COMPUTING SCIENCE

Revising Basic Theorem Proving Algorithms to Cope with the Logic of Partial Functions

Cliff B. Jones, Matthew J. Lovert and L. Jason Steggles

TECHNICAL REPORT SERIES

No. CS-TR-1414 March 2014

TECHNICAL REPORT SERIES

No. CS-TR-1414 March, 2014

Revising Basic Theorem Proving Algorithms to Cope with the Logic of Partial Functions

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

Abstract

Partial terms are those that can fail to denote a value; such terms arise frequently in the specification and development of programs. Earlier papers describe and argue for the use of the non-classical "Logic of Partial Functions" (LPF) to facilitate sound and convenient reasoning about such terms. This paper reviews the fundamental theorem proving algorithms -such as resolution- and identifies where they need revision to cope with LPF. Particular care is needed with "refutation" procedures. The modified algorithms are justified with respect to a semantic model. Indications are provided of further work which could lead to efficient support for LPF.

Printed and published by Newcastle University,
Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.

Bibliographical details

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

Revising Basic Theorem Proving Algorithms to Cope with the Logic of Partial Functions
[By] C. B. Jones, M. J. Lovert and L. J. Steggles

Newcastle upon Tyne: Newcastle University: Computing Science, 2014.

(Newcastle University, Computing Science, Technical Report Series, No. CS-TR-1414)

Added entries

NEWCASTLE UNIVERSITY
Computing Science. Technical Report Series. CS-TR-1414

Abstract

Partial terms are those that can fail to denote a value; such terms arise frequently in the specification and development of programs. Earlier papers describe and argue for the use of the non-classical "Logic of Partial Functions" (LPF) to facilitate sound and convenient reasoning about such terms. This paper reviews the fundamental theorem proving algorithms -such as resolution- and identifies where they need revision to cope with LPF. Particular care is needed with "refutation" procedures. The modified algorithms are justified with respect to a semantic model. Indications are provided of further work which could lead to efficient support for LPF.

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. He is PI of two EPSRC-funded responsive mode projects “AI4FM” and "Taming Concurrency", CI on the Platform Grant TrAmS-2 and 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.

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

PARTIAL FUNCTIONS
LOGIC OF PARTIAL FUNCTIONS
RESOLUTION
AUTOMATED THEOREM PROVING
REFUTATION PROCEDURE

Revising Basic Theorem Proving Algorithms to Cope with the Logic of Partial Functions

Cliff B. Jones, Matthew J. Lovert and L. Jason Steggles
School of Computing Science, University of Newcastle, U. K.

Abstract

Partial terms are those that can fail to denote a value; such terms arise frequently in the specification and development of programs. Earlier papers describe and argue for the use of the non-classical "Logic of Partial Functions" (LPF) to facilitate sound and convenient reasoning about such terms. This paper reviews the fundamental theorem proving algorithms –such as resolution– and identifies where they need revision to cope with LPF. Particular care is needed with "refutation" procedures. The modified algorithms are justified with respect to a semantic model. Indications are provided of further work which could lead to efficient support for LPF.

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 [18, 24, 30]. 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 both establishing consistency of specifications at any level of abstraction and for justifying development steps between levels (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 [8, 12, 9, 10, 13, 16, 1, 14, 34].

The issue of reasoning about partial functions is by no means purely theoretical: in [35], it is identified as a significant source of inconsistencies in the theorem provers for Event-B; after the 2011 "Landin seminar" by one of the current authors, David Crocker pointed out that one of very few inconsistencies in "Perfect Developer" again revolved around the issue of undefined terms.

This paper is intended to interest computer scientists in alternatives to bending partial functions into the classical model of first-order predicate calculus. It is not so much aimed at logicians.

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

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

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 3).[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

[1] As explained in earlier papers, the problem of non-denoting terms is pervasive and most of these papers have used examples with recursive functions; in this paper, the fact that division is a partial operator is used to present the essential points with a minimum of extra machinery.

the truth of disjunctions such as $(1 \div 1 = 1) \vee (0 \div 0 = 1)$, which reduces to $(1 = 1) \vee (\bot_{\mathbb{Z}} = 1)$. With strict (weak/computational) equality (undefined if either operand is undefined), this further reduces to $\mathit{true} \vee \bot_{\mathbb{B}}$ which makes no sense in classical logic since its truth tables only define the propositional 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*) [3, 8, 10, 23, 24, 20], where "gaps" are handled by lifting the logical operators. Property 1 is true in LPF and its proof presents no difficulty (after some explanation, this proof is given in Figure 2). However, Property 1 can cause "issues" in other approaches to coping with non-denoting terms — for example, with McCarthy's conditional version[2] of the logical operators [29], where disjunctions and conjunctions are not commutative and quantifiers are problematic with respect to undefined values.

However, the availability of a large body of proof techniques for classical logic presents an argument against the adoption of LPF. These fundamental automated proof techniques are the foundation on which many advanced automated proof techniques are built and as such represent a natural starting point for considering the development of proof support for LPF. Determining how to modify these proof procedures for LPF and analysing the associated performance issues provides key insights into mechanising proof for a logic like LPF. Furthermore, it provides the essential foundation to facilitate the modification of more advanced proof techniques for LPF. The main contribution of this paper (Section 5) is to pinpoint the issues that arise for the adaption of techniques such as proof by refutation and resolution to cope with LPF. In some cases, the justification of the extended algorithms is essentially the same as with their classical counterparts; only where there are significant changes are new proofs provided. In particular, the soundness of the modified resolution procedure is proved; resolution completeness is the subject of on-going research.

Structure of the paper: Section 2 provides an introduction to LPF. Section 3 provides a semantics for the LPF version of the Predicate Calculus — the rest of the paper is grounded on this semantic model. Section 4 discusses normal forms. Section 5 outlines the issues present –and the changes required– for the proof procedures to cover LPF. Finally, Section 6 provides some conclusions and an indication of further work.

Note that this technical report reworks and extends the previous technical report [22].

2 An Introduction to LPF

LPF is a first order logic that can handle non-denoting logical values that arise from terms that apply partial functions and operators; it is the logic that underlies the *Vienna Development Method* (*VDM*) [18, 5, 15]; there was an instantiation of LPF on the *mural* formal development support system [19]. Arguments for the use of LPF are documented in several of the previously cited references, particulary [10, 24, 20].

It is straightforward to lift the standard two-valued truth tables for propositional operators to cover logical values that may fail to denote (see for example [26, §64]). Such tables provide the strongest possible *monotonic* extension of the familiar classical propositional operators with

[2] McCarthy defined, for example, the disjunction of e_1, e_2 as **if** e_1 **then** tt **else** e_2 and referred to the first variable in such conditional expressions as the "inevitable variable" since conditionals are strict in their first argument. This interpretation is implemented in some programming languages –sometimes with distinct keywords– and this imposes a proof obligation when copying LPF expressions into program texts cf. [18].

(a)

$\vee$	$\mathfrak{tt}$	$\bot_{\mathbb{B}}$	$\mathfrak{ff}$
$\mathfrak{tt}$	$\mathfrak{tt}$	$\mathfrak{tt}$	$\mathfrak{tt}$
$\bot_{\mathbb{B}}$	$\mathfrak{tt}$	$\bot_{\mathbb{B}}$	$\bot_{\mathbb{B}}$
$\mathfrak{ff}$	$\mathfrak{tt}$	$\bot_{\mathbb{B}}$	$\mathfrak{ff}$

(b)

e	Δe
$\mathfrak{tt}$	$\mathfrak{tt}$
$\bot_{\mathbb{B}}$	$\mathfrak{ff}$
$\mathfrak{ff}$	$\mathfrak{tt}$

(c)

e	δe
$\mathfrak{tt}$	$\mathfrak{tt}$
$\bot_{\mathbb{B}}$	$\bot_{\mathbb{B}}$
$\mathfrak{ff}$	$\mathfrak{tt}$

Figure 1: The LPF truth tables for disjunction and the definedness operators Δ and δ.

respect to the ordering on truth values: $\bot_{\mathbb{B}} \preceq \mathfrak{tt}$ and $\bot_{\mathbb{B}} \preceq \mathfrak{ff}$. As an example, the truth table for disjunction is given in Figure 1(a). Alternatively, such truth tables can be viewed as describing a parallel (lazy) evaluation of the operands that delivers a result as soon as enough information is available; such a result would not be contradicted if a $\bot_{\mathbb{B}}$ were evaluated to a proper Boolean value.

