

COMPUTING SCIENCE

Towards a Mechanisation of a Logic that Copes with Partial Terms

C. B. Jones, M. J. Lovert and L. J. Steggles

TECHNICAL REPORT SERIES

No. CS-TR-1314

February 2012

Towards a Mechanisation of a Logic that Copes with Partial Terms

C.B. Jones, M.J. Lovert, L.J. Steggles

Abstract

It has been pointed out by a number of authors that partial terms (i.e. terms that can fail to denote a value) arise frequently in the specification and development of programs. Furthermore, earlier papers describe and argue for the use of a non-classical logic (the "Logic of Partial Functions") to facilitate sound and convenient reasoning about such terms. This paper addresses some of the issues that arise in trying to provide (semi-)decision procedures -such as resolution- for such a logic. Particular care is needed with the use of "proof by refutation". The paper is grounded on a semantic model.

Bibliographical details

JONES, C.B., LOVERT, M.J., STEGGLES, L.J.

Towards a Mechanisation of a Logic that Copes with Partial Terms

[By] C.B. Jones, M.J. Lovert, L.J. Steggles

Newcastle upon Tyne: Newcastle University: Computing Science, 2012.

(Newcastle University, Computing Science, Technical Report Series, No. CS-TR-1314)

Added entries

NEWCASTLE UNIVERSITY

Computing Science. Technical Report Series. CS-TR-1314

Abstract

It has been pointed out by a number of authors that partial terms (i.e. terms that can fail to denote a value) arise frequently in the specification and development of programs. Furthermore, earlier papers describe and argue for the use of a non-classical logic (the "Logic of Partial Functions") to facilitate sound and convenient reasoning about such terms. This paper addresses some of the issues that arise in trying to provide (semi-)decision procedures -such as resolution- for such a logic. Particular care is needed with the use of "proof by refutation". The paper is grounded on a semantic model.

About the authors

Cliff B. Jones is currently Professor of Computing Science at Newcastle University. As well as his academic career, Cliff has spent over 20 years in industry. His 15 years in IBM saw among other things the creation –with colleagues in Vienna– of VDM which is one of the better known “formal methods”. Under Tony Hoare, Cliff wrote his doctoral thesis in two years. From Oxford, he moved directly to a chair at Manchester University where he built a world-class Formal Methods group which –among other projects– was the academic lead in the largest Software Engineering project funded by the Alvey programme (IPSE 2.5 created the “mural” (Formal Method) Support Systems theorem proving assistant). He is now applying research on formal methods to wider issues of dependability. Until 2007 his major research involvement was the five university IRC on “Dependability of Computer-Based Systems” of which he was overall Project Director. He is also PI on an EPSRC-funded project “AI4FM” and coordinates the “Methodology” strand of the EU-funded DEPLOY project. He also leads the ICT research in the ITRC Program Grant. Cliff is a Fellow of the Royal Academy of Engineering (FREng), ACM, BCS, and IET. He has been a member of IFIP Working Group 2.3 (Programming Methodology) since 1973 (and was Chair from 1987-96).

Matthew is a PhD student at Newcastle University under the supervision of Prof. Cliff Jones and Dr. Jason Steggles. He is undertaking research on mechanised proof support tools for the Logic of Partial Functions. Matthew gained his BSc in Computing Science with First Class Honours from Newcastle University in 2008, during which time he was awarded with a BCS prize, a Scott Logic prize and a British Airways prize for outstanding performance in each stage of his degree course.

Dr L. Jason Steggles is a lecturer in the School of Computing Science, University of Newcastle. His research interests lie in the use of formal techniques to develop correct computing systems. In particular, he has worked extensively on using algebraic methods for specifying, prototyping and validating computing systems.

Suggested keywords

LOGIC

PARTIAL FUNCTIONS

LPF

RESOLUTION

UNIFICATION

REFUTATION PROCEDURE

Towards a Mechanisation of a Logic that Copes with Partial Terms

C. B. Jones

M. J. Lovert

L. J. Steggles

February 12, 2012

Abstract

It has been pointed out by a number of authors that partial terms (i.e. terms that can fail to denote a value) arise frequently in the specification and development of programs. Furthermore, earlier papers describe and argue for the use of a non-classical logic (the “Logic of Partial Functions”) to facilitate sound and convenient reasoning about such terms. This paper addresses some of the issues that arise in trying to provide (semi-)decision procedures –such as resolution– for such a logic. Particular care is needed with the use of “proof by refutation”. The paper is grounded on a semantic model.

1 Introduction

Within logical expressions, terms can fail to denote proper values and as a result logical formulae involving such terms may not denote Booleans [Jon90, Jon06, MS97]. Such partial terms arise frequently –for example when applying recursive functions– in the specification of computer programs; more tellingly, reasoning about such terms is required when discharging the proof obligations generated in justifying development steps (such proof obligations can be very large for industrial applications). This raises the question of how one can reason about such formulae. Numerous approaches have been conceived over the years — most are documented in [Che86, CJ90, CJ91, Jon06, Owe85, GSE95, MS97, Fit07, JL11, Sch11].

The issue of non-denoting terms can be illustrated by the following property using integer division:

$$\forall i: \mathbb{Z} \cdot (i \div i = 1) \vee ((i - 1) \div (i - 1) = 1) \quad (1)$$

When i has the value 0, the first disjunct fails to denote a value; similarly, the second disjunct fails to denote a value when i has the value 1. The best way of thinking about the issue is to see that there is a “gap” in the denotation of the integer division operator (this view is formalised in Section 4).¹ It is however convenient to illustrate the difficulties by writing $\perp_{\mathbb{Z}}$ to stand for a missing integer value (and $\perp_{\mathbb{B}}$ for a missing Boolean value). The validity of Property 1 relies on the truth of disjunctions such as $(1 \div 1 = 1) \vee (0 \div 0 = 1)$, which further reduces to $(1 = 1) \vee (\perp_{\mathbb{Z}} = 1)$. With strict (weak/computational) equality (undefined if either operand is undefined), this further reduces to $true \vee \perp_{\mathbb{B}}$ which makes no sense in classical logic since its truth tables only define the logical operators for proper Boolean values.

The approach that the current authors take to reasoning about logical formulae that include partial terms is to employ a *non-classical logic* known as the *Logic of Partial Functions (LPF)* [BCJ84, Che86, CJ91, JM94, Jon06, JL11], where “gaps” are handled by extending the logical operators. Property 1 is true in LPF and its proof presents no difficulty. However, Property 1 can be the cause of “issues” in other approaches to coping with non-denoting terms — for example, with McCarthy’s conditional interpretation² of the logical operators [McC67], where disjunctions and conjunctions are not commutative and quantifiers are problematic with respect to undefined values.

However, the availability of a large body of (semi-)decision procedures (as well as tool support) for classical logic presents an argument against the adoption of LPF. The main contribution of this paper (Section 5) is to pinpoint the issues that arise for the adaption of procedures such as resolution and proof by refutation to cope with LPF.

Structure of the paper: Section 2 provides an introduction to LPF. Section 3 briefly introduces a number of definitions and results for the mechanisation of classical logic, focusing on techniques such as the clausal form representation, resolution and proof by refutation. Section 4 provides a semantics for the LPF version of the Predicate Calculus. The rest of the paper is grounded on this semantic model. Section 5 outlines the issues present –and the changes required– to cover LPF by the techniques introduced for classical logic in Section 3. Finally, Section 6 provides some conclusions and an indication of further work.

2 An Introduction to LPF

LPF is a first order predicate logic designed to handle non-denoting logical values that can arise from terms that apply partial functions and operators. LPF is the logic that underlies the *Vienna Development Method (VDM)* [Jon90, BFL⁺94, Fit07]. The arguments that support the use of LPF are documented in several of the previously cited references, particularly [CJ91, Jon06, JL11]. There was also an instantiation of LPF on the *mural* [JJLM91] formal development support system.

The truth tables in Figure 1 (presented in [Kle52, §64]) illustrate the way in which the propositional operators in LPF have been extended to handle logical values that may fail to denote. These truth tables provide the strongest possible *monotonic* extension of the familiar classical propositional operators with respect to the following ordering on truth values: $\perp_{\mathbb{B}} \preceq \mathbf{true}$ and $\perp_{\mathbb{B}} \preceq \mathbf{false}$. The truth tables can be viewed as describing a parallel lazy evaluation of the operands, delivering a result as soon as enough information is available; such a result will not be contradicted if a $\perp_{\mathbb{B}}$ later evaluates to a proper Boolean value.

The way in which the propositional operators have been extended to handle logical values that may fail to denote is depicted in Figure 2, writing \mathbb{Z}_{\perp} (\mathbb{B}_{\perp}) for $\mathbb{Z} \cup \{\perp_{\mathbb{Z}}\}$ ($\mathbb{B} \cup \{\perp_{\mathbb{B}}\}$) respectively. Although the conditional operators

¹As explained in earlier papers, the problem of non-denoting terms is pervasive and most of these papers have used example formulae containing recursive functions; the fact that division is a partial operator (undefined with a zero divisor) is used, in this paper, to present the essential points with a minimum of extra machinery.

²McCarthy defined, for example, the disjunction of p, q as **if p then true else q** and referred to the first variable in such conditional expressions as the “inevitable variable” since the conditionals are strict in their first argument.

\vee	true	$\perp_{\mathbb{B}}$	false
true	true	true	true
$\perp_{\mathbb{B}}$	true	$\perp_{\mathbb{B}}$	$\perp_{\mathbb{B}}$
false	true	$\perp_{\mathbb{B}}$	false

\wedge	true	$\perp_{\mathbb{B}}$	false
true	true	$\perp_{\mathbb{B}}$	false
$\perp_{\mathbb{B}}$	$\perp_{\mathbb{B}}$	$\perp_{\mathbb{B}}$	false
false	false	false	false

\neg	
true	false
$\perp_{\mathbb{B}}$	$\perp_{\mathbb{B}}$
false	true

Figure 1: The LPF truth tables for disjunction, conjunction and negation.

$$\forall i: \mathbb{Z} \cdot \underbrace{\underbrace{(i \div i = 1)}_{\in \mathbb{Z}_{\perp}} \vee \underbrace{\underbrace{((i-1) \div (i-1) = 1)}_{\in \mathbb{Z}_{\perp}}}_{\in \mathbb{B}_{\perp}}}_{\in \mathbb{B}_{\perp}}$$

Figure 2: An illustration of where “undefinedness” can be “caught” with LPF.

in [McC67] fail to retain familiar properties like the commutativity of disjunctions and conjunctions, they broadly fit the picture depicted in Figure 2.

The quantifiers of LPF are a natural extension of the propositional operators — viewing existential quantification as an infinite disjunction (in the worst case) and universal quantification as an infinite conjunction. Thus, an existentially quantified expression in LPF is true if a witness value exists even if the quantified expression is undefined or false for some of the bound values. Such an expression is false if the quantified expression is everywhere false; it is undefined in the remaining case (a mixture of false and undefined). Similar comments apply, *mutatis mutandis*, for universally quantified expressions. In LPF quantification is only ever performed over a set of proper (i.e. defined) values.

One issue with the use of LPF is that the, so called, *law of the excluded middle* [Har09]

$$\overline{p \vee \neg p}$$

does not hold because the disjunction of two undefined Boolean values is still undefined: thus $(0 \div 0 = 1) \vee \neg(0 \div 0 = 1)$ is not a tautology in LPF.

LPF includes all of the propositional operators and quantifiers of classical logic. For expressive completeness, LPF adds a definedness operator Δ whose truth table is presented in Figure 3. Unlike all of the other operators presented, the Δ operator is not monotone. It also gives rise to an alternative property for LPF which is known as the *law of the excluded fourth*:

$$\overline{p \vee \neg p \vee \neg \Delta p}$$

that is, p is true, false or undefined. (This property is exploited to define a modified version of a refutation procedure for LPF in Section 5.4.) Adding definedness hypotheses for all terms in some logical expression p is sufficient to make the validity of p in LPF and classical logic coincide.

