A $\leq$ B, then $\alpha \leq \beta$ and vice-versa.

Armed with this, we consider the set S itself. The process of generating resolvents from an initial set must eventually terminate, since resolution introduces no new literals - as a result of this we know that Res*(S) is a finite set (we use this fact in the algorithm below). We assume at this point that S $\nvdash \perp$.

Construction of the model of S is done by examining each clause and resolvent of S in turn, and adding leading literals to the model when certain conditions are satisfied (it is worth noting that if we did not assume that S $\nvdash \perp$, clauses for which these conditions do

not hold are potential counter-examples of the satisfiability of S).

To do this, we first assign levels of precedence to the original clauses of S using the extended version of $\leq$. With an order now imposed, we apply the following algorithm -

1. Initialise all clauses as being 'unexamined', and the model M as $\emptyset$. [This ensures that S = $\emptyset$ is satisfiable.]
2. Consider the lowest precedence unexamined clause, which we denote as λ.
   - If it is possible to generate a (previously unseen) resolvent from λ and another clause of higher precedence, give this (unexamined) resolvent a precedence one rank lower than λ and loop back to step 2.
   - If λ contains multiple instances of the same literal (ie. [A, B, B, C]) then generate a new clause with repetitions removed (ie. [A, B, C]), assign this (unexamined) clause a precedence one rank lower than λ and loop back to step 2.
3. Consider the leading literal of λ, which we denote Δ.
   - If Δ is a strict leading literal (no repetitions of Δ in λ) occurring *positively* in λ, and M $\not\models \lambda$, then add Δ to M.
   - If this is not the case, M remains unchanged.
4. Mark λ as examined. If all clauses have been examined, terminate - else loop back to step 2.

Once the algorithm has terminated, it is a comparatively easy exercise to confirm that this M models S, and the result follows. ■

Chapter 5

Conclusion and Future Work

5.1 Critical Appraisal and Conclusion

To wrap everything up, we look at what this dissertation set out to achieve, both as a grand vision and a real-world implementation, discussing the differences the author uncovered between the two over the course of the eight months in which work was done. We then finish outright with a handful of suggestions for further work which the author would like to implement/see implemented.

We began by saying that we wished to implement a propositional logic theorem prover in Haskell, true to the title of the dissertation. We chose propositional logic over first-order logic due to intractability reasons, a theorem prover because of the varied applications which they can be applied to and Haskell over any other language due to its suitability to symbol processing.

In terms of the write-up itself, this was done by first discussing the syntax and semantics of propositional logic, introducing all of the notation and terminology used throughout the rest of the work. Then we discussed logical inference, how we need to manipulate theorems into a form which a computer can apply iterative rules to and how the chosen rule of resolution refutation can prove the (maybe lack of) validity of a theorem.

We then moved on to discussing the algorithm which we use throughout the work to implement the resolution rule, illustrating it with a running example. We used this algorithm in a shell-only version of a theorem prover which was described by a description of all major data types involved as well as a summary of each module, how they fit together and an illustration of an example run through the program. This version is available in the electronic submission as *ShellProver.exe*.

This shell-only version was then extended by wrapping a Glade interface around it (and cutting out a number of unnecessary functions) and introducing a number of other features such as exporting proof logs and detecting parse errors. The final version of the theorem prover is available in the electronic submission as *PropProver.exe*.

We finished by discussing the testing which was done throughout the construction of the theorem prover - QuickCheck for the shell version and HPC for the interfaced one. Both toolkits were introduced alongside quick sections explaining how they are applied to the theorem prover.

As nice as it would be to say that all of the above was thought of and implemented with ruthless precision, the reality is that a number of issues were remarkably difficult to get to grips with (if even the difficulty only lay in translating from pseudocode into Haskell). Moreover several seemingly easy tasks - such as implementing axioms of logical equivalence for conversion into CNF and writing the Happy parser generator file as two examples - ended up taking weeks rather than days simply because the author had either forgotten the finer details of what needed to be done or never knew them in the first place.

With that said though, taking the knowledge of propositional logic learnt from textbooks and taught courses and applying it as we have done has proven very satisfying! The author has learnt more about logic and theorem proving (as well as a fair few of the inherent difficulties involved) through his own research and simply hacking away in Haskell than he could have hoped to over the course of the rest of his degree or otherwise.

Comparing the end product *PropProver.exe* to expectations taken from the introductory chapter, there is just one major comment to be made. It was initially thought that implementing an interface for the program would be the hardest part of the coding work that needed to be done - however as it turned out, Glade is an amazingly efficient tool once the trial by fire of installing/configuring it has been dealt with. The result is an interface much slicker than the author expected, adding far more by way of usability than simply a handful of buttons to save on effort spent entering a theorem.

In closing - all of the features listed in the 'feature checklist' of §1.2 have been met either fully or to the point where exception cases are expected to be very rare - we refer here to theorems which force the program to crash unexpectedly or introduce nontermination - however, QuickCheck and HPC have hopefully sufficiently dealt with this. The author believes that this work has succeeded in what it set out to do - at least two first year students have expressed interest in using the program as an educational aid for getting to grips with propositional logic.