The way in which non-denoting values can be "caught" by these extended propositional operators can be depicted as follows[3]

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

where $\mathbb{Z}_\bot$ stands for $\mathbb{Z} \cup \{\bot_{\mathbb{Z}}\}$ and $\mathbb{B}_\bot$ stands for $\mathbb{B} \cup \{\bot_{\mathbb{B}}\}$.

Quantifiers in LPF are a natural extension of the propositional operators: existential quantification is equivalent to an (in the worst case) infinite disjunction and universal quantification to 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); it 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, quantified variables range over proper (i.e. defined) values.

Standard algebraic laws (de Morgan) relate $\vee/\wedge$ and the quantifiers; implication has its normal definition; commutativity and distribution hold as in standard first-order predicate calculus. One issue with the use of LPF is that the, so called, *law of the excluded middle* $(p \vee \neg p)$ does not hold because the disjunction of two undefined Boolean values is undefined: thus $(0 \div 0 = 1) \vee \neg(0 \div 0 = 1)$ is not a tautology in LPF.

For expressive completeness, LPF adds a definedness operator Δ whose truth table is in Figure 1(b). Unlike all of the other operators, the Δ operator is not monotone. It also gives rise to the property for LPF which is known as the *law of the excluded fourth* $(p \vee \neg p \vee \neg \Delta p)$. Adding definedness hypotheses for all terms in some logical expression e is sufficient to make the validity of e in LPF and classical logic coincide.

Whilst providing lifted truth tables is straightforward, it is less obvious how to present an axiomatisation. This is done for untyped LPF in [3, 8] and for typed LPF in [23].

The normal notion of a proof is that one proceeds from assumptions and derives their consequences. A *sequent* $e_1, \ldots, e_n \vdash e$ is used to represent the situation when the formula e can be logically derived from the assumptions $e_1, \ldots, e_n$. 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 (concluding $\vdash e_1 \Rightarrow e_2$ from $e_1 \vdash e_2$) — this does not hold in LPF because e_1 could be an arbitrary 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

[3] Comparisons of several differing approaches to handling undefined values are supported by pictures of this style in [21]. Note that it is *not* claimed that such types are syntactically decidable.

	from $\forall i \cdot i = 0 \;\Rightarrow\; \neg\,(i - 1 = 0);\;\; \forall i \cdot \neg\,(i = 0) \;\Rightarrow\; i \div i = 1;$	
	$\forall i, j \cdot (i - j) \in \mathbb{Z}$	
1	**from** $i : \mathbb{Z}$	
1.1	$\neg\,(i = 0) \vee \neg\,(i - 1 = 0)$	$\forall$-$E(h, h1), \Rightarrow$-*defn*
1.2	**from** $\neg\,(i = 0)$	
1.2.1	$i \div i = 1$	$\Rightarrow$-$E(\forall$-$E(h, h1), h1.2)$
	infer $(i \div i = 1) \vee ((i - 1) \div (i - 1) = 1)$	$\vee$-I-$R(1.2.1)$
1.3	**from** $\neg\,(i - 1 = 0)$	
1.3.1	$(i - 1) \in \mathbb{Z}$	$\forall$-$E(h, h1)$
1.3.2	$(i - 1) \div (i - 1) = 1$	$\Rightarrow$-$E(\forall$-$E(h, 1.3.1), h1.3)$
	infer $(i \div i = 1) \vee ((i - 1) \div (i - 1) = 1)$	$\vee$-I-$L(1.3.1)$
	infer $(i \div i = 1) \vee ((i - 1) \div (i - 1) = 1)$	$\vee$-$E(1.1, 1.2, 1.3)$
	infer $\forall i \cdot (i \div i = 1) \vee ((i - 1) \div (i - 1) = 1)$	$\forall$-$I(1)$

Figure 2: An illustrative proof of Property 1 in LPF.

can provide a sound "$\Rightarrow$-I" rule for LPF. 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 (cf. Figure 1(c)) can be used which is monotone and whose definition is the same as Δ except that $\delta \bot_{\mathbb{B}} = \bot_{\mathbb{B}}$ rather than false, thus δe_1 is equivalent to the assertion $e_1 \vee \neg\, e_1$ (see Figure 1(c)). Therefore, the following "$\Rightarrow$-I" rule for LPF

$$\boxed{\Rightarrow\text{-}I}\ \frac{\vdash \delta e_1; \qquad e_1 \vdash e_2}{\vdash e_1 \;\Rightarrow\; e_2}$$

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 6.2 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 [23] 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 "intended" semantics of the operators $-/\div$:

$$\begin{array}{l} \forall i : \mathbb{Z} \cdot i = 0 \;\Rightarrow\; \neg\,((i - 1) = 0);\;\; \forall i : \mathbb{Z} \cdot \neg\,(i = 0) \;\Rightarrow\; i \div i = 1 \\ \quad \forall i, j \cdot (i - j) \in \mathbb{Z} \vdash \\ \quad\quad \forall i : \mathbb{Z} \cdot (i \div i = 1) \vee ((i - 1) \div (i - 1) = 1) \end{array}$$

Since all uses of i and j as quantified variables are of integer type, the type is left implicit for the remainder of the paper (and, below, the type is omitted in $\sigma : \Sigma$). The proof of this property in LPF is straightforward and, as can be seen in Figure 2, is not complicated by "undefinedness" issues despite the fact that the example has been deliberately chosen so that either of the disjuncts could be undefined.

3 Semantics of LPF

This section presents a semantics for the LPF version of Predicate Calculus. The semantics is used to redefine standard logical notions –such as notions of a formula being satisfiable and valid– for LPF.

3.1 The Semantic Function $[\![e]\!]$

The operators and quantifiers for LPF include those of the standard predicate calculus; the addition of the definedness operator Δ is explained below. Also –as is standard– conjunction, implication and universal quantification are viewed as syntactic sugar for expressions using a basic set of operators (it is one of the advantages of LPF over say McCarthy's conditional operators that the standard definitions apply).

A concrete syntax for LPF using *Extended Backus-Naur Form* is provided in [22] but the cases in the following semantic definition ought to provide an adequate view of the syntax of LPF. Context conditions for LPF –that limit the formulae to which semantics need be given– are outlined in [20] and spelt out fully in [27].

The challenge of giving a semantics that covers partial terms (e.g $i \div 0$) is met by mapping formulae to relations between states and values.[4] A state for which the expression does not denote a value is absent from the domain of the relation. States map identifiers to their values. The set of identifiers *Id* is partitioned into four subsets for propositions *Prop*, (integer) variables *Var*, functions *Fn* and predicates *Pred*. Furthermore, it is assumed that formulae bind all variables by quantifiers; quantified values range only over proper (i.e. defined) integers. (The limitation to integers is for brevity only.)

The set Σ of all maps from identifiers to their values is defined as:

$$\Sigma = \mathit{Id} \xrightarrow{m} \mathit{Value}$$
$$\mathit{Value} = \mathbb{B} \cup \mathbb{Z} \cup \mathit{Function} \cup \mathit{Predicate}$$

It is assumed that each identifier maps to a value of appropriate type. The map involving *Prop* can be partial: a propositional identifier can be absent from the domain of a specific σ to allow for undefined propositional identifiers. The maps involving *Var*, *Fn* and *Pred* are however total.

The denotations of *Function*s and *Predicate*s are relations, thus (using $\mathcal{P}$ for power set):

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

Functions/predicates have a fixed arity in any given σ but can be partial — this is the reason for using relations as their denotations but they always return the same result for any given argument(s) in a given σ. Functions and predicates are assumed to be strict: if there is a "gap" in an argument then there is a "gap" in the result of applying the function/predicate to that argument.

The semantic function $[\![e]\!]$ is given in Figure 3. (The limited use of VDM notation should provide no difficulty except perhaps: **dom** yields the domain of a relation; similarly, **rng** provides the range; $m_1 \dagger m_2$ yields a relation in which the second argument overwrites matching values in the first; $s \lhd m$ is a sub-relation of m containing only those pairs whose first elements are also in s. Full details can be found in [18]). This semantics defines the "lifted" LPF operators in terms of (set theory and) the standard logical operators and quantifiers on $\mathbb{B}$. To emphasise the distinction the standard operators and quantifiers are marked with a subscript as in $\vee_{\mathbb{B}}$. It is easy to check that their operands must be defined because they all rely on set membership. The definition of

[4] This is in contrast to the more common use of partial functions in *denotational semantics* [36]. Since the aim is to explain a logic over partial functions, a clear distinction between the concepts of the meta and object languages seems sensible.

function application deserves some explanation: the values of the elements of the argument list al are evaluated to form vl; if any such argument does not yield a proper value, no application is made; in the defined case, r is determined by $(vl, r) \in \sigma(f)$.