The normal notion of a proof is that one proceeds from assumptions and derives their consequences. A *sequent* $e_1, \dots, e_n \vdash e$ is used to represent the situation when the formula e can be logically derived from the assumptions e_1, \dots, e_n .

At any point in a proof, a line should be true if all of the assumptions are true. For this reason, “undefinedness” plays little part in LPF proofs. The only real intrusion is where one wants to use what is, in classical logic, the unrestricted deduction theorem:

$$\frac{p \vdash q}{p \Rightarrow q}$$

Δ	
true	true
$\perp_{\mathbb{B}}$	false
false	true

Figure 3: The LPF truth table for the definedness operator (Δ).

from	$\forall i: \mathbb{Z} \cdot i = 0 \Rightarrow \neg((i - 1) = 0); \forall i: \mathbb{Z} \cdot \neg(i = 0) \Rightarrow i \div i = 1$	
1	from $i: \mathbb{Z}$	
1.1	$i = 0 \vee \neg(i = 0)$	$h1, \mathbb{Z}$
1.2	from $i = 0$	
1.2.1	$\neg((i - 1) = 0)$	$\Rightarrow -E-L(\forall-E(h1, h), h1.2)$
1.2.2	$(i - 1) \div (i - 1) = 1$	$\Rightarrow -E-L(\forall-E(h1, h), 1.2.1)$
	infer $(i \div i = 1) \vee ((i - 1) \div (i - 1) = 1)$	$\vee-I-L(1.2.2)$
1.3	from $\neg(i = 0)$	
1.3.1	$i \div i = 1$	$\Rightarrow -E-L(\forall-E(h1, h), h1.3)$
	infer $(i \div i = 1) \vee ((i - 1) \div (i - 1) = 1)$	$\vee-I-R(1.3.1)$
	infer $(i \div i = 1) \vee ((i - 1) \div (i - 1) = 1)$	$\vee-E(1.1, 1.2, 1.3)$
infer	$\forall i: \mathbb{Z} \cdot (i \div i = 1) \vee ((i - 1) \div (i - 1) = 1)$	$\forall-I(1)$

Figure 4: An illustrative proof of Property 1 in LPF.

which does not hold in LPF because p could be an arbitrary (local) assumption that is potentially undefined. Admitting this form of the deduction rule effectively gives rise to the *law of the excluded middle*. The use of Δ provides a sound “ $\Rightarrow -I$ ” rule for LPF:

$$\boxed{\Rightarrow -I} \frac{\Delta p; p \vdash q}{p \Rightarrow q}$$

However, the non-monotone Δ operator is not normally used in assertions and is generally considered to be a meta-level operator; to claim definedness in a proof, the related δ operator can be used which is monotone and is defined as Δ except that $\delta \perp_{\mathbb{B}} = \perp_{\mathbb{B}}$ rather than false, thus δp is equivalent to the assertion $p \vee \neg p$. Therefore, the following “ $\Rightarrow -I$ ” rule for LPF

$$\boxed{\Rightarrow -I} \frac{\delta p; p \vdash q}{p \Rightarrow q}$$

is more common. In practice, there are normally trivial ways of showing definedness since typical implications have terms like $i \geq j$ on the left and its definedness follows immediately from the type $i, j: \mathbb{Z}$. (The observation about proof only leading to (defined and) true expressions is echoed when it is noted in Section 5.4 that “cancellation” in resolution is valid on clauses to the left of a turnstile.)

Anyone familiar with natural deduction proofs will find it straightforward to adapt to LPF. The axioms in [JM94] include extra rules such as $\neg \vee -I$ that ameliorate the loss of (but do not imply) the law of the excluded middle.

To conduct a proof of Property 1, it is necessary to introduce some properties of division and subtraction, since a proof is a game with symbols — it cannot use the semantics of the operators $-/\div$:

$$\forall i: \mathbb{Z} \cdot i = 0 \Rightarrow \neg((i - 1) = 0); \forall i: \mathbb{Z} \cdot \neg(i = 0) \Rightarrow i \div i = 1 \vdash \forall i: \mathbb{Z} \cdot (i \div i = 1) \vee ((i - 1) \div (i - 1) = 1)$$

The proof of this property in LPF is straightforward and, as can be seen in Figure 4, does not become complicated by “undefinedness” issues despite the fact that the example has been deliberately chosen so that either of the disjuncts could be undefined.

3 A Brief Background to Mechanising Classical Logic

This section briefly recalls a number of definitions and results from classical logic which are considered with respect to LPF in Section 5, where each is treated more formally. Thus, this section considers key techniques that arise in automated theorem proving, including: unification, resolution and proof by refutation. For a more detailed description, the reader is referred to [BA01] and [Har09].

In classical logic, a formula is an expression constructed using the standard logical connectives and quantifiers together with the variable, predicate and function symbols given by the context. Given a formula e , an *interpretation* σ is a mapping that associates a meaning to all variables, predicates and function symbols in e , thus allowing it to be evaluated to either true or false. A formula e is said to be *satisfiable* iff there exists an interpretation for which e evaluates

to true and unsatisfiable iff it is not satisfiable — that is, that there exists no interpretation for which e evaluates to true. Any interpretation that leads to a formula e being evaluated to true is known as a *model* of e . Two formulae which have the same set of models are said to be *logically equivalent*. A formula e is said to be *valid*, denoted $\models e$, iff e evaluates to true for every possible interpretation. The notation $\not\models e$ indicates that e is not valid. Two formulae e_1 and e_2 are *equi-satisfiable* when e_1 is satisfiable iff e_2 is satisfiable. Note that two equi-satisfiable formulae may have different models and thus may not be logically equivalent.

Let $\Gamma = \{e_1, \dots, e_n\}$ be a set of formulae then Γ is simultaneously satisfiable iff there exists an interpretation that satisfies each e_1, \dots, e_n . Let e be a single formula then e is a *logical consequence* of Γ , denoted $\Gamma \models e$, if every interpretation that makes e_1, \dots, e_n true also makes e true [BA01]. If $\Gamma = \{\}$, then logical consequence is the same as validity. Since Γ is finite, $\Gamma \models e$ is equivalent in classical logic to the assertion $\models e_1 \wedge \dots \wedge e_n \Rightarrow e$. The notation $\Gamma \not\models e$ is used when e is not a logical consequence of Γ .

3.1 Normal Forms

In order to mechanise and to help optimise decision procedures for classical logic, a range of normal forms and representations for formulae are used.

A propositional formula is said to be in *Conjunctive Normal Form (CNF)* iff it is a conjunction of disjunctions of literals, where a literal is an atomic formula (a positive literal) or the negation of an atomic formula (a negative literal). In classical logic, any propositional formula can be converted into a logically equivalent formula in CNF.

Any propositional formula in CNF can be represented using a logically equivalent set based notation called *clausal form*. A formula in clausal form is represented as a set of clauses $\{C_1, \dots, C_n\}$ which represent the conjunction $C_1 \wedge \dots \wedge C_n$, where each clause C_i is a set of literals $\{l_{i1}, \dots, l_{im_i}\}$ representing the disjunction $l_{i1} \vee \dots \vee l_{im_i}$. Clausal form is used extensively in automated theorem proving since it provides a concise and efficient representation for formulae.

A predicate formula is said to be in *Prenex Normal Form (PNF)* if all of its quantifiers occur on the outside in front of the rest of the formula (normally referred to as the *matrix*). So a formula in PNF is of the form $Q_1 x_1 \dots Q_n x_n \cdot e$ where each Q_i is either a universal or existential quantifier and e is the matrix. Again, in classical logic, any predicate formula can be converted into a logical equivalent formula in PNF.

The conversion of a closed predicate formula (that is, a formula with no *free variables*) into clausal form involves first converting the formula into PNF. The existential quantifiers are then removed through a procedure known as *Skolemisation*, where each existentially quantified variable is replaced with either a new constant (a *Skolem constant*) or a new function (a *Skolem function*). Note that this Skolemisation results in a formula which is equi-satisfiable to the original formula but not necessarily logically equivalent. The matrix is then to be converted into CNF and subsequently represented using clausal form, after removing the universal quantifiers, since the variables in each clause are implicitly universally quantified.

As an example the formula: $\forall x \cdot \forall y \cdot (P(x, y) \vee \exists z \cdot Q(y, z))$ in clausal form is: $\{\{P(x, y), Q(y, f(x, y))\}\}$, where f is a Skolem function of the corresponding universally quantified variables.

3.2 Unification

Unification [Rob65, BA01, Har09] is the process of finding, if it exists, a substitution (a map of variables to terms) for the variables in two terms that makes the terms identical. For example, applying unification to the terms $f(g(x), z)$ and $f(y, h(7))$ could result in the substitution $\phi = \{y \mapsto g(x), z \mapsto h(7)\}$ which gives:

$$\phi[f(g(x), z)] = f(g(x), h(7)) = \phi[f(y, h(7))]$$

where $\phi[\alpha]$ is the application of a substitution ϕ to a term α , which is the simultaneous replacement of each $\psi_i \in \mathbf{dom} \phi$ in α with the respective $\phi(\psi_i)$. If such a substitution exists then it is known as a *unifier* for the given terms.

Not all terms can be unified; for example, there is no unifier for the terms $f(x)$ and $g(y)$ where f and g are different function symbols. There is also no unifier for $f(x)$ and $f(g(x))$, since the variable x “occurs” within the larger term $g(x)$ (cf. the *occurs check*).

It is possible to have more than one unifier for two terms. However, if two terms can be unified, then they have what is known as a *most general unifier (mgu)* which is unique up to the renaming of variables. A unifier ϕ is an mgu iff any other unifier for the terms can be derived by composing ϕ with an appropriate substitution.

3.3 Resolution

Resolution [Rob65, BA01, Har09] is a widely used decision procedure for checking the satisfiability of a set of clauses. It works by finding clauses containing *contradictions* (i.e. literals of the form l and $\neg l$) and then *resolving* on these

clauses to derive new clause(s). This is formalised by the *resolution rule*: given two contradictory clauses C_1 and C_2 , where $\{l\} \subseteq C_1$ and $\{\neg l\} \subseteq C_2$, a resolvent clause can be derived:

$$(C_1 \setminus \{l\}) \cup (C_2 \setminus \{\neg l\})$$

The resolution rule maintains satisfiability: if the two “clashing clauses” are simultaneously satisfiable then so is the resolvent.

The resolution procedure works by taking a set of clauses and then repeatedly applying the resolution rule to derive new clauses, each time adding the new clause derived to the set of clauses so far accumulated. If the empty clause \square is derived, then the set of clauses is not satisfiable and if no more new clauses can be derived, then the set of clauses must be satisfiable.

For predicate logic, the *ground resolution* procedure is inefficient because the set of ground terms is unbounded. The *general resolution* procedure addresses this problem by using *unification* (see Section 3.2) to generate “clashing clauses” that can be resolved. Since the clauses can contain variables, the aim is to resolve on the most general forms of clauses. It is this general resolution procedure that is considered for predicate LPF in Section 5.3 onwards. So for instance if there exist two literals l_1 and $\neg l_2$ that can be unified, where $\{l_1\} \subseteq C_1$ and $\{\neg l_2\} \subseteq C_2$, then the two clauses C_1 and C_2 can be resolved. The resolvent will take the form of:

$$(\phi[C_1] \setminus \phi[\{l_1\}]) \cup (\phi[C_2] \setminus \phi[\{\neg l_2\}])$$

where ϕ is the mgu of l_1 and l_2 .

A procedure called *factoring* is often used alongside the resolution procedure to allow for unifiable literals that occur in a single clause to be merged. Numerous other heuristic techniques have been developed to improve the efficiency of the resolution procedure (e.g. to guide the search) but these are not considered in this paper (the interested reader is referred to [WCR64, Cor96]).