5.2 Possible Improvements

There are several ways in which the functionality and performance of the current theorem prover can be improved upon, which are briefly discussed here. Performance wise, resolution as a procedure is bounded above by time and space complexities exponential in the number of unique literals n in the original KB which we apply our algorithm to[24]. The improvements which we *can* implement do not (and cannot) aim to improve upon these things, but rather the confidence which we have in our answers and some 'tweaks' to avoid employing our algorithm when unnecessary.

5.2.1 Implement the DPLL Algorithm

A seminal paper by Davis and Putnam suggests an alternative resolution-based algorithm for finding proof by contradictions in quantification theory (first-order logic)[25], further refined into what we now know as the *Davis-Putnam-Logemann-Loveland* (DPLL) algorithm. The author feels that this algorithm may be worth implementing to compare it against the currently used algorithm, particularly in the senses that less Resolution record fields may need to be tracked, and debugging may become easier.

Alternately, the algorithm could be implemented and a choice offered to the user as to which algorithm they wish to apply to their theorem specification, be it for a 'second opinion' of sorts or to produce a proof-log which they find easier to follow.

5.2.2 Look For Immediate Contradictions

The first step of the DPLL algorithm mentioned above is to check if the KB is 'self-contradictory' - or rather if there exist two unit clauses [K] and [¬K] leading to a contradiction. Whilst this is a part of the DPLL algorithm itself, the principle can easily be justified for the currently used algorithm - there is no need for the algorithm to be applied to the clausal list [[A, B], [C, ¬D], [E], [¬A, B], [¬E]] until termination, when the contradiction is obvious.

Implementing this check should be a matter of adding a few lines of code to the algorithm, to be executed before each new resolution clause is selected.

5.2.3 Integrate *Propcode.hs*

An extended example within Huttons' book[14] to explain types and classes is a tautology checker which uses boolean assignments (this was mentioned in §1.3). Whilst initially written for a datatype only representing a few constructors of propositional logic, this was

easily extended to accept the Sentence datatype used in this work. The end result is a function *isTaut*, accepting a Sentence and returning a boolean value.

Whilst not directly related to the business of proving a theorem using resolution refutation, this function may serve a purpose to check that the result arrived at both the algorithm and by assignment of booleans to each literal agree. Whilst the code is readily available, it was not implemented in the submitted program as it would serve more purpose in a testing environment. To this end, a number of QuickCheck propositions could be defined to check that both approaches agree on random input.

Chapter 6

Bibliography

1. McCune W. *Robbins Algebras Are Boolean* [online]. 1996. Available at: http://www.cs.unm.edu/ mccune/papers/robbins [5th December 2009]

2. Appel K, Haken W. Solution of the Four Color Map Problem. *Scientific American*, October 1977, 237(4), pp. 108-121

3. Hales T C. A proof of the Kepler conjecture. *Annals of Mathematics*, 2005, Second Series 162(3), pp. 1065-1185

4. Zeng Y. *Propositional vs First-Order Logic [Lecture Notes: Artificial Intelligence Programming - Logic Programming]* [online]. Aalborg University, Spring 2009. Available at: http://www.cs.aau.dk/ yfzeng/course/AIP/index.html [5th December 2009]

5. Mironov I, Zhang L. Applications of SAT Solvers to Cryptanalysis of Hash Functions, *Lecture Notes in Computer Science*, July 2006, 4121, pp. 102-115

6. Kuphaldt T R. *Lessons In Electric Circuits* [online]. 4th ed. November 2007. Volume IV. Available at: http://openbookproject.net/electricCircuits/Digital/DIGI_7.html [5th December 2009]

7. Schellhorn G, Ahrendt W. The WAM Case Study: Verifying Compiler Correctness for PROLOG with KIV. *In:* Bibel W, Schmitt P. editors, *Automated Deduction - A Basis for Applications.* Volume III: Applications. Dordrecht: Kluwer Academic Publishers, 1998

8. Butler, R W. *Formalization of the Integral Calculus in the PVS Theorem Prover*. NASA Langley Research Center: October 2004, (L-18391)

9. Koncaliev D. *Pentium FDIV bug* [online]. Available at: http://www.cs.earlham.edu/ dusko/cs63/fdiv.html [5th December 2009]

10. Palka M. *Functional Programming - Purity* [online]. Last updated by user Nedervold, 24th November 2009. Available at: http://haskell.org/haskellwiki/Functional_programming#Purity [5th December 2009]

11. Sutcliffe G, Suttner C. The state of CASC. *AI Communications*, 2006, 19(1), pp. 35-48

12. Riazanov A, Voronkov A. The design and implementation of VAMPIRE. *AI Communications*, 2002, 15(2-3), pp. 91-110

13. Eriksson L-H. *A Comparison of Four Propositional Theorem Provers*, Stockholm, 2000, (L4i-00/111)

14. Hutton G M. *Programming in Haskell*. 1st ed. Cambridge: Cambridge University Press, 2007

15. Wilson F. *Third Year Dissertation Project - A Haskell Theorem Prover*. BSc dissertation, University of Sheffield, 2006