(A paper [21] by the current authors compares some of the main approaches to handling partial terms; for each approach, it presents similar semantic models to that in Figure 3 — thus illustrating where "undefinedness" is handled in each approach.)

$$
\begin{array}{l}
[\![\text{-}]\!] : \mathit{Expr} \to \mathcal{P}(\Sigma \times \mathit{Value}) \\
[\![e]\!] \triangleq \\
\quad \textbf{cases } e \textbf{ of} \\
\quad e \in \mathit{Value} \to \{(\sigma, e) \mid \sigma \in \Sigma\} \\
\quad e \in \mathit{Prop} \ \to \{(\sigma, \sigma(e)) \mid \sigma \in \Sigma \wedge_{\mathbb{B}} e \in \textbf{dom}\, \sigma\} \\
\quad e \in \mathit{Var} \ \to \{(\sigma, \sigma(e)) \mid \sigma \in \Sigma\} \\
\quad f(al) \qquad \to \{(\sigma, r) \mid \sigma \in \Sigma \wedge_{\mathbb{B}} \\
\qquad\qquad\qquad f \in (\mathit{Fn} \cup \mathit{Pred}) \wedge_{\mathbb{B}} \\
\qquad\qquad\qquad \forall i : \textbf{inds}\, al \cdot (\sigma, vl(i)) \in [\![al(i)]\!] \wedge_{\mathbb{B}} \\
\qquad\qquad\qquad (vl, r) \in \sigma(f)\} \\
\quad \neg\, e' \qquad \to \{(\sigma, \mathrm{tt}) \mid (\sigma, \mathrm{ff}) \in [\![e']\!]\} \cup \{(\sigma, \mathrm{ff}) \mid (\sigma, \mathrm{tt}) \in [\![e']\!]\} \\
\quad e_1 \vee e_2 \quad \to \{(\sigma, \mathrm{tt}) \mid (\sigma, \mathrm{tt}) \in [\![e_1]\!] \vee_{\mathbb{B}} (\sigma, \mathrm{tt}) \in [\![e_2]\!]\} \cup \\
\qquad\qquad \{(\sigma, \mathrm{ff}) \mid (\sigma, \mathrm{ff}) \in [\![e_1]\!] \wedge_{\mathbb{B}} (\sigma, \mathrm{ff}) \in [\![e_2]\!]\} \\
\quad \exists x \cdot e' \quad \to \{(\sigma, \mathrm{tt}) \mid \sigma \in \Sigma \ \wedge_{\mathbb{B}} \\
\qquad\qquad\qquad \exists_{\mathbb{B}}\, i \in \mathbb{Z} \cdot (\sigma \dagger \{x \mapsto i\}, \mathrm{tt}) \in [\![e']\!]\} \cup \\
\qquad\qquad \{(\sigma, \mathrm{ff}) \mid \sigma \in \Sigma \ \wedge_{\mathbb{B}} \\
\qquad\qquad\qquad \forall_{\mathbb{B}}\, i \in \mathbb{Z} \cdot (\sigma \dagger \{x \mapsto i\}, \mathrm{ff}) \in [\![e']\!]\} \\
\quad \Delta e' \qquad \to \{(\sigma, \mathrm{tt}) \mid \sigma \in \textbf{dom}\, [\![e']\!]\} \cup \\
\qquad\qquad \{(\sigma, \mathrm{ff}) \mid \sigma \in (\Sigma \setminus \textbf{dom}\, [\![e']\!])\} \\
\quad \textbf{end}
\end{array}
$$

Figure 3: The semantic function $[\![e]\!]$ which defines the semantics of LPF.

The motivating example of Property 1 uses functions and predicates whose denotations might be:

$$
\begin{array}{l}
\sigma(\textbf{minus}) \triangleq \{((a, b), a - b) \mid a, b : \mathbb{Z}\} \\
\sigma(\textbf{div}) \triangleq \{((a, b), a \div b) \mid a, b : \mathbb{Z} \wedge_{\mathbb{B}} b \neq 0\} \\
\sigma(\textbf{equals}) \triangleq \{((a, b), a = b)) \mid a, b : \mathbb{Z}\}
\end{array}
$$

where the operators on the right hand sides of these definitions have their standard mathematical meaning. Notice that, whereas subtraction is total, division and (strict) equality are partial. These denotations have only been given for illustrative purposes — it is important to realise that the hypotheses in the proof in Figure 2 are less constraining.

3.2 Reasoning using the semantics

A convenient abbreviation, which emphasises the fact that a relation is involved, is to write $(\sigma, b) \in [\![e]\!]$ as $\sigma [\![e]\!] b$.

It is useful to record that the definition of any relation $[\![e]\!]$ is deterministic (or "functional"):

Lemma 1. For any expression e it follows that $\sigma [\![e]\!] v_1 \wedge \sigma [\![e]\!] v_2 \ \Rightarrow \ v_1 = v_2$.

Proof. This follows from the fact that there is exactly one rule for each type of expression and, where the resulting relation is defined by uniting sets, the domains of the relations are disjoint. □

It is a useful property of LPF that the standard definitions of extended operators apply:

Definition 2. The following abbreviations are used:
$e_1 \wedge e_2$ for $\neg\,(\neg\, e_1 \vee \neg\, e_2)$
$e_1 \;\Rightarrow\; e_2$ for $\neg\, e_1 \vee e_2$
$\forall x \cdot e$ for $\neg\, \exists e \cdot \neg\, e$

Thus:

$$
\begin{array}{l}
[\![e_1 \wedge e_2]\!] = \\
\quad \{(\sigma, \mathfrak{tt}) \mid (\sigma, \mathfrak{tt}) \in [\![e_1]\!] \wedge_{\mathbb{B}} (\sigma, \mathfrak{tt}) \in [\![e_2]\!]\} \cup \\
\quad \{(\sigma, \mathfrak{ff}) \mid (\sigma, \mathfrak{ff}) \in [\![e_1]\!] \vee_{\mathbb{B}} (\sigma, \mathfrak{ff}) \in [\![e_2]\!]\} \\
[\![\forall x \cdot e']\!] = \\
\quad \{(\sigma, \mathfrak{tt}) \mid \sigma \in \Sigma \;\wedge_{\mathbb{B}} \forall_{\mathbb{B}} i \in \mathbb{Z} \cdot (\sigma \dagger \{x \mapsto i\}, \mathfrak{tt}) \in [\![e']\!]\} \cup \\
\quad \{(\sigma, \mathfrak{ff}) \mid \sigma \in \Sigma \;\wedge_{\mathbb{B}} \exists_{\mathbb{B}} i \in \mathbb{Z} \cdot (\sigma \dagger \{x \mapsto i\}, \mathfrak{ff}) \in [\![e']\!]\}
\end{array}
$$

Lemma 3. Many standard results can be proved for LPF using $[\![e]\!]$ and the above definitions.
(i) $e \equiv \neg\neg\, e$, for any formula e.
(ii) $\neg\,(e_1 \vee e_2) \equiv (\neg\, e_1) \wedge (\neg\, e_2)$
(iii) $\neg\,(e_1 \wedge e_2) \equiv (\neg\, e_1) \vee (\neg\, e_2)$
(iv) $e_1 \vee (e_2 \wedge e_3) \equiv (e_1 \vee e_2) \wedge (e_1 \vee e_3)$
(v) $(e_1 \vee e_1) \equiv e_1$ and $(e_1 \wedge e_1) \equiv e_1$
(vi) $(e_1 \vee e_2) \equiv (e_2 \vee e_1)$ and $(e_1 \wedge e_2) \equiv (e_2 \wedge e_1)$
(vii) $e_1 \vee (e_2 \vee e_3) \equiv (e_1 \vee e_2) \vee e_3$ and $e_1 \wedge (e_2 \wedge e_3) \equiv (e_1 \wedge e_2) \wedge e_3$.
(viii) $p \wedge (p \vee q) \equiv p$

Proof. Proofs of this and subsequent trivial lemmas that use only expansion of $[\![e]\!]$ are generally omitted for brevity. However, a range of illustrative proofs for such lemmas can be found in the appendix. □

3.3 Satisfiability, Validity and Logical Consequence

As with the type of i being integers, the remainder of the paper omits the type of σ.