3.4 Refutation Procedure

In classical logic, a formula e is valid iff $\neg e$ is unsatisfiable. This well-known duality between validity and satisfiability forms the basis of a decision procedure for validity known as a *refutation procedure*. The idea is that in order to show that $\Gamma \models e$ is valid,³ it suffices to show that

$$e_1 \wedge \dots \wedge e_n \wedge \neg e \tag{2}$$

is unsatisfiable. Suppose Formula 2 is unsatisfiable; then this means in classical logic that every interpretation making e_1, \dots, e_n simultaneously true must make $\neg e$ false and therefore e true. However, if formula 2 is satisfiable then there must exist at least one interpretation that makes e_1, \dots, e_n simultaneously true and $\neg e$ true and therefore e false, so the logical consequence $\Gamma \models e$ cannot be valid ($\Gamma \not\models e$).

The above refutation procedure is normally implemented alongside resolution. Resolution is *refutationally complete*: if $\Gamma \models e$, then the resolution procedure will be able to derive the empty clause from the set of clauses $\Gamma \cup \{\neg e\}$. However, resolution may fail to terminate when given a satisfiable set of predicate logic clauses as input. Indeed, there is no (full) decision procedure for validity in the predicate calculus [BA01, §7.5].

4 Semantics of LPF

This section provides a semantics for the LPF version of the Predicate Calculus. The semantics that follows is a simplified version of what was originally presented in [JL11]. A concrete syntax for LPF –using *Extended Backus-Naur Form*– is presented in Figure 5. It is this concrete syntax that is used in the semantic definition of LPF, although when writing expressions that include references to functions/predicates such as $-$, \div and $=$ in examples, they are written using the standard infix notation for readability.

Only a subset of the logical operators is considered since all of the other logical operators can be defined in terms of this subset — just as in two-valued classical logic. Thus, as in classical logic, conjunctions (universal quantifiers) can be defined in terms of disjunctions (existential quantifiers) along with appropriate use of negations. Furthermore, $p \Rightarrow q$ is equivalent to $\neg p \vee q$ as in classical logic (and δp is equivalent to $p \vee \neg p$).

³The assumptions Γ could, for instance, be a consistent set of axioms that are assumed to be true independent of the theorem e that is to be proved.

```

EXPR = INEXPR | BOOLEXP
ID = (* The set of identifiers, see the explanation below. *);
INEXPR = ID | INTEGER | FUNAPP;
INTEGER = NUMBER | ["-"] NONZERO {NUMBER};
NUMBER = "0" | NONZERO;
NONZERO = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
FUNAPP = ID "(" [INEXPR {"," INEXPR}] "(";
BOOLEXP = ID | BOOLEAN | PRED | UNARY | BINARY | QUANT;
BOOLEAN = "true" | "false";
PRED = ID "(" [INEXPR {"," INEXPR}] "(";
UNARY = "(" UNARYOP BOOLEXP "(";
UNARYOP = "¬" | "Δ";
BINARY = "(" BOOLEXP BINARYOP BOOLEXP "(";
BINARYOP = "√";
QUANT = "(" QUANTIFIER ID "." BOOLEXP "(";
QUANTIFIER = "∃";

```

Figure 5: The concrete syntax of the language.

The Δ logical operator is presented in the following semantics as it is necessary to introduce it for the treatment of a modified *refutation procedure* in LPF in Section 5. Recall however that Δ tends not to be used in normal assertions and is considered to be an operator on the meta-level.

Context conditions for such a language are outlined in [JL11] and spelt out formally in [Lov10]. The context conditions ensure that the semantics only need be given for expressions which are well-formed.

Four sorts of identifiers can occur in expressions, those for propositions (*Prop*), for integer variables (*Var*), for functions (*Fn*) and for predicates (*Pred*):

$$Id = Prop \mid Var \mid Fn \mid Pred$$

Prop, *Var*, *Fn* and *Pred* are assumed to be disjoint sets. One of the functions of the context conditions is to ensure that identifiers are used appropriately. Furthermore, it is required that all integer variables are explicitly captured by quantifiers. The set Σ of all maps from identifiers to their values is defined as the union of four maps:

$$\Sigma = Prop \xrightarrow{m} \mathbb{B} \mid Var \xrightarrow{m} \mathbb{Z} \mid Fn \xrightarrow{m} Function \mid Pred \xrightarrow{m} Predicate$$

where the denotations of *Functions* and *Predicates* are relations:

$$Function = \mathcal{P}(\mathbb{Z}^* \times \mathbb{Z})$$

$$Predicate = \mathcal{P}(\mathbb{Z}^* \times \mathbb{B})$$

The map involving *Prop* can be partial in the sense that a propositional identifier can be absent from the domain of a specific map ($\sigma \in \Sigma$) to allow for the possibility of undefined propositional identifiers. However, the maps involving *Var*, *Fn* and *Pred* are total. Of course the function and predicate denotations themselves can be partial. All functions and predicates are considered to be strict, that is, if there is a “gap” in an argument then there is a “gap” in the result of applying function/predicate to that argument.

The semantic function \mathcal{E} yields a relation and is presented formally in Figure 6. The semantics provides a set theoretic definition of the values that are denoted by expressions. Here the “gaps” that arise from partial terms and propositional expressions are modelled by choosing *relations* as the space of *denotations*. This is in contrast to the use of partial functions as is classical in *denotational semantics* [Sto77]. The use of relations might suggest non-determinacy but Lemma 1 below establishes that all denotations are in fact single valued. The following notes should be enough to make the \mathcal{E} semantics readable.

- The only constant values in the language are the set of Boolean values and the set of integer values, thus $Value = \mathbb{B} \mid \mathbb{Z}$.
- There is no undefined value, instead the treatment of undefinedness is as “gaps” in the denotation.

$\mathcal{E} : Expr \rightarrow \mathcal{P}(\Sigma \times Value)$

$\mathcal{E}(e) \triangleq$

cases e **of**

$e \in Value \rightarrow \{(\sigma, e) \mid \sigma \in \Sigma\}$

$e \in Prop \rightarrow \{(\sigma, \sigma(e)) \mid \sigma \in \Sigma \wedge e \in \mathbf{dom} \sigma\}$

$e \in Var \rightarrow \{(\sigma, \sigma(e)) \mid \sigma \in \Sigma\}$

$f(al) \rightarrow \{(\sigma, r) \mid$
 $f \in (Fn \cup Pred) \wedge$
 $\sigma \in \Sigma \wedge$
 $\forall i: \mathbf{inds} \ al \cdot (\sigma, vl(i)) \in \mathcal{E}(al(i)) \wedge$
 $(vl, r) \in \sigma(f)\}$

$\neg p \rightarrow \{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}(p)\} \cup$
 $\{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(p)\}$

$\Delta p \rightarrow \{(\sigma, \mathbf{true}) \mid \sigma \in \mathbf{dom} \mathcal{E}(p)\} \cup$
 $\{(\sigma, \mathbf{false}) \mid \sigma \in (\Sigma \setminus \mathbf{dom} \mathcal{E}(p))\}$

$p \vee q \rightarrow \{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(p)\} \cup$
 $\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(q)\} \cup$
 $\{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}(p) \wedge (\sigma, \mathbf{false}) \in \mathcal{E}(q)\}$

$\exists x \cdot p \rightarrow \{(\sigma, \mathbf{true}) \mid$
 $\sigma \in \Sigma \wedge$
 $\mathbf{true} \in \mathbf{rng} \{(\sigma \uparrow \{x \mapsto i\} \mid i: \mathbb{Z}) \triangleleft \mathcal{E}(p)\} \cup$
 $\{(\sigma, \mathbf{false}) \mid$
 $\sigma \in \Sigma \wedge$
 $\mathbf{rng} \{(\sigma \uparrow \{x \mapsto i\} \mid i: \mathbb{Z}) \triangleleft \mathcal{E}(p)\} = \{\mathbf{false}\}\}$

end

Figure 6: The semantic function \mathcal{E} which defines the semantics of LPF.

- Functions/predicates have a fixed arity in any given σ and will always return the same result for any given argument(s) in a given σ .
- The way in which “gaps” can be handled can be seen clearly for disjunctions.
- For simplicity, all quantification is performed over the set of integers and, moreover, for LPF the quantified values range only over the set of proper (i.e. defined) integer values.
- The semantics for quantifiers ensures that “gaps” are handled by non-denoting propositional expressions being absent from the domain of \mathcal{E} .
- Note that $\mathbf{dom} \{(\sigma_1, v_1), (\sigma_2, v_2)\} = \{\sigma_1, \sigma_2\}$ and $\mathbf{rng} \{(\sigma_1, v_1), (\sigma_2, v_2)\} = \{v_1, v_2\}$.

(A paper by the current authors [JLS12] compares some of the main approaches to handling partial terms arising in logical formulae; it presents a similar semantic model to that presented in Figure 6 for each approach considered which illustrates where “undefinedness” can be caught/handled in each approach. The semantic models are used to compare and contrast the different approaches.)

It is useful to record that the definition of any relation $\mathcal{E}(e)$ is deterministic (or “functional”):

Lemma 1. For any expression e it follows that $(\sigma, v_1) \in \mathcal{E}(e) \wedge (\sigma, v_2) \in \mathcal{E}(e) \Rightarrow v_1 = v_2$.

Proof. This follows from the fact that there is exactly one rule for each type of expression construct. Even though the case for the disjunction operator is defined by the use of two set unions, the domains of the relations only overlap in the case of $\mathbf{true} \vee \mathbf{true}$ where the result is the same regardless. \square

To provide intuition for Property 1 the denotations of the (total) subtraction operator, the partial division operator and the (strict) equality relational operator can be defined as follows (where $\llbracket op \rrbracket(a, b)$ is defined to return the standard result of applying the operator op on the given operands a and b):

$$\begin{aligned}
\sigma(-) &= \{((a, b), \llbracket - \rrbracket(a, b)) \mid a, b: \mathbb{Z}\} \\
\sigma(\div) &= \{((a, b), \llbracket \div \rrbracket(a, b)) \mid a, b: \mathbb{Z} \wedge b \neq 0\} \\
\sigma(=) &= \{((a, b), \llbracket = \rrbracket(a, b)) \mid a, b: \mathbb{Z}\}
\end{aligned}$$

It is the division operator that has been chosen to introduce partial terms in the illustrative examples in this paper. However, it is important to realise that the hypotheses in the proof in Figure 4 do not actually constrain the denotation of division (nor that of subtraction or equality) to fit exactly these denotations; they have only been presented for illustrative purposes.

Making use of the \mathcal{E} semantic function, the notions of validity, satisfiability, unsatisfiability, logical consequence etc. are now defined more formally for LPF.

4.1 Validity and Satisfiability

Where e is a Boolean formula, *validity*, *satisfiability* and *unsatisfiability* for LPF can be defined as:

$$\begin{aligned}
\text{valid}(e) &\text{ iff } \forall \sigma: \Sigma \cdot (\sigma, \mathbf{true}) \in \mathcal{E}(e) \\
\text{satisfiable}(e) &\text{ iff } \exists \sigma: \Sigma \cdot (\sigma, \mathbf{true}) \in \mathcal{E}(e) \\
\text{unsatisfiable}(e) &\text{ iff } \neg \exists \sigma: \Sigma \cdot (\sigma, \mathbf{true}) \in \mathcal{E}(e)
\end{aligned}$$

(These quantifiers are safe in the sense that there is no “undefinedness” to complicate their semantics.)

A satisfiable interpretation for e is an interpretation σ such that $(\sigma, \mathbf{true}) \in \mathcal{E}$. A valid formula is also satisfiable; a formula is unsatisfiable iff it is not satisfiable; and a formula e is not valid ($\not\models e$) iff for some $\sigma \in \Sigma$ it is the case that $(\sigma, \mathbf{true}) \notin \mathcal{E}(e)$.

Since LPF can be thought of as passing the “gaps” from terms into the space of the logic, it is vital that the above formulation is used for unsatisfiability in LPF. Were, for example, *unsatisfiable*(e) defined as $\forall \sigma: \Sigma \cdot (\sigma, \mathbf{false}) \in \mathcal{E}(e)$, the set of unsatisfiable expressions would be smaller since an expression e not evaluating to true is not, in LPF, the same as it evaluating to false. In classical logic, the only possible outcomes are true and false but in LPF it is necessary to take a position on the “gaps”, that is, $\sigma \notin \mathbf{dom} \mathcal{E}(e)$.