16. Brachman R J, Levesque H J. *Knowledge Representation and Reasoning*. 1st ed. San Francisco, CA: Morgan Kaufmann, 2004

17. Robinson J A. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 1965, 12(1), pp. 23-41

18. Hilbert D, Ackermann W, ed. Luce R E. *Principles of Mathematical Logic*. 2nd ed. American Mathematical Society, 1999

19. Bezhanishvili G, Moss L S. *Undecidability of First-Order Logic* [online]. 2008. Available at: http://www.cs.nmsu.edu/historical-projects/Projects/FoLundecidability.pdf [12th May 2010]

20. O'Sullivan B, Goerzen J, Stewart D. *Real World Haskell*. 1st ed. Sebastopol, CA: O'Reilly Media Inc., 2008

21. Peyton-Jones S. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. *In:* Hoare T, Broy M, Steinbruggen R. editors, *Engineering Theories of Software Construction*. IOS Press, 2001, pp. 47-96

22. Marlow S, Gill A. *Happy User Guide* [online]. 1997. Available at: http://www.haskell.org/happy/doc/html/index.html [5th December 2009]

23. Garshol L M. *BNF and EBNF: What are they and how do they work?* [online]. 2003. Available at: http://www.garshol.priv.no/download/text/bnf.html [5th December 2009]

24. Schöning U. Resolution Proofs, Exponential Bounds, and Kolmogorov Complexity. *Lecture Notes in Computer Science*, 1997, 1295, pp. 110-116

25. Davis M, Putnam H. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 1960, 7(3), pp. 201-215

26. Waldmann U. *Refutational Completeness of Resolution [Lecture Notes: Automated Reasoning]* [online]. Max Planck Institut Informatik, Summer 2004. Available at: http://www.mpi-inf.mpg.de/ uwe/lehre/autreas/readings.html [9th March 2010]

27. Claessen K, Hughes J. QuickCheck: a lightweight tool for random testing of Haskell programs. *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, 2000, 35(9), pp. 268-279

28. Gill A, Runciman C. Haskell Program Coverage. *Proceedings of the 2007 ACM SIGPLAN Workshop on Haskell*, 2007, pp 1-12

Chapter 7

Appendix

7.1 Happy Parser Generator File

```
{
module Main where
}

%name parse
%tokentype { Token }
%error { parseError }

%token
    '(' { TokenOB }
    ')' { TokenCB }
    '!' { TokenNeg }
    "/\\"   { TokenConj }
    "\\/"   { TokenDisj }
    "==>"   { TokenImpl }
    "<=>"   { TokenEqui }
    ',' { TokenCmma }
    str { TokenLit $$ }

%%

Sentence    : Sentence "/\\" Sentence   { Conj $1 $3 }
        | Sentence "\//" Sentence   { Disj $1 $3 }
        | Sentence "==>" Sentence   { Impl $1 $3 }
        | Sentence "<=>" Sentence   { Equi $1 $3 }
        | '!' Sentence          { Neg $2 }
        | '(' Sentence ')'      { Brack $2 }
        | Lit String            { Lit $2 }

{
```

```
parseError :: [Token] -> a
parseError _ = error "Parse Error"

data Sentence = Sentence `Conj` Sentence
              | Sentence `Disj` Sentence
              | Sentence `Impl` Sentence
              | Sentence `Equi` Sentence
              | Neg Sentence
              | Lit String
              | Brack Sentence
              | Null
              | ParseError

data Token = TokenOB
       | TokenCB
       | TokenNeg
       | TokenConj
       | TokenDisj
       | TokenImpl
       | TokenEqui
       | TokenCmma
       | TokenLit String
       | NullT

 lexer :: String -> [Token]
 lexer []                 = [NullT]
 lexer ('\n':cs)          = [NullT]
 lexer ('\r':cs)          = [NullT]
 lexer (c:cs)     | isSpace c = lexer cs
                  | isAlpha c = lexLit (c:cs)
 lexer ('(':cs)           = TokenOB:(lexer cs)
 lexer (')':cs)           = TokenCB:(lexer cs)
 lexer ('!':cs)           = TokenNeg:(lexer cs)
 lexer ('\\':'/':cs)      = TokenDisj:(lexer cs)
 lexer ('/':'\\':cs)      = TokenConj:(lexer cs)
 lexer ('=':'=':'>':cs)   = TokenImpl:(lexer cs)
 lexer ('<':'=':'>':cs)   = TokenEqui:(lexer cs)
 lexer (',':cs)           = TokenCmma:(lexer cs)
 lexer (_:cs)             = lexer cs

 lexLit cs = case (span isAlpha cs) of
                  (l,rest)   -> (TokenLit l):lexer rest

 }
```

7.2 HPC Report Overview

module	Top Level Definitions	
	%	covered / total
module ConvertToCNF	100%	9/9
module Main	100%	1/1
module Parse	76%	51/67
module Resolution	100%	17/17
Program Coverage Total	82%	78/94

Alternatives		Expressions	
%	covered / total	%	covered / total
85%	52/61	84%	236/280
93%	40/43	95%	541/569
61%	140/227	74%	438/591
90%	40/44	94%	258/274
72%	272/375	85%	1473/1714