The notions of *satisfiability*, *validity* and *logical consequence* [4] can be defined for LPF using the semantics of Section 3.1. A formula e is said to be *satisfiable* in LPF iff $\sigma[\![e]\!]\mathfrak{tt}$, for some interpretation σ. A formula e is *unsatisfiable* iff it is not satisfiable (i.e. for all interpretations σ, $(\sigma, \mathfrak{tt}) \notin [\![e]\!]$). It is essential that the above formulation is used for unsatisfiability in LPF. Consider, for example, defining unsatisfiability by $\sigma[\![e]\!]\mathfrak{ff}$, for all interpretations σ. In this case the set of unsatisfiable expressions would be smaller since in LPF an expression e not evaluating to $\mathfrak{tt}$ is not the same as it evaluating to $\mathfrak{ff}$ due to the presence of "gaps". The above definition of satisfiable can be extended to a set of formulae: a set of formulae S is said to be *(simultaneously) satisfiable* iff there exists an interpretation $\sigma \in \Sigma$ such that $\sigma[\![e_i]\!]\mathfrak{tt}$, for all $e_i \in S$. A set of formulae S is said to be *(simultaneously) unsatisfiable* iff they are not satisfiable.

A formula e is said to be *valid* in LPF, denoted $\models e$, iff $\sigma[\![e]\!]\mathfrak{tt}$ holds for all interpretations σ. The notation $\not\models e$ is used to represent that e is not valid.

Let $\Gamma = \{e_1, \ldots, e_n\}$ be a set of formulae and e be a single formula. Then $\Gamma \models e$ denotes that e is a *logical consequence* of Γ. This can be formally defined using $[\![e]\!]$ as follows: $\Gamma \models e$ holds iff for all interpretations σ, whenever $\sigma[\![e_1]\!]\mathfrak{tt}, \ldots, \sigma[\![e_n]\!]\mathfrak{tt}$ hold it follows that $\sigma[\![e]\!]\mathfrak{tt}$ holds.

Two formulae e_1 and e_2 are *logically equivalent*, denoted $e_1 \equiv e_2$, iff $[\![e_1]\!] = [\![e_2]\!]$. Furthermore, two formulae e_1 and e_2 are *equi-satisfiable* iff $\exists \sigma \cdot \sigma[\![e_1]\!]\mathfrak{tt} \;\Leftrightarrow\; \exists \sigma' \cdot \sigma'[\![e_2]\!]\mathfrak{tt}$.

4 Normal Forms in LPF

In order to mechanise and optimise proof procedures for classical logic a range of normal form representations for logical formulae are employed. One well-used normal form is *clausal form* [4], a set based representation for logical formulae which are structured as conjunctions of disjunctions. In this section the process of converting a predicate LPF formula into clausal form is investigated and a range of results are shown. While the standard conversion techniques considered here have well-known shortcomings (i.e. the potential rapid expansion of formulae), they provide an important foundation on which further investigations into more advanced optimisation techniques (see Section 6) can be based.

4.1 Conjunctive Normal Form and Clausal Form

In the standard literature a *literal* is an atomic formula or the negation thereof. A propositional formula is said to be in *conjunctive normal form (CNF)* iff it is a conjunction of disjunctions of literals. An important result in classical propositional logic is that every propositional formula can be converted into a logically equivalent one in CNF [4]. The standard conversion process to CNF can be summarised as follows:

1. eliminate any propositional operators other than conjunction, disjunction and negation by applying the standard syntactic conversions;
2. use *de Morgan's Laws* to push all negations inwards;
3. eliminate all double negations; and
4. use the *distributive law* to remove conjunctions within disjunctions.

It turns out that all of the equivalences used in the conversion process above hold in LPF as shown by Lemma 3.

However, the process of converting a propositional formula into CNF needs to be extended in LPF to incorporate rules for the non-monotone definedness Δ operator (although this occurs rarely in normal proofs, it has a significant role to play in Theorem 11). The first step is to extend the definition of a literal in LPF to include formulae of the form Δl and $\neg\, \Delta l$, for any literal l in the standard sense. The CNF conversion process defined above can then be extended by inserting a new step after Step 1 in which all occurrences of Δ are pushed inwards in a formula (similar to the approach taken for negation).

The Δ operator has some surprising properties. For example, although $\Delta e \vdash e \vee \neg\, e$ and $e \vee \neg\, e \vdash \Delta e$ are both valid deductions, the semantics shows that $[\![\Delta e]\!] \neq [\![e \vee \neg\, e]\!]$. This points to care being needed in its expansion. In order to facilitate moving Δ inwards, a range of equivalences are needed, for example: $\Delta(e_1 \vee e_2)$ is logically equivalent to

$$\neg((\neg e_1 \wedge \neg \Delta e_2) \vee (\neg \Delta e_1 \wedge \neg e_2) \vee (\neg \Delta e_1 \wedge \neg \Delta e_2))$$

which when converted to CNF gives

$$(e_1 \vee \Delta e_2) \wedge (\Delta e_1 \vee e_2) \wedge (\Delta e_1 \vee \Delta e_2).$$

(It would be wrong to use $e_1 \vee e_2 \vee \Delta e_1 \wedge \Delta e_2$ because this would not yield $\mathfrak{ff}$ in the case where e_1 was undefined and e_2 was $\mathfrak{ff}$.)

The following lemma gives the key logical equivalences required for dealing with Δ during the CNF conversion process in LPF.

Lemma 4. Let e_1 and e_2 be LPF formulae. Then the following logical equivalences involving Δ hold in LPF:
i) $\Delta(e_1 \vee e_2) \equiv (e_1 \vee \Delta e_2) \wedge (\Delta e_1 \vee e_2) \wedge (\Delta e_1 \vee \Delta e_2)$;
ii) $\Delta(e_1 \wedge e_2) \equiv (\neg e_1 \vee \Delta e_2) \wedge (\Delta e_1 \vee \neg e_2) \wedge (\Delta e_1 \vee \Delta e_2)$;
iii) $\Delta(\neg\, e_1) \equiv \Delta(e_1)$.

Proof. Based on expanding the relevant expressions using $[\![e]\!]$. □

Note that using these rules can result in significant expansion of formulae during the conversion process for formulae containing Δ. However, it is important to remember that Δ is not normally written in LPF and the expansion only becomes an issue in refutation procedures.

Recall that the classical equivalences of $(e \vee \neg\, e) \equiv \mathfrak{tt}$ and $(e \wedge \neg\, e) \equiv \mathfrak{ff}$ no longer hold in LPF. However, in LPF both $\Delta(\Delta e)$ and $(e \vee \neg\, e \vee \Delta e)$ can be shown to be logically equivalent to the truth value $\mathfrak{tt}$, and $(\Delta e \wedge \neg\, \Delta e)$ is equivalent to $\mathfrak{ff}$. These equivalences can be used during the conversion process to simplify terms.

The above results lead to the following equivalence theorem for CNF and propositional LPF formulae.

Theorem 5. Every propositional LPF formula can be converted into a logically equivalent propositional LPF formula that is in CNF.

Proof. This theorem follows immediately from Lemmas 3–4. □

In classical propositional logic, a CNF formula

$$((l_{1,1} \vee \cdots \vee l_{1,n_1}) \wedge \cdots \wedge (l_{m,1} \vee \cdots \vee l_{m,n_m}))$$

can be represented in *clausal form* [4] as a set of sets:

$$\{\{l_{1,1}, \ldots, l_{1,n_1}\}, \ldots, \{l_{m,1}, \ldots, l_{m,n_m}\}\}$$

The above representation relies on the properties of *idempotency*, *commutativity* and *associativity* that hold for $\vee$ and $\wedge$ in classical propositional logic. Thus, in order to use clausal form in LPF, properties (v)–(vii) of Lemma 3 are needed.

4.2 Prenex Normal Form

In the standard literature a predicate logic formula is said to be in *Prenex Normal Form (PNF)* if it consists of a sequence of quantifiers followed by a quantifier free formula (normally referred to as the *matrix*). So a formula in PNF is of the form

$$Q_1 x_1 \cdot \ldots \cdot Q_n x_n \cdot e$$

where each Q_i is either a universal or existential quantifier and e is the matrix.