The notion of *logical equivalence* can be defined in terms of the \mathcal{E} semantics. Two formulae (e_1 and e_2) are *logically equivalent* if they have the same truth value in every model. Thus e_1 and e_2 are logically equivalent iff for all $\sigma \in \Sigma$ it is the case that: if $(\sigma, v) \in \mathcal{E}(e_1)$ then $(\sigma, v) \in \mathcal{E}(e_2)$; and if $(\sigma, v) \in \mathcal{E}(e_2)$ then $(\sigma, v) \in \mathcal{E}(e_1)$. In the \mathcal{E} semantics it thus follows that e_1 and e_2 are logically equivalent iff $\mathcal{E}(e_1) = \mathcal{E}(e_2)$.

Furthermore, two formulae (e_1 and e_2) are *equi-satisfiable* when e_1 is satisfiable iff e_2 is satisfiable, that is, there is a model for e_1 , $(\sigma_1, \mathbf{true}) \in \mathcal{E}(e_1)$, iff there is a model for e_2 , $(\sigma_2, \mathbf{true}) \in \mathcal{E}(e_2)$.

As a simple example of reasoning with the \mathcal{E} semantics, consider the following lemma which shows that double negation can be eliminated in LPF:

Lemma 2. Any formula $\neg \neg p$ is logically equivalent to p .

Proof. By the definition of \mathcal{E} , $\mathcal{E}(\neg \neg p)$ expands to

$\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}(\neg p)\} \cup \{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(\neg p)\}$. By the definition of \mathcal{E} this further expands to $\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(p)\} \cup \{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}(p)\}$. This immediately reduces to $\mathcal{E}(p)$ as required. \square

4.2 Judgements

Let $\Gamma = \{e_1, \dots, e_n\}$ be a set of Boolean expressions and e a single Boolean formula. Then $\Gamma \models e$ records that e is a *logical consequence* of Γ .⁴ In terms of the \mathcal{E} semantics presented, this can be formalised through the following set definitions. Consider all of the interpretations that satisfy each of the hypotheses:

$$S = \{\sigma \mid \sigma: \Sigma \wedge (\sigma, \mathbf{true}) \in \mathcal{E}(e_1) \wedge \dots \wedge (\sigma, \mathbf{true}) \in \mathcal{E}(e_n)\}$$

then e should be satisfied in all of those interpretations:

$$S \subseteq \{\sigma \mid \sigma: \Sigma \wedge (\sigma, \mathbf{true}) \in \mathcal{E}(e)\}$$

⁴While in classical logic this is equivalent to the assertion $e_1 \wedge \dots \wedge e_n \Rightarrow e$, this does not hold in LPF, cf. $\Rightarrow -I$.

When Γ is empty $\models e$ is written and every σ must make e true since $S = \Sigma$.

In order to avoid writing unnecessarily long set comprehensions in Section 5, a more concise notation is adopted. For any Boolean expression $e \in Expr$, e^σ represents $(\sigma, \mathbf{true}) \in \mathcal{E}(e)$ and $e^{\bar{\sigma}}$ represents $(\sigma, \mathbf{false}) \in \mathcal{E}(e)$. Additionally, Γ^σ represents $e_1^\sigma \wedge \dots \wedge e_n^\sigma$. It can then be defined that $e^\Sigma = \{\sigma \mid \sigma: \Sigma \wedge e^\sigma\}$ and that $\Gamma^\Sigma = \{\sigma \mid \sigma: \Sigma \wedge \Gamma^\sigma\}$.

Thus, a Boolean formula e is:

- *valid* iff $e^\Sigma = \Sigma$;
- *satisfiable* iff $e^\Sigma \neq \{\}$, (i.e. e evaluates to true in at least one σ); and
- *unsatisfiable* iff $e^\Sigma = \{\}$, (i.e. e does not evaluate to true in any σ).

Logical consequence is now $\Gamma^\Sigma \subseteq e^\Sigma$.

5 Decision Procedures

Section 5.1 provides an indication of the issues involved in adapting the techniques described in Section 3 to cope with LPF; and Sections 5.2–5.4 examine formally each of the techniques introduced in Section 3 detailing any changes required –from the classical logic case– to cover LPF. All proofs in this section are with respect to the \mathcal{E} semantic function definition presented in Section 4.

5.1 Problems Created by Gaps

The succeeding sub-sections show that most of the procedures for classical logic work (sometimes with minor modifications) for LPF. However, refutation is more problematic and is treated in Section 5.4.

Because of the position that is taken on “gaps” in LPF, the law of the excluded middle does not hold. There is, however, an additional law that holds in LPF, namely the *law of the excluded fourth*: $p^\Sigma \cup (\neg p)^\Sigma \cup (\neg \Delta p)^\Sigma = \Sigma$.

Consider first the obvious approach for propositional logic of writing out truth tables – the number of rows required in a truth table increases for LPF. In terms of the \mathcal{E} semantics every assignment of values is a $\sigma \in \Sigma$. For propositional LPF, the truth table method would require checking 3^n instances of σ to check a formula for validity, as opposed to just 2^n for classical logic. It might not be immediately obvious why it is necessary to check the result when propositional identifiers fail to denote but consider an example like $\neg \Delta p \vdash \neg \Delta \neg p$.

It is immediately clear that there is a problem with refutation procedures in LPF. As an illustrative example consider the logical consequent $\models p \vee \neg p$: after negating the *goal* formula and converting it into clausal form by applying one of the de Morgan’s Laws and eliminating a resulting double negation, the set of clauses is $\{\{\neg p\}, \{p\}\}$. Performing resolution on this set would immediately result in the empty clause since the two clauses “clash” (they are both unit-clauses and the only literals “clash”). Thus using a refutation procedure without modification would lead to the conclusion that $\models p \vee \neg p$ is valid which is not the case in LPF, since for instance, p could be an undefined propositional identifier.

5.2 Normal Forms

This section outlines how to convert LPF formulae into *clausal form*. Propositional logic is considered first, followed by predicate logic. Occurrences of Δ in LPF require additional conversions to be able to convert a formula into clausal form. However, as mentioned earlier, Δ is not usually written in normal assertions, so the use of Δ is left until Section 5.4 when it is necessary to introduce it (in a restricted form) for the treatment of refutation procedures in LPF.

5.2.1 Propositional Logic

The process of converting a propositional formula into CNF is as follows:

- eliminate any propositional operators other than conjunction, disjunction and negation by applying the syntactic definitions presented in Section 4. For example, replace any $p \Rightarrow q$ with $\neg p \vee q$;
- use *de Morgan’s Laws* to force negations inwards, (cf. Lemma 3);
- eliminate all double negations, (cf. Lemma 2); and
- use the *distributive laws* to remove conjunctions within disjunctions.

A proof that illustrates the truth of one form of de Morgan's laws in LPF follows.

Lemma 3. Any formula $\neg(p \vee q)$ is logically equivalent to $(\neg p) \wedge (\neg q)$.

Proof. By the definition of \mathcal{E} , $\mathcal{E}(\neg(p \vee q))$ expands to

$$\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}(p \vee q)\} \cup \{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(p \vee q)\}.$$

By the definition of \mathcal{E} this further expands to

$$\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}(p) \wedge (\sigma, \mathbf{false}) \in \mathcal{E}(q)\} \cup \\ \{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(p)\} \cup \{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(q)\}.$$

By the definition of \mathcal{E} , $\mathcal{E}((\neg p) \wedge (\neg q))$ expands to

$$\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(\neg p) \wedge (\sigma, \mathbf{true}) \in \mathcal{E}(\neg q)\} \cup \\ \{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}(\neg p)\} \cup \{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}(\neg q)\}.$$

By the definition of \mathcal{E} this further expands to

$$\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}(p) \wedge (\sigma, \mathbf{false}) \in \mathcal{E}(q)\} \cup \\ \{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(p)\} \cup \{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(q)\}.$$

Thus $\mathcal{E}(\neg(p \vee q)) = \mathcal{E}((\neg p) \wedge (\neg q))$ as required. \square

All of the equivalences used when converting a formula into CNF hold in LPF so the conversion to CNF is unchanged providing no use of Δ is present. Additionally, every propositional formula in LPF can be converted into an equivalent formula that is in CNF. In the sequel, $Expr \setminus \Delta$ is used to denote the set of expressions constructed without the use of Δ .

Theorem 4. Every LPF propositional formula $e \in Expr \setminus \Delta$, can be converted into an equivalent formula that is in CNF.

Proof. This theorem follows immediately from the fact that all of the required conversions hold in LPF (see Lemmas 2 and 3 — other laws whose proofs are not presented in this paper are similar). \square

The conversion of a CNF formula into *clausal form* relies on idempotence and the commutativity of \wedge and \vee — these properties hold in LPF and the proof of one of these properties is presented here.

Lemma 5. Any formula $p \vee p$ is logically equivalent to p .

Proof. By the definition of \mathcal{E} , $\mathcal{E}(p \vee p)$ expands to

$$\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(p)\} \cup \{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(p)\} \cup \\ \{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}(p) \wedge (\sigma, \mathbf{false}) \in \mathcal{E}(p)\}.$$

By the definition of a set, the first two sets from the set union definition presented above are equivalent ($A \cup A = A$) and the third set additionally can be simplified. The resulting set union immediately reduces to $\mathcal{E}(p)$ as required. \square

Theorem 6. Every LPF propositional formula $e \in Expr \setminus \Delta$, can be converted into an equivalent clausal form.

Proof. This immediately follows from Theorem 4, Lemma 5, the other idempotent property ($\mathcal{E}(p \wedge p) = \mathcal{E}(p)$) as well as the fact that the commutativity of \vee and \wedge all hold in LPF which all follow by the definition of \mathcal{E} . \square

5.2.2 Predicate Logic

In order to derive the clausal form for expressions in predicate LPF, it is necessary to show that dropping quantifiers is valid. The process of converting a formula into PNF is as follows:

- *standardise the variables apart*, i.e. rename variables, where necessary, so that no two quantifiers bind the same variable name;
- push any negation operators inwards so that they only apply to atomic formulas, e.g. through the use of de Morgan's Laws and through conversions such as $\neg \exists x \cdot p$ to $\forall x \cdot \neg p$; and
- move any quantifiers out of the matrix, e.g. through conversions such as $p \vee \exists x \cdot q$ to $\exists x \cdot (p \vee q)$.

The above process remains valid in LPF; for example, a proof of the final property above is given.

Lemma 7. Let p be a formula that contains no free occurrences of the variable x . Then any formula $p \vee \exists x \cdot q$

is logically equivalent to $\exists x \cdot (p \vee q)$.

Proof. This proof assumes that all variables are standardised apart. Also remember that, in the \mathcal{E} semantics all quantification is performed only over the set of integers (\mathbb{Z}).