The conversion of a predicate formula into clausal form in classical logic (see for example [4, 17]) proceeds by converting the formula into PNF and then *Skolemising* it. This removes any existential quantifiers in the formula replacing each with a Skolem function which takes as arguments any universally quantified variables that preceded the existential quantifier. The resulting Skolemised formula is equi-satisfiable to the original formula. The standard conversions are used on the matrix of the formula to ensure it is in CNF. The universal quantifiers can then be dropped allowing the formula to be represented in clausal form (the variables are interpreted as being implicitly universally quantified).

A key result in classical logic is that any predicate formula can be converted into a logically equivalent formula in PNF. This conversion process for PNF can be summarised as follows [4]:

1. standardise the variables apart by renaming variables, where necessary, so that no two quantifiers bind the same variable name;

2. push all negations inwards so that they only apply to atomic formulae through the use of de Morgan's Laws and quantifier conversions; and

3. push all quantifiers outwards through the use of appropriate conversions, such as $e_1 \vee \exists x \cdot e_2$ to $\exists x \cdot (e_1 \vee e_2)$ and $e_1 \vee \forall x \cdot e_2$ to $\forall x \cdot e_1 \vee e_2$.

The above conversion rules again turn out to be valid in LPF as the following result (with Lemma 3 and Definitions 2) shows.

From here on the abbreviation $\sigma^i = \sigma \dagger \{x \mapsto i\}$ is used freely.

Lemma 6. Let e_1 and e_2 be LPF formulae. Then, assuming e_1 contains no free occurrences of the variable x, $e_1 \vee \exists x \cdot e_2 \equiv \exists x \cdot (e_1 \vee e_2)$ and $e_1 \vee \forall x \cdot e_2 \equiv \forall x \cdot (e_1 \vee e_2)$.

Proof. The argument consists of expansion of the two formulae using the definition in Figure 3; it is simpler to trace these steps if the $\mathfrak{tt}/\mathfrak{ff}$ cases are separated; the intervening lines starting $\Leftrightarrow$ indicate the operators whose definitions justify the step from the preceding to the succeeding lines.

$\sigma[\![e_1 \vee \exists x \cdot e_2]\!]\mathfrak{tt}$
$\quad \Leftrightarrow$ expand using cases for $\vee$ and $\exists$
$\sigma[\![e_1]\!]\mathfrak{tt} \vee \exists i \cdot \sigma^i[\![e_2]\!]\mathfrak{tt}$

$\sigma[\![\exists x \cdot e_1 \vee e_2]\!]\mathfrak{tt}$
$\quad \Leftrightarrow$ expand using cases for $\exists$ and $\vee$
$\exists i \cdot (\sigma^i[\![e_1]\!]\mathfrak{tt} \vee \sigma^i[\![e_2]\!]\mathfrak{tt})$

Since x is not free in e_1, $\sigma^i[\![e_1]\!]\mathfrak{tt} \Leftrightarrow \sigma[\![e_1]\!]\mathfrak{tt}$ and the standard laws of predicate calculus complete the proof.

The argument for $\sigma[\![e_1 \vee \exists x \cdot e_2]\!]\mathfrak{ff} \Leftrightarrow \sigma[\![\exists x \cdot e_1 \vee e_2]\!]\mathfrak{ff}$ is similar. □

In LPF the process of converting a predicate formula into PNF needs to be extended to incorporate rules for handling the definedness Δ operator (as was done above for CNF). Any Δ surrounding a quantified formula needs to be pushed inwards and so appropriate equivalences are required. Just as with the discussion preceding Lemma 4, any surprise that the term $e \wedge \Delta e$ is needed is overcome by checking that the expansion yields $\mathfrak{ff}$ in all required cases.

Lemma 7. For any LPF formula p the following logical equivalences involving Δ hold in LPF:
i) $\Delta(\exists x \cdot e) \equiv \forall x \cdot \Delta e \vee \exists x \cdot (e \wedge \Delta e)$
ii) $\Delta(\forall x \cdot e) \equiv \forall x \cdot \Delta e \vee \exists x \cdot (\neg\, e \wedge \Delta e)$

Proof. For (i):

$\sigma[\![\Delta \exists x \cdot e]\!]\mathfrak{tt}$
$\quad \Leftrightarrow$ expand using cases for Δ
$\exists i \cdot \sigma^i[\![e]\!]\mathfrak{tt} \vee \forall i \cdot \sigma^i[\![e]\!]\mathfrak{ff}$

$\sigma[\![\forall x \cdot \Delta e \vee \exists x \cdot (e \wedge \Delta e)]\!]\mathfrak{tt}$
$\quad \Leftrightarrow$ expand using cases for $\forall$
$\exists i \cdot (\sigma^i[\![e]\!]\mathfrak{tt} \wedge \sigma^i \in \mathbf{dom}\,[\![e]\!]) \vee \forall i \cdot \sigma^i \in \mathbf{dom}\,[\![e]\!]$

These expressions can be proved (e.g. in a natural deduction style) to be equivalent.

The $\mathfrak{ff}$ case is argued in the same way. (Step-by-step expansions and full Natural Deduction proofs are contained in Appendix A of this paper.)

For (ii), Definition 2 is key to the argument. □

The following result for predicate LPF formulae follows.

Theorem 8. Every LPF formula can be converted into an equivalent formula in PNF.

Proof. This follows by Theorem 5, Lemmas 6 and 7. □

4.3 Skolemisation

Once an expression is in PNF, any existential quantifiers it contains can be removed by *Skolemisation*: each existentially quantified variable is replaced by (a reference to) a *Skolem function*.[5] Skolemisation creates a formula which is equi-satisfiable with the original formula. The Skolemised expression cannot be logically equivalent to the original one because they depend on different states/interpretations.

As is standard, Skolem functions depend on any embracing universally quantified variables; thus: $\exists x \cdot x = 42$ is changed to $(\lambda() \cdot 42)() = 42$ and $\forall x \cdot \exists y \cdot x + y = 42$ is changed to $\forall x \cdot x + (\lambda y \cdot 42 - y)(x) = 42$. The following theorem shows that the Skolemisation procedure can be applied in LPF. The result is essentially as in classical logic with the explicit requirement that the Skolem functions are total (defined for all defined arguments).

Theorem 9. Given some expression S, an expression S' created by Skolemisation will be equi-satisfiable with the original S.

Proof. Without loss of generality, the discussion is couched in terms of $S = \forall x \cdot \exists y \cdot e(x, y)$; for which Skolemisation yields $S' = \forall x \cdot e(x, f(x))$ with f being an unused name with which a total function is associated.

$$\begin{array}{l} \exists \sigma \cdot \sigma[\![\forall x \cdot \exists y \cdot e(x,y)]\!]\mathfrak{tt} \\ \quad \Leftrightarrow \text{ expanding using cases for } \forall, \exists \\ \exists \sigma \cdot \forall i \cdot \exists j \cdot (\sigma \dagger \{x \mapsto i, y \mapsto j\})[\![e(x,y)]\!]\mathfrak{tt} \end{array}$$

With $f \in \mathbf{dom}\, \sigma^f$:

$$\begin{array}{l} \exists \sigma^f \cdot \forall x \cdot e(x, f(x)) \\ \quad \Leftrightarrow \text{ expanding using case for } \forall \\ \exists \sigma^f \cdot \forall i \cdot ((\sigma \cup \{f \mapsto \lambda x \cdot \ldots\}) \dagger \{x \mapsto i\})[\![e(x, f(x))]\!]\mathfrak{tt} \end{array}$$

The *Axiom of Choice* [17, §3.6] guarantees that f can be associated with an appropriate result for any specific argument x. □

After Skolemising a PNF formula, the matrix of the formula can then be converted into CNF and any universal quantifiers can be removed since all free variables are assumed to be implicitly universally quantified in LPF. The resulting formula can then be directly represented in clausal form.

Theorem 10. Every closed LPF formula can be converted into an LPF formula in clausal form which is equi-satisfiable.

Proof. This is an immediate consequence of Theorem 8, Theorem 9, Theorem 5 and Lemma 3. □

[5] Some texts distinguish *Skolem constants* but these are just functions with zero parameters.

In order to reduce the size of formulae represented in clausal form various *absorption properties*, such as $p \wedge (p \vee q) \equiv p$, are used — as Lemma 3 indicates, they hold in LPF.

5 Refutation and Resolution for LPF

This section investigates the application of refutation procedures and resolution to LPF. These proof procedures are aimed at supporting or refuting the validity of a useful proportion of sequents. Refutation converts the issue of sequent satisfaction into a validity question; resolution offers a proof procedure for validity; and unification enlarges the scope of resolvents. There are many extensions to the basic procedures for standard first-order logic some of which are discussed in Section 6.

5.1 Refutation Procedures for LPF

In two-valued classical logic, a formula e is valid iff $\neg\, e$ is unsatisfiable and hence the validity of a formula can be proved by refuting its negation. The above result is important since it means that, in classical logic, a proof procedure for satisfiability can be used as a *refutation procedure* for checking validity [4]. Furthermore, the same approach can be used to check logical consequence [4, 6]: let $\Gamma = \{e_1, \ldots, e_n\}$ be a set of formulae, then, in classical logic, $\Gamma \models e$ holds iff the formula

$$e_1 \wedge \ldots \wedge e_n \wedge \neg\, e \tag{2}$$

is unsatisfiable. This follows in classical logic since any formula must evaluate to one of two truth values.

(Note that the formulae in Γ are normally assumed to be consistent, i.e. there exists an interpretation that makes all of the formulae in Γ true.)

The application of a refutation procedure in LPF is complicated by the fact that formulae might not be defined in all interpretations and this breaks the duality between validity and satisfiability: if $\neg\, e$ is unsatisfiable in LPF, it might evaluate to either false or undefined for any interpretation; it is therefore not possible to infer that e is valid since any interpretation making $\neg\, e$ undefined will also make e undefined. Note that it is still true in LPF that if $\neg\, e$ is satisfiable then e cannot be valid.

The same issue arises with using a refutation procedure to check a logical consequence in LPF:
i) If $e_1 \wedge \ldots \wedge e_n \wedge \neg\, e$ is satisfiable then it is clear that (as in classical logic) $\Gamma \not\models e$ in LPF;
ii) However, if $e_1 \wedge \ldots \wedge e_n \wedge \neg\, e$ is unsatisfiable, it cannot be inferred that $\Gamma \models e$ since there may exist interpretations in LPF in which $e_1, \ldots, e_n$ are all true but e is undefined (e.g. $\models i \div 0 = 1 \vee \neg\,(i \div 0 = 1)$).

One way forward is to note that, in order to show $\Gamma \models e$ holds in LPF, it is possible to both refute the false case (as in classical logic) and also to refute the undefined ("gap") case. So, if unsatisfiable is returned from the initial refutation, the definedness of the formula e under Γ is also checked (i.e. $\Gamma \models \Delta e$). Note that no circularity is introduced by this additional refutation proof since Δe must evaluate to a defined value. (An optimisation is explored in Section 6.2.)

Theorem 11. Let $\Gamma = \{e_1, \ldots, e_n\}$ be a set of formulae. Then $\Gamma \models e$ iff both $e_1 \wedge \ldots \wedge e_n \wedge \neg\, e$ and $e_1 \wedge \ldots \wedge e_n \wedge \neg\, \Delta e$ are unsatisfiable.

Proof. $\Rightarrow$ Suppose $\Gamma \models e$. Then every interpretation σ making all the formulae in Γ true must result in $\sigma[\![e]\!]\mathfrak{tt}$. Therefore, by the definition of $[\![\text{-}]\!]$, it must follow that $\sigma[\![e_1 \wedge \ldots \wedge e_n \wedge \neg\, e]\!]\mathfrak{ff}$ and $\sigma[\![\Delta e]\!]\mathfrak{tt}$ as required.

$\Leftarrow$ Suppose i) $e_1 \wedge \ldots \wedge e_n \wedge \neg\, e$ is unsatisfiable and ii) $\Gamma \models \Delta e$. Then by assumption ii), $\sigma[\![\Delta e]\!]\mathfrak{tt}$ holds for every interpretation σ which makes all the formulae in Γ true. Therefore by the definition of $[\![\Delta e]\!]$ it follows that for any such σ either $\sigma[\![e]\!]\mathfrak{tt}$ or $\sigma[\![e]\!]\mathfrak{ff}$. However, by assumption i), $(\sigma, \mathfrak{tt}) \notin [\![\neg\, e]\!]$ and so by the definition of $[\![\neg\, e]\!]$ it follows that $\sigma[\![e]\!]\mathfrak{tt}$ as required. □

In order to automate the check $\Gamma \models \Delta e$, the refutation procedure being employed needs to be extended appropriately to handle the Δ operator. Such an extension for resolution is developed in the next section.

5.2 Resolution for Propositional LPF

Resolution [33, 4, 17] is a proof procedure for checking the satisfiability of a set of clauses which is widely used as a refutation procedure. The basic idea behind resolution is to find clauses containing contradictory literals (e.g. literals of the form l and $\neg\, l$) and then *resolve* these to form a new clause. This generalises *modus ponens* to the following *(ground) resolution rule*:

$$\boxed{\text{resolve}}\ \frac{l \vee e_1;\ \neg\, l \vee e_2}{e_1 \vee e_2}$$

Not only does this remain valid in LPF but it is also true that:

$$\boxed{\text{resolve}\Delta}\ \frac{l \vee e_1;\ \neg\, \Delta l \vee e_2}{e_1 \vee e_2}$$

and similarly for $\neg\, l \vee e_1$. In the following, "syntactic clash" is defined as the pairs $(l, \neg\, l)$, $(l, \neg\, \Delta l)$, $(\neg\, l, \neg\, \Delta l)$.

Resolution works on clauses by repeatedly applying the resolution rule to a set of clauses, each time adding the newly derived resolved clause to the current set of clauses. (Each pair of potentially clashing clauses is examined only once.)

In classical propositional logic, the resolvent of two clauses is satisfiable if the two clauses are simultaneously satisfiable. The following extends this to propositional LPF.

Theorem 12. Let C_1 and C_2 be two propositional LPF clauses that contain a syntactic clash. Let C_3 be the resolvent clause formed by using the resolution rule to remove the clashing clauses. If C_1 and C_2 are true in some interpretation σ then then C_3 is also true in σ.

Proof. As a representative syntactic clash, consider the example of $\{l\} \subseteq C_1$ and $\{\neg\, l\} \subseteq C_2$, l is a literal, $C_3 = (C_1 \setminus \{l\}) \cup (C_2 \setminus \{\neg\, l\})$. For an arbitrary interpretation $\sigma \in \Sigma$ in which C_1 and C_2 evaluate to true, there are three cases to consider:

1. $\sigma[\![l]\!]\mathfrak{tt}$: By the definition $\sigma[\![\neg\, l]\!]\mathfrak{ff}$; since by assumption some clause in C_2 is true in σ, there must exist another disjunct than $\neg\, l$ in C_2 such that $\sigma[\![l']\!]\mathfrak{tt}$. This ensures that a clause of C_3 evaluates to true in σ.

2. $\sigma[\![l]\!]\mathfrak{ff}$: This follows by a symmetrical argument to the preceding case.

3. $\sigma \notin \mathbf{dom}\,[\![l]\!]$: The argument is again similar except that there must be other satisfying clauses in both C_1 and C_2.

The argument for other syntactic clashes is similar. □

Remembering that a set of clauses corresponds to a conjunction, it follows that two syntactically contradicting single element clauses indicate that the whole set is unsatisfiable. In resolution, such a step is said to yield an "empty clause" and the following lemma is trivial.

Lemma 13. If the empty clause is derived when applying the resolution rule to a set of clauses S, then S is unsatisfiable.

Proof. Without loss of generality, take as a representative case two clauses: $\{\neg\, \Delta l\}$ and $\{\neg\, l\}$. From the definition given in Figure 3, it is immediate that there can be no σ such that $\sigma[\![\neg\, \Delta l]\!]\mathfrak{tt}$ and $\sigma[\![\neg\, l]\!]\mathfrak{tt}$. □

The following theorem confirms that resolution is a sound proof procedure for propositioal LPF.

Theorem 14. *(Soundness)* If the empty clause is derived by applying the ground resolution procedure to a set of clauses S, then S is unsatisfiable.

Proof. Follows from Theorem 12 and Lemma 13 by induction on the number of steps of the resolution procedure. □

Theorem 15. *(Completeness)* If a set of propositional LPF clauses is unsatisfiable then applying ground resolution procedure to S will result in the empty clause being derived.