By the definition of \mathcal{E} , $\mathcal{E}(p \vee \exists x \cdot q)$ expands to
 $\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(p)\} \cup$
 $\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(\exists x \cdot q)\} \cup$
 $\{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}(p) \wedge (\sigma, \mathbf{false}) \in \mathcal{E}(\exists x \cdot q)\}.$

By the definition of \mathcal{E} this further expands to
 $\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(p)\} \cup$
 $\{(\sigma, \mathbf{true}) \mid \sigma \in \Sigma \wedge \mathbf{true} \in \mathbf{rng}(\{\sigma \dagger \{x \mapsto i\} \mid i: \mathbb{Z}\} \triangleleft \mathcal{E}(q))\} \cup$
 $\{(\sigma, \mathbf{false}) \mid \sigma \in \Sigma \wedge (\sigma, \mathbf{false}) \in \mathcal{E}(p) \wedge$
 $\mathbf{rng}\{\sigma \dagger \{x \mapsto i\} \mid i: \mathbb{Z}\} \triangleleft \mathcal{E}(q) = \{\mathbf{false}\}\}.$

By the definition of \mathcal{E} , $\mathcal{E}(\exists x \cdot (p \vee q))$ expands to
 $\{(\sigma, \mathbf{true}) \mid \sigma \in \Sigma \wedge \mathbf{true} \in \mathbf{rng}(\{\sigma \dagger \{x \mapsto i\} \mid i: \mathbb{Z}\} \triangleleft \mathcal{E}(p \vee q))\} \cup$
 $\{(\sigma, \mathbf{false}) \mid \sigma \in \Sigma \wedge \mathbf{rng}\{\sigma \dagger \{x \mapsto i\} \mid i: \mathbb{Z}\} \triangleleft \mathcal{E}(p \vee q) = \{\mathbf{false}\}\}.$

By the definition of \mathcal{E} this further expands to
 $\{(\sigma, \mathbf{true}) \mid \sigma \in \Sigma \wedge \mathbf{true} \in \mathbf{rng}(\{\sigma \dagger \{x \mapsto i\} \mid i: \mathbb{Z}\} \triangleleft \mathcal{E}(p))\} \cup$
 $\{(\sigma, \mathbf{true}) \mid \sigma \in \Sigma \wedge \mathbf{true} \in \mathbf{rng}(\{\sigma \dagger \{x \mapsto i\} \mid i: \mathbb{Z}\} \triangleleft \mathcal{E}(q))\} \cup$
 $\{(\sigma, \mathbf{false}) \mid \sigma \in \Sigma \wedge \mathbf{rng}\{\sigma \dagger \{x \mapsto i\} \mid i: \mathbb{Z}\} \triangleleft \mathcal{E}(p) = \{\mathbf{false}\} \wedge$
 $\mathbf{rng}\{\sigma \dagger \{x \mapsto i\} \mid i: \mathbb{Z}\} \triangleleft \mathcal{E}(q) = \{\mathbf{false}\}\}.$

Since the variables have first been standardised apart and by the assumption that x is not free in p , if p denotes **true** or **false** when p contains no reference to x , then quantifying over x causes no change in the result. Thus the two sets formed are equivalent as required. \square

Every first-order LPF formula can be converted into an equivalent formula that is in PNF.

Theorem 8. Every LPF formula $e \in Expr \setminus \Delta$, can be converted into an equivalent formula that is in PNF.

Proof. This follows since the conversions required for converting a classical logic formula into PNF all hold in LPF. The proofs of these conversions follow in a similar way to the proof of one of the conversions presented in Lemma 7 and the fact that here the renaming of variables has no effect on logical equivalence. \square

Skolemisation also carries over to LPF and furthermore, because satisfiability is being sought there is no question of any of the Skolem constants (0-ary functions)/functions introduced being partial; all Skolem functions introduced are total.

Theorem 9. Let S' be the formula formed by Skolemising the formula S , where it is assumed that every Skolem function introduced is a distinct function symbol not present in S . It must then follow that S and S' are equi-satisfiable.

Proof. Recall that quantification is only defined over the set of integer values (\mathbb{Z}). If S contains no existential quantifier then no change results from performing Skolemisation and the result follows immediately. In the case that S contains at least the one existential quantifier then there are two cases to consider:

1. If S is satisfiable then S' must be satisfiable: Suppose $\forall x \cdot \exists y \cdot P(x, y)^\sigma$, where $\sigma \in \Sigma$, then show that an interpretation $\sigma' \in \Sigma$ exists such that $\forall x \cdot P(x, f(x))^{\sigma'}$. The fact that σ is a model here is equivalent to saying that for every possible value for x there exists a value for y that causes $P(x, y)$ to evaluate to true. A function $f(x) = y$ can be chosen and the interpretation σ' can then be defined as $\sigma' = \sigma \dagger \{f \mapsto \beta\}$, where β is a *Function* object from each possible value for x to a corresponding result y . There may be many witness values for the existential quantifier, but for a function it requires restricting to the one such witness value (i.e. one specific value for y for each value of x), cf. the *Axiom of Choice* [Har09, §3.6]. It then follows that if S is satisfiable (with the existential quantifier) then S' is satisfiable since the $f(al)$ case of the \mathcal{E} semantic function can return a value that would otherwise have been produced by the existential quantifier case of the \mathcal{E} semantic function (a witness value), i.e. f can be set up so that it produces a *satisfying value* given the required values of any argument(s).
2. If S' is satisfiable then S must be satisfiable: In the example from case 1 if it is the case that $\forall x \cdot P(x, f(x))^{\sigma'}$ then σ' must have an interpretation for the Skolem function f . Therefore σ is also a model for $\forall x \cdot \exists y \cdot P(x, y)$

by taking $y = f(x)$ and so on. \square

As usual the matrix can now be put into CNF to arrive at the clausal form representation. The universal quantifiers can be omitted from the clausal form representation, as in LPF the clauses are still considered to be universally quantified sentences.

Theorem 10. Every closed LPF formula $e \in Expr \setminus \Delta$, can be converted into a formula that is in clausal form, such that the original formula and the formula in clausal form are equi-satisfiable.

Proof. This is an immediate consequence of Theorems 8 and 9 and, for the conversion of the matrix, Theorems 4 and 6. \square

In order to try to reduce the size of a resulting clausal form formula, the *absorption properties* can be used, whereby both $p \wedge (p \vee q)$ and $p \vee (p \wedge q)$ can be simplified to p . Notice that the following lemma covers more than idempotence.

Lemma 11. Any formula $p \wedge (p \vee q)$ and any formula $p \vee (p \wedge q)$ are both logically equivalent to p .

Proof. First consider the case of $p \wedge (p \vee q)$ being logically equivalent to p .

By the definition of \mathcal{E} , $\mathcal{E}(p \wedge (p \vee q))$ expands to

$$\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(p) \wedge (\sigma, \mathbf{true}) \in \mathcal{E}(p \vee q)\} \cup \\ \{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}(p)\} \cup \\ \{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}(p \vee q)\}.$$

By the definition of \mathcal{E} this further expands to

$$\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(p) \wedge (\sigma, \mathbf{true}) \in \mathcal{E}(p)\} \cup \\ \{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(p) \wedge (\sigma, \mathbf{true}) \in \mathcal{E}(q)\} \cup \\ \{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}(p)\} \cup \\ \{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}(p) \wedge (\sigma, \mathbf{false}) \in \mathcal{E}(q)\}.$$

The second set from the set union is a subset of the first set; similarly, the fourth set from the set union is a subset of the third set. The first set (after the trivial simplification) and the third set immediately match the expansion of $\mathcal{E}(p)$, which is: $\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(p)\} \cup \{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}(p)\}$ and this concludes the first case. The proof of the $p \vee (p \wedge q)$ being logically equivalent to p case is similar to the proof of the case presented above. \square

5.3 Resolution

The proofs in this section assume no use of Δ ; the addition of Δ is considered in Section 5.4.

The key property underlying resolution is the cancellation of contradictory information (literals) from clauses. In LPF (as in classical logic) an assertion p and its negation $\neg p$ cannot both be true in an interpretation.

Lemma 12. The set of clauses $\{\{p\}, \{\neg p\}\}$ cannot be true in an interpretation, i.e. there exists no $\sigma \in \Sigma$ such that $(p \wedge \neg p)^\sigma$.

Proof. By the definition of \mathcal{E} , $\mathcal{E}(p \wedge \neg p)$ expands to

$$\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(p) \wedge (\sigma, \mathbf{true}) \in \mathcal{E}(\neg p)\} \cup \dots$$

By the definition of \mathcal{E} this further expands to

$$\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(p) \wedge (\sigma, \mathbf{false}) \in \mathcal{E}(p)\} \cup \dots$$

By Lemma 1 it follows that p cannot be both true and false in any σ and therefore the set above is equivalent to $\{\}$, i.e. $p^\Sigma \cap (\neg p)^\Sigma = \{\}$. \square

Recall that, in classical logic, an inferred resolvent holds iff the two clauses resolved on are simultaneously satisfiable — this also applies to the LPF case. First consider the proof for the propositional case in LPF.

Theorem 13. Given two propositional clauses C_1 and C_2 which are simultaneously satisfiable in a σ , where $\{l\} \subseteq C_1$ and $\{\neg l\} \subseteq C_2$ and l is a literal, then the resolvent $C_3 = (C_1 \setminus \{l\}) \cup (C_2 \setminus \{\neg l\})$, is satisfiable in the same σ .

Proof. By assumption, C_1^σ and C_2^σ hold for some $\sigma \in \Sigma$. For an arbitrary satisfiable σ there are three cases to consider:

1. l^σ : By the definition of \mathcal{E} , it follows that both C_1^σ and $\neg l^\sigma$. Since by assumption it is known that C_2^σ there must exist another disjunct (literal, $\neg l \neq l'$) $\{l'\} \subseteq C_2$ that satisfies C_2 , i.e. $(C_2 \setminus \{\neg l\})^\sigma$. Thus C_3 is satisfied (C_3^σ) by the definition of \mathcal{E} since $\{l'\} \subseteq C_3$ and l'^σ .

2. \bar{l}^σ : This follows by a similar argument to case 1, as there must exist another disjunct $(l \neq l') \{l'\} \subseteq C_1$ that satisfies C_1 , i.e. $(C_1 \setminus \{l\})^\sigma$ and C_2^σ holds because $\neg l^\sigma$. Thus C_3 is satisfied (C_3^σ) by the definition of \mathcal{E} since $\{l'\} \subseteq C_3$ and l'^σ .
3. $\sigma \notin \mathbf{dom} \mathcal{E}(l)$: By the definition of \mathcal{E} , it also follows that $\sigma \notin \mathbf{dom} \mathcal{E}(\neg l)$. Thus another disjunct $(\{l'\} \subseteq C_1)$ must satisfy C_1 , i.e. $(C_1 \setminus \{l\})^\sigma$ and another disjunct $(\{l''\} \subseteq C_2)$ must satisfy C_2 , i.e. $(C_2 \setminus \{\neg l\})^\sigma$, where $l \neq l'$ and $\neg l \neq l''$. Thus C_3 is satisfied (C_3^σ) by the definition of \mathcal{E} since $\{l', l''\} \subseteq C_3$ and l'^σ and l''^σ .

Thus, in all cases for an arbitrary σ where both C_1^σ and C_2^σ hold, it is the case that C_3^σ . The proof of the converse follows by a similar argument. \square

Now the predicate case is considered for LPF which, as mentioned earlier, uses unification.

Corollary 14. Given two clauses C_1 and C_2 which are simultaneously satisfiable in a σ , where $\{l_1\} \subseteq C_1$ and $\{\neg l_2\} \subseteq C_2$ and both l_1 and $\neg l_2$ are literals which can be unified by an mgu ϕ , then a resolvent $C_3 = (\phi[C_1] \setminus \phi[\{l_1\}]) \cup (\phi[C_2] \setminus \phi[\{\neg l_2\}])$, is satisfiable in the same σ .

Proof. By assumption, C_1^σ and C_2^σ hold for some $\sigma \in \Sigma$. Since ϕ can make the two literals l_1 and l_2 identical (l'), i.e. $l' = \phi[l_1] = \phi[l_2]$, it cannot be the case that both l' and $\neg l'$ are true in any $\sigma \in \Sigma$ by Lemma 12. The result then follows in a similar way to Theorem 13, since satisfiability is being sought. \square

Factoring also carries over to LPF.

(When using the resolution decision procedure for satisfiability in a refutation procedure, the use of unification needs restricting since validity is now being sought. A refutation procedure for LPF is considered in Section 5.4.)

If a resolvent is ever the empty clause then the set of clauses must have been unsatisfiable, i.e. there must be a contradiction. However, if the empty clause cannot be inferred and no more *new* resolvents can be inferred, then the set of clauses must have been satisfiable.

Theorem 15. If the empty clause is ever inferred by resolution on the set of clauses S , then S must be unsatisfiable.

Proof. The argument is by induction on the number of resolution steps. The fewest number of clauses that can be used to infer the empty clause is two where the only literal in each clause is identical (same propositional variable or they unify), only the literal is positive in the one clause and negative in the other. By Lemma 12, it follows that both of these clauses cannot be true –the set of clauses is unsatisfiable– and thus the empty clause is inferred.