Proof. Follows, for example, Ben Ari's proof [4, Th 4.23] based on semantic trees. □

5.3 General Resolution Procedure for LPF

As in the standard literature, a *substitution* is a map of variables to terms of the form:

$$\phi = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$$

where each x_i is a distinct variable and each t_i is an integer term. The application of a substitution ϕ to a term t, denoted $\phi[t]$, is the simultaneous replacement of each $x_i \in \mathbf{dom}\,\phi$ in t with the respective $\phi(x_i)$. In a slight abuse of notation, $\phi[C]$ is used to represent the application of a substitution ϕ to all the terms in a clause C. *Unification* [33, 4, 17, 7] is the process of finding a substitution that makes two terms identical, that is, finding whether there exists a substitution ϕ for the variables in two terms t_1 and t_2, such that:

$$\phi[t_1] = \phi[t_2]$$

If such a substitution exists then it is known as a *unifier* for t_1 and t_2. If two terms can be unified then they have a *most general unifier* (*mgu*), which is unique up to variable renaming. A mgu for two terms t_1 and t_2 is a unifier ϕ such that any other unifier ϕ' for t_1 and t_2 can be derived by composing ϕ with a further substitution ϕ''.

For predicate logic, the *general resolution* procedure [4] uses unification 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. This is captured by the *general resolution rule*: Let C_1 and C_2 be two clauses such that $l_1 \in C_1$ and $\neg\, l_2 \in C_2$; Then if the literals l_1 and l_2 have an mgu ϕ then the two clauses C_1 and C_2 can be resolved to the new resolvent clause

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

Given a pair of literals of the form $(l_1, \neg\, l_2)$, $(l_1, \neg\, \Delta l_2)$ or $(\neg\, l_1, \neg\, \Delta l_2)$ then they are said to represent a *(unification) syntactic clash* iff the underlying literals l_1 and l_2 can be unified.

Formalising the above rule in LPF by extending it to syntactic clashes needs care since unification may substitute variables, which range over only defined values, for terms which have undefined

interpretations. To illustrate the potential problems with unification in LPF, consider that if a clause $p(t)$ holds in an interpretation then $p(\phi[t])$ must also hold for any substitution ϕ in classical logic. This property requires qualification in LPF since a substitution may introduce an undefined term into the clause resulting in $p(\phi[t])$ being undefined.

Given the above problem with unification, the general resolution rule as stated above can produce unexpected results. For example, consider the two clauses $C_1 = \{p(x, f(0)), q(x)\}$ and $C_2 = \{p(g(0), y), r(y)\}$, and the resolvent clause $C_3 = \{q(g(0)), r(f(0))\}$, derived using mgu $\phi = \{x \mapsto g(0), y \mapsto f(0)\}$. Suppose C_1 and C_2 are true in an interpretation σ in which the terms $f(0)$ and $g(0)$ are undefined. Then it must be true that the literals $q(x)$ and $r(y)$ are true. However, the resolvent C_3 cannot be true in σ, it must be undefined given the assumption of strictness. This highlights again the problem with unification; the literals $q(x)$ and $r(y)$ are implicitly universally quantified and are true in σ for all well-defined integer values. However, unification allows the variables to be substituted by undefined terms thus forcing the literals to become undefined.

A slightly different problem can be observed when cancelling using literals of the form $\neg\, \Delta l$. Consider the two clauses $\{p(x), q(y)\}$ and $\{\neg\, \Delta(p(f(0)), r(z)\}$. Applying the general resolution rule using the mgu $\phi = \{x \mapsto f(0)\}$ will result in the resolvent clause $\{q(y), r(z)\}$. However, note that in an interpretation σ where $f(0)$ is undefined it is possible for both the clauses $p(x)$ and $\neg\, \Delta(p(f(0))$ to be true (i.e. they no longer clash). Therefore, in such an interpretation there is no guarantee that the resolvent clause $\{q(y), r(z)\}$ will be true.

To address the above issues additional *unification constraints* need to be included in the resolvent clause to take account of the possibility of undefined terms. These unification constraints make use of a definedness operator ($t \in \mathbb{Z}$) that indicates if a term t represents a defined (integer) value. More formally, given a term t the definedness operator ($t \in \mathbb{Z}$) is defined by

$$[\![(t \in \mathbb{Z})]\!] = \{(\sigma, \mathfrak{tt}) \mid \sigma \in \mathbf{dom}\, [\![t]\!]\} \cup \{(\sigma, \mathfrak{ff}) \mid \sigma \in \Sigma \wedge_{\mathbb{B}} \sigma \notin \mathbf{dom}\, [\![t]\!]\}$$

In the example above, the clauses $\{p(x), q(y)\}$ and $\{\neg\, \Delta(p(f(0)), r(z)\}$ are now resolved to the clause $\{q(y), r(z), (f(0) \in \mathbb{Z})\}$. Thus, if the term $f(0)$ is interpreted as being undefined in σ then the resolvent clause will still be true in σ. Such additional unification constraints will need to be resolved with definedness conditions, as illustrated in step 12 of the resolution trace example for Property 1 given in Figure 4 (and, incidentally, foreshadowed in step 1.3.1 of the natural deduction proof in Figure 2).

The above idea can be formalised by an *LPF resolution rule* as follows. Let C_1 and C_2 be clauses and suppose $l_1 \in C_1$ and $l_2 \in C_2$ such that (l_1, l_2) is a syntactic clash under an mgu $\phi = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$. Then C_1 and C_2 can be resolved to a new clause

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

where $\theta = \{\neg\, (t_1 \in \mathbb{Z}), \ldots, \neg\, (t_n \in \mathbb{Z})\}$ is a set of unification constraints derived from the mgu ϕ.

The following result shows that the new LPF resolution rule preserves satisfiability.

Theorem 16. Let C_1 and C_2 be two LPF clauses that contain a (unification) syntactic clash. Let C_3 be the resolvent clause formed by using the resolution rule to remove the clashing clauses. If C_1 and C_2 are true in some interpretation σ then then C_3 is also true in σ.

Proof. As a representative syntactic clash, consider the example of $p(t_1) \in C_1$ and $\neg\, p(t_2) \in C_2$ such that t_1 and t_2 unify with an mgu ϕ. Then they resolve to the following clause:

$$C_3 = (\phi[C_1] \setminus \phi[\{p(t_1)\}]) \cup (\phi[C_2] \setminus \phi[\{\neg\, p(t_2)\}]) \cup \theta$$

For an arbitrary interpretation σ in which $\sigma[\![C_1]\!]\mathfrak{tt}$ and $\sigma[\![C_2]\!]\mathfrak{tt}$, there are two cases to consider. (Note in a slight abuse of notation the clausal form representation is used below in which variables are implicitly universally quantified.)

Case 1: $\sigma[\![p(t_1)]\!]\mathfrak{tt}$: It follows that $(\sigma, \mathfrak{tt}) \notin [\![\neg\, p(t_2)]\!]$ and so $\sigma[\![C_2 \setminus \{\neg\, p(t_2)\}]\!]\mathfrak{tt}$ must hold. Given that applying the substitution ϕ allows for the introduction of undefined terms there are two subcases to consider:
i) $\sigma[\![\phi[C_2 \setminus \{\neg\, p(t_2)\}]]\!]\mathfrak{tt}$: Then clearly $\sigma[\![C_3]\!]\mathfrak{tt}$ holds for the resolvent clause.
ii) $\sigma \notin \mathbf{dom}\, [\![\phi[C_2 \setminus \{\neg\, p(t_2)\}]]\!]$: Then the substitution ϕ must have introduced an undefined term and so there must exist a unification constraint $\theta_i \in \theta$ such that $\sigma[\![\theta_i]\!]\mathfrak{tt}$. It therefore follows that $\sigma[\![C_3]\!]\mathfrak{tt}$.

Case 2: $\sigma[\![p(t_1)]\!]\mathfrak{ff}$ or $\sigma \notin \mathbf{dom}\, [\![p(t_1)]\!]$: This follows along similar lines to the preceding case but is based on the possible values of the term $\phi[C_1] \setminus \phi[\{p(t_1)\}]$.

The arguments for the other syntactic clashes follow by similar case analyses. □

The general resolution rule can be shown to be sound for LPF. Again, soundness means that deriving the empty clause when applying the general resolution procedure indicates that the original set of clauses is not simultaneously satisfiable.

Theorem 17. *(Soundness)* If the empty clause is derived when applying general resolution to a set of clauses S, then S is not simultaneously satisfiable.