If after $k + 1$ resolution steps S is unsatisfiable, then after k resolution steps S (without the $k + 1$ resolvent) must have been unsatisfiable. After each resolution step only contradictory literals are removed since they cannot both be true by Lemma 12. It is also known as a corollary of Theorem 13 and of Corollary 14 that if S is satisfiable, then the set of clauses S' formed after performing the one resolution step on S must be satisfiable. Thus if the empty clause is ever inferred, it must follow that the original set of clauses must have been unsatisfiable. \square

5.4 Refutation Procedure

The application of refutation procedures in LPF is complicated by the presence of “gaps” in denotations which affect the duality between validity and satisfiability. In LPF, if $\neg e$ is satisfiable then e cannot be valid, but if $\neg e$ is unsatisfiable then it is not possible to infer that e is valid. The following results clarify the relationship between satisfiability and validity in LPF.

Lemma 16. In LPF, if e is valid then $\neg e$ is unsatisfiable.

Proof. By the definition of validity, it is known that e is valid in LPF iff $e^\Sigma = \Sigma$ and by the definition of unsatisfiability, that e is unsatisfiable iff $e^\Sigma = \{\}$. By assumption it is the case that e^σ for each $\sigma \in \Sigma$. By the definition of \mathcal{E} , if e^σ then $\neg e^\sigma$ and since the truth value false is an unsatisfiable value the result is concluded as required. \square

Lemma 17. In LPF, if $\neg e$ is unsatisfiable then e may not be valid.

Proof. This result is due to the presence of “gaps” in LPF and can be shown using a simple counter example. Consider the Boolean formula $p \vee \neg p$ and its negation $\neg(p \vee \neg p)$ which is unsatisfiable, i.e. $\neg(p \vee \neg p)^\Sigma = \{\}$. However, $p \vee \neg p$ is not valid in LPF since any interpretation $\sigma \in \Sigma$ which has a “gap” for p , that is $\sigma \notin \mathbf{dom} \mathcal{E}(p)$, results in a “gap” for $p \vee \neg p$ ($\sigma \notin \mathbf{dom} \mathcal{E}(p \vee \neg p)$) and so $(p \vee \neg p)^\Sigma \subset \Sigma$. \square

5.4.1 Logical Consequence

It is therefore interesting to consider how the results of applying a refutation procedure to a logical consequence statement can be interpreted in LPF (assuming that a well-defined decision procedure for satisfiability is available for LPF). Consider the logical consequence $\Gamma \models e$ in LPF, where $\Gamma = \{e_1, \dots, e_n\}$:

1. Suppose $e_1 \wedge \dots \wedge e_n \wedge \neg e$ is shown to be satisfiable. Then there must exist at least one interpretation that makes all the expressions $e_1, \dots, e_n, \neg e$ true and so $\Gamma \models e$ cannot hold.
2. Suppose $e_1 \wedge \dots \wedge e_n \wedge \neg e$ is shown to be unsatisfiable. Then there does not exist an interpretation that makes all the expressions $e_1, \dots, e_n, \neg e$ true. Further information is now needed to be able to use this fact to make a judgement about the logical consequence. If it can be shown that e is defined in every interpretation that makes Γ true, that is, $\Gamma \models \Delta e$, then it can be inferred that $\Gamma \models e$ holds.

Resolution is the satisfiability decision procedure that is used in a refutation procedure in this paper. While a refutation procedure for classical logic need only consider refuting the set of clauses $\Gamma \cup \{\neg e\}$, in LPF the situation is more involved as the case that e denotes a “gap” also needs refuting. In other words, if $\Gamma \cup \{\neg e\}$ is unsatisfiable then it is necessary to ensure that $\Gamma \models \Delta e$ holds in order to be able to conclude that $\Gamma \models e$ holds. An appropriately extended refutation procedure can be used to check the logical consequent $\Gamma \models \Delta e$ (recall the *law of the excluded fourth*). Notice that no circularity is introduced because Δe is guaranteed to always return either true or false. The use of the meta-level operator Δ is banned from any formula in Γ and from the formula e ; Δ is only to be used when it is introduced around e for a refutation procedure to refute the “gap” case. The following results formalise the above discussion.

Theorem 18. If $\Gamma \cup \{\neg e\}$ is satisfiable then $\Gamma \not\models e$.

Proof. By assumption $e_1 \wedge \dots \wedge e_n \wedge \neg e$ is satisfiable and so it must be the case that $(e_1 \wedge \dots \wedge e_n \wedge \neg e)^\sigma$, for some interpretation $\sigma \in \Sigma$. Therefore it follows by the definition of \mathcal{E} that $e_1^\sigma, \dots, e_n^\sigma$ and $\neg e^\sigma$ and thus e^σ . Thus, there is an interpretation $\sigma \in \Gamma^\Sigma$ that makes the expression e evaluate to false and therefore $\Gamma^\Sigma \not\subseteq e^\Sigma$. By the definition of logical consequence it follows that $\Gamma \not\models e$. \square

Lemma 19. If $\Gamma \cup \{\neg e\}$ is unsatisfiable then $\Gamma \models e$ may not hold.

Proof. Consider a counter example that illustrates that if $\Gamma \cup \{\neg e\}$ is unsatisfiable then $\Gamma \models e$ does not hold. Given the logical consequence $\models p \vee \neg p$, from which –by Lemma 17– it follows that $(p \vee \neg p)^\Sigma \subseteq \Sigma$; but if $\Gamma = \{\}$ then by the definition of logical consequence $\Gamma^\Sigma = \Sigma$. \square

Theorem 20. If $\Gamma \cup \{\neg e\}$ is unsatisfiable and $\Gamma \models \Delta e$, then $\Gamma \models e$.

Proof. By assumption $\Gamma \cup \{\neg e\}$ is unsatisfiable and so it follows that $(\Gamma \cup \{\neg e\})^\Sigma = \{\}$. This means that for any interpretation $\sigma \in \Gamma^\Sigma$ we have either: (1) $\neg e^\sigma$; or (2) $\sigma \notin \text{dom } \mathcal{E}(\neg e)$. Now, by the assumption $\Gamma \models \Delta e$, it follows that $\Gamma^\Sigma \subseteq (\Delta e)^\Sigma$. Therefore, $(\Delta e)^\sigma$ holds for any interpretation $\sigma \in \Gamma^\Sigma$ and so by the definition of \mathcal{E} it follows that $\sigma \in \text{dom } \mathcal{E}(\neg e)$ holds. Thus only possibility (1) from above pertains for any $\sigma \in \Gamma^\Sigma$ and so by the definition of \mathcal{E} it follows that e^σ . Therefore, $\Gamma^\Sigma \subseteq e^\Sigma$ and so by the definition of logical consequence it follows that $\Gamma \models e$. \square

It is clear from Lemma 19 that as well as refuting the false case as in two-valued classical logic, in LPF the undefined (“gap”) case also needs refuting. If unsatisfiable is returned by applying the resolution procedure for satisfiability on $\Gamma \cup \{\neg e\}$, then the undefined “gap” case needs refuting. In order to show that $\Gamma \models \Delta e$ holds, one approach is to apply resolution on the set of clauses $\Gamma \cup \{\neg \Delta e\}$. If unsatisfiable is returned from this proof when refuting that e is undefined then validity ($\Gamma \models e$) can be concluded according to Theorem 20. If satisfiable is returned from this proof when refuting that e is undefined, then $\Gamma \not\models \Delta e$ and thus $\Gamma \not\models e$ must be concluded.

The following subsections consider extending using resolution in a refutation procedure to take into account the additional case needed for LPF. The introduction of the necessary Δ logical operator into the clausal form representation of Section 5.2 is considered first.

5.4.2 Introducing Δ into the Clausal Form

The use of Δ can lead to larger clausal form formulae, but fortunately the use of Δ is restricted to being introduced around the goal of a logical consequent statement for a refutation procedure. Any Δ that does occur needs pushing inwards so that eventually any Δ in a formula that is in CNF/clausal form will only surround an atom. Thus in LPF, what is meant by a literal is extended to also include Δl and $\neg \Delta l$ as literals, where l is an literal. Pushing a Δ operator inwards is first discussed for CNF and then for PNF.

CNF: Each Δ operator used in an LPF formula must be pushed inwards so that a Δ surrounds only atoms. Notice that it is tempting to define $\Delta(p \vee q)$ as $p \vee q \vee (\neg p \wedge \neg q)$ but this formula is not two-valued and Δ is a two-valued operator, and logical equivalence is being sought. So it is necessary to write:

- $\Delta(p \vee q)$ is logically equivalent to: $\neg((\neg p \wedge \neg \Delta q) \vee (\neg \Delta p \wedge \neg q) \vee (\neg \Delta p \wedge \neg \Delta q))$, (i.e. the negation of the three cases that make $\Delta(p \vee q)$ denote false), which converted into CNF is: $(p \vee \Delta q) \wedge (\Delta p \vee q) \wedge (\Delta p \vee \Delta q)$; and
- $\Delta(p \wedge q)$ is logically equivalent to the CNF formula: $(\neg p \vee \Delta q) \wedge (\Delta p \vee \neg q) \wedge (\Delta p \vee \Delta q)$.

Given a formula such as $\neg \Delta(p \vee q)$ the Δ should be pushed inwards first and then the negation can be pushed inwards. For instance, the formula $\neg \Delta(p \vee q)$ is first converted to: $\neg((p \vee \Delta q) \wedge (\Delta p \vee q) \wedge (\Delta p \vee \Delta q))$, which is then converted into the CNF formula: $(\neg p \vee \neg \Delta p) \wedge (\neg q \vee \neg \Delta q) \wedge (\neg \Delta p \vee \neg \Delta q)$.

It is also the case that $\Delta \neg l$ can be simplified to Δl and this case is handled by the following lemma.

Lemma 21. Any formula $\Delta \neg l$ is logically equivalent to Δl .

Proof. By the definition of \mathcal{E} , $\mathcal{E}(\Delta \neg l)$ expands to $\{(\sigma, \mathbf{true}) \mid \sigma \in \mathbf{dom} \mathcal{E}(\neg l)\} \cup \{(\sigma, \mathbf{false}) \mid \sigma \in (\Sigma \setminus \mathbf{dom} \mathcal{E}(\neg l))\}$.

By the definition of \mathcal{E} this further expands to $\{(\sigma, \mathbf{true}) \mid \sigma \in \mathbf{dom} \mathcal{E}(l)\} \cup \{(\sigma, \mathbf{false}) \mid \sigma \in (\Sigma \setminus \mathbf{dom} \mathcal{E}(l))\}$, which is equivalent according to the definition of \mathcal{E} to $\mathcal{E}(\Delta l)$ as required. \square

Also note that $p \vee \neg p \vee \neg \Delta p$ is equivalent to the truth value true, (cf. the law of the excluded fourth), and the formula $\Delta(\Delta p)$ is also equivalent to the truth value true. All of these conversions provided for Δ maintain logical equivalence.

PNF: Additionally, when considering the predicate LPF, the process outlined in Section 5.2 for converting a classical logic formula into PNF needs extending, since any Δ needs pushing into the matrix, before the CNF conversions can be used as normal on the matrix. Given Δe , where e is a quantified formula, then e should first be put into PNF and then the Δ can be pushed inwards using the rules that follow (then any negation that is to the left of the Δ can be dealt with). The following conversions are required:

- $\Delta(\forall i \cdot p(i))$ is logically equivalent to: $\neg(\exists i \cdot \neg \Delta p(i) \wedge \forall i \cdot (p(i) \vee \neg \Delta p(i)))$, (i.e. the negation of the cases that make $\Delta(\forall i \cdot p(i))$ false, which is when $p(i)$ is always undefined, or when $p(i)$ is true at least once, undefined at least once and is always true or undefined), which gives rise to: $\forall i \cdot \Delta p(i) \vee \exists i \cdot (\neg p(i) \wedge \Delta p(i))$; and
- $\Delta(\exists i \cdot p(i))$ is logically equivalent to: $\neg(\exists i \cdot \neg \Delta p(i) \wedge \forall i \cdot (\neg p(i) \vee \neg \Delta p(i)))$.

Again, note that $\Delta(\forall/\exists i \cdot p(i))$ is two-valued, and thus the formulations above are needed since logical equivalence is being sought. The formula $\Delta(\forall i \cdot p(i))$ is represented in clausal form as: $\{\{\neg p(c), \Delta p(x)\}, \{\Delta p(c), \Delta p(x)\}\}$, while the formula $\neg \Delta(\forall i \cdot p(i))$ is represented in clausal form as: $\{\{p(x), \neg \Delta p(x)\}, \{\neg \Delta p(c)\}\}$, where c is a Skolem constant.

An optimisation to this treatment is considered next since Δ is only introduced around the goal formula of a logical consequent statement in a particular circumstance.

Optimisation: So far, a Δ surrounding a quantifier is replaced with two quantifiers whereby logical equivalence is maintained and any occurrence of the Δ operator is now inside the quantifiers. However, this approach leads to an expensive clausal form. The objective when considering only universal quantifiers is to reduce the resulting clausal form size, (what follows does not apply to existential quantifiers).

When trying to show $\Gamma \models e$, the aim is to show that e is true and defined. If unsatisfiable is returned from the resolution proof on the set of clauses $\Gamma \cup \{\neg e\}$, then it can only be the case that e is either true if defined or undefined. A resolution proof on the set of clauses $\Gamma \cup \{\neg \Delta e\}$ would then follow and in this proof it is known that e cannot be false, otherwise satisfiable would have been returned from the first proof. This second proof is to show definedness by refuting that it is undefined and recall that Δ can only return true or false.

Consider that the goal e is $\forall i \cdot p(i)$. While $\Delta(\forall i \cdot p(i))$ is not logically equivalent to $\forall i \cdot \Delta p(i)$ –consider the case that $p(i)$ is false at least once and undefined at least once– they are logically equivalent in the restricted case when $\forall i \cdot p(i)$ is not false. The clausal form of $\neg \Delta(\forall i \cdot p(i))$ would now be $\{\{\neg \Delta p(c)\}\}$.

5.4.3 Refuting the Possibility of a “Gap”

Following on from the discussion in Section 5.4.1 and from Theorem 20, if unsatisfiable is returned from applying resolution on the set of clauses $\Gamma \cup \{\neg e\}$, then $\Gamma \models \Delta e$ needs to be shown to hold in order to conclude $\Gamma \models e$. To show that $\Gamma \models \Delta e$ holds, the approach taken here is to refute the undefined (“gap”) case by performing resolution on the set of clauses $\Gamma \cup \{\neg \Delta e\}$. This leads to extra “resolvent” possibilities that arise in LPF, which include allowing for resolving on p and $\neg \Delta p$, and on $\neg p$ and $\neg \Delta p$. The following result shows that the Δ “resolvent” possibilities are sound.

Lemma 22. The literal pairs p and $\neg \Delta p$, and $\neg p$ and $\neg \Delta p$ are contradictory and their simultaneous satisfaction is impossible.

Proof. The goal is to show that $p^\Sigma \cap (\neg \Delta p)^\Sigma = \{\}$ and $(\neg p)^\Sigma \cap (\neg \Delta p)^\Sigma = \{\}$. Consider an arbitrary $\sigma \in \Sigma$:

- if $\sigma \in \mathbf{dom} \mathcal{E}(p)$ holds (then also $\sigma \in \mathbf{dom} \mathcal{E}(\neg p)$ holds) then by the definition of \mathcal{E} it follows that $(\Delta p)^\sigma$ (and $(\Delta \neg p)^\sigma$ which is equivalent to $(\Delta p)^\sigma$ by Lemma 21); and
- if $\sigma \notin \mathbf{dom} \mathcal{E}(p)$ holds then by the definition of \mathcal{E} it follows that $(\Delta p)^\sigma$ and thus $(\neg \Delta p)^\sigma$.

Thus these literal pairs are contradictory and therefore no $\sigma \in \Sigma$ can simultaneously satisfy both p and $\neg \Delta p$ nor both $\neg p$ and $\neg \Delta p$. \square

Lemma 22 establishes that in LPF further contradictory literals exist that can be used by the resolution rule. The use of these “extra” resolvent possibilities provides a way of refuting “undefinedness” by applying resolution on the set of clauses $\Gamma \cup \{\neg \Delta e\}$.

Recall that in this paper only strict functions as well as strict predicates are being considered. Surrounding the goal with Δ , however, is not enough to show definedness as can be illustrated by the following simple example:

$$\forall x \cdot x = x \models 5 \div 0 = 5 \div 0$$

Performing standard resolution and unification (as part of a refutation procedure) on the clausal form of this logical consequent leads to the empty clause. Surrounding the goal with Δ and performing a new resolution proof (again as part of a refutation procedure) to refute the presence of a “gap” this time, again leads to the empty clause by using an “extra” resolvent possibility. But clearly this formula is not valid in LPF: consider the counter example of a weak equality predicate and $5 \div 0$ denoting a “gap”; a defined term x has been unified with a term $5 \div 0$ that *can* be undefined. (The term x must be defined because it is a quantified variable and quantification can only be over a set of proper values in LPF.) Therefore the application of unification within a resolution step in LPF (when considering validity — a refutation procedure) due to the presence of “gaps” needs guarding in certain circumstances.

The approach taken is for constraint(s) to be included as literal(s) in an inferred resolvent when using resolution as part of a refutation procedure. These constraints effectively take the form of further *well-definedness conditions*. A resolvent inferred by resolving on the clauses C_1 and C_2 where $\{l_1\} \subseteq C_1$ and $\{\neg l_2\} \subseteq C_2$ is defined to be:

$$(\phi[C_1] \setminus \phi[\{l_1\}]) \cup (\phi[C_2] \setminus \phi[\{\neg l_2\}]) \cup \theta$$

where l_1 and l_2 unify with an mgu ϕ and where θ is a set of unification constraint(s), where each $\psi_i \in \theta$ is a literal. This form of the resolvent is needed whenever a clause from the right (*goal*) side (containing the potentially undefined term — a function) is resolved (and thus unified) with a clause from the left (*assumption*) side of the logical consequent. Any resolvent that is inferred when at least one of the two clauses resolved on is a goal clause is deemed to be a goal clause for the purposes of introducing unification constraint(s).

The unification constraints θ can be built up by considering an mgu ϕ , where given $\phi = \{x_1 \mapsto \alpha_1, \dots, x_n \mapsto \alpha_n\}$, then $\theta = \{\neg(\alpha_1 \in \mathbb{Z}), \dots, \neg(\alpha_n \in \mathbb{Z})\}$, where each $\psi_i \in \theta$ is a literal (a disjunct). For instance, if $\phi = \{x \mapsto f(\dots)\}$, where $x \in \mathit{Var}$ and $f \in \mathit{Fn}$, then $\theta = \{\neg(f(\dots) \in \mathbb{Z})\}$.

The operator \in is defined to be a *BOOLEXP*, with the concrete syntax of: *DEF* = “(“ *INTEXP* “ \in ” \mathbb{Z} “)”;. In \mathcal{E} , $\alpha \in \mathbb{Z}$ is defined as: $\alpha \in \mathbb{Z} \rightarrow \{(\sigma, \mathbf{true}) \mid \sigma \in \mathbf{dom} \mathcal{E}(\alpha)\}$. (Note that the \in operator is monotone, and that $\alpha \in \mathbb{Z}$ is true if the integer operand α is defined, otherwise it is undefined.) A literal of the form $\alpha \in \mathbb{Z}$ is to be included only as a positive literal on the assumption side Γ of a logical consequent statement, to state that a function α is defined. This treatment of adding unification constraints into the resolvent for every maplet in ϕ can, however, be improved upon.

First consider that the function identifiers Fn can be seen as a shorthand for $\mathit{Fn} = \mathit{SkolemFun} \mid \mathit{Fun}$, where the identifier names in *SkolemFun* and *Fun* are disjoint. This split is illustrated explicitly for the purposes of including the unification constraints because a Skolem function can be shown to be total, but a *Function* mapped to by any *Fun* can be a partial function.

1	$\forall i: \mathbb{Z} \cdot i = 0 \Rightarrow \neg((i - 1) = 0)$	<i>assumption</i>
2	$\forall i: \mathbb{Z} \cdot \neg(i = 0) \Rightarrow i \div i = 1$	<i>assumption</i>
3	$\forall i: \mathbb{Z} \cdot (i \div i = 1) \vee ((i - 1) \div (i - 1) = 1)$	<i>goal</i>
4	$\{\neg(i = 0), \neg((i - 1) = 0)\}$	<i>clausal_form(1)</i>
5	$\{i = 0, i \div i = 1\}$	<i>clausal_form(2)</i>
6	$\{\neg(c \div c = 1)\}$	
7	$\{\neg((c - 1) \div (c - 1) = 1)\}$	<i>}deny(clausal_form(3))</i>
8	$\{c = 0\}$	<i>resolve(5, 6)</i>
9	$\{(c - 1) = 0\}$	<i>resolve(5, 7)</i>
10	$\{\neg((c - 1) = 0)\}$	<i>resolve(4, 8)</i>
11	\square	<i>resolve(9, 10)</i>

Figure 7: An illustrative proof of Property 1 using resolution as part of a refutation procedure.

Therefore in certain circumstances the use of unification requires no additional constraints to be included into a resolvent, for instance, when unifying x with y when $x \in Var$ and $y \in Var$ and when unifying x with $f(\dots)$ when $x \in Var$ and $f \in SkolemFun$. However, when unifying x with $f(\dots)$ when $x \in Var$ and $f \in Fun$ then a constraint must be introduced into any inferred resolvent arising from a resolution step. (Notice that a predicate in the semantic model considered in this paper cannot be unified with any variable, as only integer and propositional variables are being considered.)

The reason behind including the unification constraints in a resolvent is that in \mathcal{E} , for any $f \in Fun$, it can only be known that $f(\dots) \in \mathbb{Z}_\perp$. If this term is unified with any $x \in Var$, when it is known that all integer variables (Var) are defined in \mathcal{E} , then unification within a resolution step that allows for something that is always defined to be unified and thus resolved with something that *can* be undefined from the goal side violates a condition that needs to hold in order that the result indicated in Theorem 20 follows.

As can be seen in an illustrative example in Section 5.4.4 a unification constraint can be removed (only if it is known to be defined) by a further resolution step.

5.4.4 Returning to the Illustrative Examples

Consider again the earlier counter example of $\models p \vee \neg p$ where resolution as part of a refutation procedure infers the empty clause (unsatisfiability). Therefore in LPF $\models \Delta(p \vee \neg p)$ needs to be shown to hold to be able to infer that $\models p \vee \neg p$ holds. In the modified LPF clausal form the negation $\neg \Delta(p \vee \neg p)$ is represented as $\{\{\neg \Delta p\}\}$ after simplification, which cannot be refuted and therefore this example is satisfiable and the result $\not\models p \vee \neg p$ is inferred.

Returning to the example presented in Property 1, an example proof of this property using a refutation procedure is presented in Figure 7, where c in this proof is a Skolem constant. This proof makes use of resolution, where unification is used in a resolution step as needed. Figure 8 presents the same proof but also establishes the definedness of the goal. (Note that an additional assumption is required for the latter proof, but this assumption states that only interpretations where subtraction is defined as a total function are to be considered.)

As can be seen from the two proofs the second proof (with Δ) has a longer clausal form. Additionally, the number of resolvents inferred *en route* to the empty clause increases and the size of the search space as expected also increases. An optimisation is considered next.

5.5 Optimisation

This optimisation considers reducing the number of cases in which Δ needs to be introduced around the goal. This optimisation does not concern reducing the introduction of any unification constraints, which still need to be introduced whenever one of the circumstances mentioned earlier arises.

By the definition of logical consequence, it follows that $\Gamma^\Sigma \subseteq e^\Sigma$ and thus cancellation of anything from the *goal* side in resolution, with anything from the *assumption* side of the logical consequent, is safe. This follows from the fact that, when considering using a refutation procedure with resolution, only those $\sigma \in \Gamma^\Sigma$ are of interest. This observation limits the extent to which Δ needs to be introduced around the goal (any resolvent inferred when at least one of the two clauses is an assumption clause can be treated as an assumption clause here — unlike for the unification constraints). The preceding observation contrasts with cancellation of the goal side with the goal side of the logical consequent; Section 5.4 illustrates that this causes a problem in LPF and leads to the necessary introduction of Δ around the goal.

1	$\forall i: \mathbb{Z} \cdot i = 0 \Rightarrow \neg((i - 1) = 0)$	<i>assumption</i>
2	$\forall i: \mathbb{Z} \cdot \neg(i = 0) \Rightarrow i \div i = 1$	<i>assumption</i>
3	$\forall i: \mathbb{Z} \cdot (i - 1) \in \mathbb{Z}$	<i>assumption</i>
4	$\Delta(\forall i: \mathbb{Z} \cdot (i \div i = 1) \vee ((i - 1) \div (i - 1) = 1))$	<i>goal</i>
5	$\{\neg(i = 0), \neg((i - 1) = 0)\}$	<i>clausal_form(1)</i>
6	$\{i = 0, i \div i = 1\}$	<i>clausal_form(2)</i>
7	$\{(i - 1) \in \mathbb{Z}\}$	<i>clausal_form(3)</i>
8	$\{\neg(c \div c = 1), \neg\Delta(c \div c = 1)\}$	
9	$\{\neg((c - 1) \div (c - 1) = 1), \neg\Delta((c - 1) \div (c - 1) = 1)\}$	<i>deny(clausal_form(4))</i>
10	$\{\neg\Delta(c \div c = 1), \neg\Delta((c - 1) \div (c - 1) = 1)\}$	
11	$\{c = 0, \neg(c \div c = 1)\}$	<i>resolve(6, 8)</i>
12	$\{c = 0\}$	<i>resolve(6, 11)</i>
13	$\{(c - 1) = 0, \neg((c - 1) \div (c - 1) = 1), \neg((c - 1) \in \mathbb{Z})\}$	<i>resolve(6, 9)</i>
14	$\{(c - 1) = 0, \neg((c - 1) \in \mathbb{Z})\}$	<i>resolve(6, 13)</i>
15	$\{\neg((c - 1) = 0)\}$	<i>resolve(5, 12)</i>
16	$\{\neg((c - 1) \in \mathbb{Z})\}$	<i>resolve(14, 15)</i>
17	\square	<i>resolve(7, 16)</i>

Figure 8: An illustrative proof of Property 1 using resolution as part of a refutation procedure which also refutes the possibility of a “gap”.

If the goal does not need surrounding with Δ then the unification constraints –which are still needed– could be considered in the first proof on the set of clauses $\Gamma \cup \{\neg e\}$.

5.6 Equality

In Section 5 equality has not been constrained to actually mean *equality*. The equality symbol used so far is just a binary predicate that is interpreted arbitrarily (that is, when considering validity there are more interpretations for the predicate written as $=$ than just equality itself). Potential issues when constraining equality are now considered.

One approach of handling the equality relational operator in first-order predicate logic is to add axioms stating that equality is *reflexive* ($\forall x \cdot x = x$), *symmetric* ($\forall x \cdot \forall y \cdot x = y \Rightarrow y = x$) and *transitive* ($\forall x \cdot \forall y \cdot \forall z \cdot x = y \wedge y = z \Rightarrow x = z$) as well as axioms that assert the *congruence* ($\forall x \cdot \forall y \cdot x = y \Rightarrow f(x) = f(y)$) of each n-ary function used and similarly for each n-ary predicate used. By providing the equality axioms explicitly, resolution can be used to solve first-order logic problems with equality. However, particularly due to the congruence axioms, this approach is inefficient as it leads to an explosion in the number of clauses required.

In LPF the notion of equality is considered to be strict and thus while reflexivity in two-valued classical logic is defined as:

$$\frac{}{x = x}$$

in LPF [BFL⁺94] it is defined as:

$$\frac{x: T}{x = x}$$

to ensure that x is defined. Additionally, the symmetric axiom and the transitive axiom carry a similar constraint in LPF. Notice that since quantification is only over defined values, this issue is avoided in \mathcal{E} . However, the function (and predicate) congruence axioms do not hold since it could be the case that x and y are equal to each other but when given as arguments to a function (or predicate) in the consequent a “gap” may arise, which would propagate up and cause a “gap” in the equality.

Another approach to handling equality is to make use of the *paramodulation* rule [RW69, Har09] alongside the resolution rule. Handling equality in LPF is left as a topic for future work.

6 Conclusions

LPF is a logic designed to provide a way of reasoning about logical formulae that can include partial terms. This paper has considered applying the classical resolution procedure alongside a refutation procedure in LPF; it has identified the

pitfalls that arise in doing so and outlined the extensions and the modifications that are required to successfully carry these techniques over to LPF. Illustrative proofs have been provided which are all based upon a semantic definition of LPF which provides a set theoretic definition of the values that are denoted by expressions.

Since LPF provides the strongest possible monotonic extension of the familiar classical logic propositional operators, properties such as the commutativity and the distributivity of disjunction and conjunction are retained. This also means that the well-known classical logic clausal form conversions carry over to LPF. However, the occurrence of the definedness operator (Δ) in LPF results in the need for extra conversion rules to be introduced into the clausal form conversion process, which has the undesired result of leading to more expensive resulting clausal form formulae — fortunately the use of Δ is constrained.

The idea of resolution carries over from the classical case to the LPF case when only considering satisfiability. However, the use of a refutation procedure in LPF brings about “extra” overhead due to the presence of “gaps”. A refutation procedure forces the introduction of Δ into the proofs since the definedness of the consequent now needs to be established when discharging proofs about validity. However, when using resolution within a refutation procedure, resolution can be extended to cope with the definedness obligations (Δ) to allow for both the false case and for the “gap” case to be refuted. The use of resolution within a refutation procedure also means that any use of unification needs to be carefully guarded. So, while for validity a less efficient procedure is required for LPF, when only considering satisfiability the existing (semi-)decision procedure of resolution carries over from the classical case to the LPF case relatively unchanged.

In [KK94] a mechanisation of Kleene logic for partial functions is presented. Kleene’s logic is formalised in an order-sorted three-valued logic and a resolution calculus is presented. This differs from what is proposed in this paper which undertakes a thorough investigation of where “undefinedness” arises and this can lead to a reduction in the number of definedness obligations that are needed (and thus have to be discharged) as well as a reduction in the size of the resulting clausal form of a formula (when using Δ).

As discussed in Section 5.6, the modifications required for LPF when constraining equality is left as a topic for further work. Additional further work is to apply the modified technique(s) to proof obligations from case studies to see how often “undefinedness” is a problem and whether such proof obligations can be efficiently discharged using the techniques proposed in this paper for the non-classical LPF.

Acknowledgements

The authors would like to thank Matthias Schmalz for a helpful discussion on work related to the topic of this paper. (In passing, Schmalz’s forthcoming ETH thesis is strongly related to the subject of the current paper and will be recommended reading as soon as it is approved.) The authors of this paper also gratefully acknowledge the funding for their research from an EPSRC PhD Studentship, the EPSRC grant for AI4FM, the EPSRC Platform Grant TrAmS-2 and the EU IP funding for DEPLOY (which last supported the contact with ETH).

References

- [BA01] Mordechai Ben-Ari. *Mathematical Logic for Computer Science*. Springer, 2 edition, 2001.
- [BCJ84] H. Barringer, J.H. Cheng, and C. B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21:251–269, 1984.
- [BFL⁺94] Juan Bicarregui, John Fitzgerald, Peter Lindsay, Richard Moore, and Brian Ritchie. *Proof in VDM: A Practitioner’s Guide*. FACIT. Springer-Verlag, 1994. ISBN 3-540-19813-X.
- [Che86] J. H. Cheng. *A Logic for Partial Functions*. PhD thesis, University of Manchester, 1986.
- [CJ90] J. H. Cheng and C. B. Jones. On the usability of logics which handle partial functions. Technical Report UMCS-90-3-1, Manchester University, February 1990. Preprint of [CJ91].
- [CJ91] J. H. Cheng and C. B. Jones. On the usability of logics which handle partial functions. In C. Morgan and J. C. P. Woodcock, editors, *3rd Refinement Workshop*, pages 51–69. Springer-Verlag, 1991.
- [Cor96] Roberto Cordeschi. The role of heuristics in automated theorem proving j.a. robinsons resolution principle. *Mathware and Soft Computing*, 3:281–293, 1996.

- [Fit07] J. S. Fitzgerald. The Typed Logic of Partial Functions and the Vienna Development Method. In D. Bjørner and M. C. Henson, editors, *Logics of Specification Languages*, EATCS Texts in Theoretical Computer Science, pages 427–461. Springer, 2007.
- [GSE95] David Gries, Fred B. Schneider, and Albert Einstein. Avoiding the undefined by underspecification. In *Computer Science Today: Recent Trends and Developments, number 1000 in Lecture Notes in Computer Science*, pages 366–373. Springer-Verlag, 1995.
- [Har09] John Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.
- [JLM91] C. B. Jones, K. D. Jones, P. A. Lindsay, and R. Moore. *mural: A Formal Development Support System*. Springer-Verlag, 1991.
- [JL11] C. B. Jones and M. J. Lovert. Semantic models for a logic of partial functions. *IJSI*, 5:55–76, 2011.
- [JLS12] C. B. Jones, M. J. Lovert, and L. J. Steggle. A semantic analysis of logics that cope with partial terms. Technical Report CS-TR-1310, Newcastle University, January 2012.
- [JM94] C.B. Jones and C.A. Middelburg. A typed logic of partial functions reconstructed classically. *Acta Informatica*, 31(5):399–430, 1994.
- [Jon90] C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, second edition, 1990.
- [Jon06] Cliff B. Jones. Reasoning about partial functions in the formal development of programs. In *Proceedings of AVOCS'05*, volume 145, pages 3–25. Elsevier, Electronic Notes in Theoretical Computer Science, 2006.
- [KK94] Manfred Kerber and Michael Kohlhase. A mechanization of strong kleene logic for partial functions. In *Proceedings of the 12th International Conference on Automated Deduction, CADE-12*, pages 371–385. Springer-Verlag, 1994.
- [Kle52] S. C. Kleene. *Introduction to Metamathematics*. Van Nostrand, 1952.
- [Lov10] M. J. Lovert. A semantic model for a logic of partial functions. In K. Pierce, N. Plat, and S. Wolff, editors, *Proceedings of the 8th Overture Workshop*, number CS-TR-1224 in School of Computing Science Technical Report, pages 33–45. Newcastle University, 2010.
- [McC67] J. McCarthy. A basis for a mathematical theory for computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland Publishing Company, 1967.
- [MS97] O. Müller and K. Slind. Treating partiality in a logic of total functions. *The Computer Journal*, 40(10):640–652, 1997.
- [Owe85] O. Owe. An approach to program reasoning based on a first order logic for partial functions. Technical Report 89, Institute of Informatics, University of Oslo, February 1985.
- [Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12:23–41, 1965.
- [RW69] G. Robinson and L. Wos. Paramodulation and theorem proving in first order theories with equality. In B. Meltzer and D. Michie, editors, *Machine Intelligence, volume IV*, pages 135–150. American Elsevier, 1969.
- [Sch11] Matthias Schmalz. Term rewriting in logics of partial functions. In Shengchao Qin and Zongyan Qiu, editors, *Formal Methods and Software Engineering*, volume 6991 of *Lecture Notes in Computer Science*, pages 633–650. Springer Berlin / Heidelberg, 2011. 10.1007/978-3-642-24559-6_42.
- [Sto77] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.
- [WCR64] Lawrence Wos, Daniel Carson, and George Robinson. The unit preference strategy in theorem proving. In *Proceedings of the October 27-29, 1964, fall joint computer conference, part I, AFIPS '64 (Fall, part I)*, pages 615–621. ACM, 1964.