Proof. The proof uses Theorem 16 and follows similar lines to Theorem 12. □

A procedure called *factoring* [4, 7] is required alongside the resolution procedure to ensure that unifiable literals that occur in a single clause are merged. Care is again needed when formulating a factoring rule in LPF due to the problems with unification in LPF as discussed above.

Lemma 18. Let C be a clause containing two literals $\{l_1, l_2\} \subseteq C$ such that l_1 and l_2 can be unified by an mgu ϕ. If C is true in an interpretation σ then $\phi[C \setminus \{l_2\}] \cup \theta$ is also true in σ, where θ is the set of unification constraints derived from ϕ.

Proof. By case analysis along similar lines to the proof of Theorem 16. □

Soundness of the modified resolution procedure is crucial; as indicated in Section 1, research is on-going to re-establish resolution completeness for LPF. The key point is that the addition of the definedness operator ($t \in \mathbb{Z}$) coupled with the assumption of strictness of functions and predicates means that further opportunities for resolving occur that are not covered by the current resolution rules. To illustrate the problem, consider the clauses $\{p(f(x))\}$ and $\{\neg\, (f(0) \in \mathbb{Z})\}$; they are not simultaneously satisfiable but we cannot derive the empty clause from them. What appears to be needed is a new resolution rule that allows such clashes to be resolved. Work is ongoing to formulate such rules and thus derive completeness for general resolution in LPF.

5.4 Illustrative Examples (resumed)

Consider again the earlier counter example that $\models p \vee \neg\, p$ does not hold in LPF. Resolution as part of a refutation procedure yields the empty clause (unsatisfiability); but for LPF, it is also necessary to prove that $\models \Delta e$ holds to be able to infer that $\models e$ holds. In the modified LPF clausal form, the negation $\neg\, \Delta(p \vee \neg\, p)$ is represented –after simplification– as the unit set containing

1	$\forall i \cdot i = 0 \ \Rightarrow \ \neg (i - 1 = 0)$	*assumption*
2	$\forall i \cdot \neg (i = 0) \ \Rightarrow \ i \div i = 1$	*assumption*
3	$\forall i, j \cdot (i - j) \in \mathbb{Z}$	*assumption*
4	$\forall i \cdot (i \div i = 1) \vee ((i - 1) \div (i - 1) = 1)$	*goal*
5	$\{\neg (i = 0), \neg (i - 1 = 0)\}$	$clausal\_form(1)$
6	$\{i = 0, i \div i = 1\}$	$clausal\_form(2)$
7	$\{(i - j) \in \mathbb{Z}\}$	$clausal\_form(3)$
8	$\{\neg (c \div c = 1)\}$	$clausal\_form(\neg 4)$
9	$\{\neg ((c - 1) \div (c - 1) = 1)\}$	$clausal\_form(\neg 4)$
10	$\{c = 0\}$	$resolve(6, 8)$
11	$\{c - 1 = 0\} \vee \neg ((i - 1) \in \mathbb{Z})$	$resolve(6, 9)$
12	$\{c - 1 = 0\}$	$resolve(11, 7)$
13	$\{\neg (c - 1 = 0)\}$	$resolve(5, 10)$
14	empty clause	$resolve(12, 13)$

Figure 4: An illustrative proof of Property 1 using resolution as part of a refutation procedure.

the clause $\{\neg \Delta p\}$ 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 of Property 1, an example proof of this property using a refutation procedure is given in Figure 4 (where c is a Skolem constant). This resolution proof makes use of unification as needed.

A proof that the goal is defined is similar but the next section indicates that there is a more efficient procedure.

6 Conclusions and Further Work

6.1 Summary

LPF is a logic designed for reasoning about logical formulae that can include partial terms. This paper considers applying the fundamental proof procedures of resolution and refutation procedures to LPF; it identifies potential pitfalls that arise in doing so and outlines extensions and modifications that are required to carry these techniques over to LPF. Illustrative proofs are provided which are based upon a semantic definition of LPF.

Since LPF retains properties such as the commutativity and distributivity, the clausal form of classical logic conversions carries over to LPF. The definedness operator Δ in LPF, however, results in the need for extra conversion rules. This has the undesirable result of leading to more expensive clausal form formulae — fortunately the use of Δ is constrained.

The concept of resolution carries over from the classical case to LPF when considering satisfiability. However, the use of a refutation procedure in LPF brings about extra overhead due to the presence of "gaps". An LPF refutation procedure requires that definedness of the consequent needs to be established. There are, however, optimisations available.

Much, of course, remains to be done. It would be tempting to initiate work on mechanising the modified procedures described in this paper but the authors intend to resolve some other issues before they start programming. The current investigation of the fundamental proof procedures needs to be extended to consider the many optimisations that have arisen over the years. Equally important is experimentation: Schmalz's thesis [35] sets an admirable example in using genuine industrial benchmarks for checking performance.

6.2 Optimisations

With respect to the proof procedures presented in this paper, there is an optimisation described in [28] that shows Δ proofs are only needed in the case where resolution is between clauses from the conclusions of sequents. Similarly, [28] discusses ways of reducing the overhead from unification constraints (e.g. by noting the Skolem functions are total).

As has been made clear in the introduction, this paper explores only the basic (but fundamental) proof procedures. Numerous other heuristic techniques have been developed to improve the efficiency of the resolution procedure — the interested reader is referred to [37, 11]. Some of the known optimisations are addressed in [28]. For example, the Davis-Putnam procedure is tackled in [28, Lemmas 28/29] and an optimisation for PNF on page 214 of the same thesis.

Also of considerable importance is support for equality. In Section 5 the symbol for equality is not constrained to match the semantics for some particular notion of equality. The equality symbol is just a binary predicate that could be interpreted arbitrarily. The obvious approach to handling the equality relational operator in first-order predicate logic is to add axioms stating that equality is *reflexive*, *symmetric* and *transitive* as well as axioms that assert the *congruence* ($\forall x \cdot \forall y \cdot x = y \Rightarrow f(x) = f(y)$) of each function and predicate used. Given such axioms, resolution can be used to solve first-order logic problems with equality. This approach is, however, inefficient since it leads to an explosion in the number of clauses required. The standard approaches –including paramodulation [32, 2, 17]– and their applicability in an LPF context are considered in [28].

6.3 Related work

A mechanisation of Kleene logic for partial functions is given in [25]. 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 as well as a reduction in the size of the resulting clausal form of a formula (when using Δ).

Matthias Schmalz's ETH thesis [35] provides great insight into the mechanisation of the logic of Event-B. As mentioned above, he uncovered a number of unsoundnesses in the RodinTools and rebuilt a sound foundation in Isabelle. Interestingly, he views Event-B as having a 3-valued logic. Other important contributions include "directed rewriting" and the link made to interpreting sequents in an "SW" semantics (cf. [31]). Schmalz's new "unlifting" process is far more efficient than its predecessors but still experiences major expansion.

Acknowledgements

The authors would like to thank Matthias Schmalz, Gordon Plotkin and Arnon Avron for helpful discussions on work related to the topic of this paper and the anonymous referees who offered valuable clarifying comments on an earlier version of this paper. The authors 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 (this last supported the contact with ETH).

References

[1] Arnon Avron. Classical gentzen-type methods in propositional many-valued logics. In *Multiple-Valued Logic*. IEEE Computer Society, 2001.

[2] Leo Bachmair and Harald Ganzinger. Resolution theorem proving. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 2, pages 19–100. Elsevier Science B.V., 2001.

[3] H. Barringer, J.H. Cheng, and C. B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21:251–269, 1984.

[4] Mordechai Ben-Ari. *Mathematical Logic for Computer Science.* Springer, 3 edition, 2012.

[5] 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.

[6] Juan C. Bicarregui and Brian M. Matthews. Proof and refutation in formal software development. In *3rd Irish Workshop on Formal Software Development*, 1999.

[7] Alan Bundy. *The Computer Modelling of Mathematical Reasoning.* Academic Press, 1983.

[8] J. H. Cheng. *A Logic for Partial Functions.* PhD thesis, University of Manchester, 1986.

[9] 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 [10].

[10] 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.

[11] Roberto Cordeschi. The role of heuristics in automated theorem proving. j.a. robinson's resolution principle. *Mathware and Soft Computing*, 3:281–293, 1996.

[12] William M. Farmer. A partial functions version of Church's simple theory of types. *Journal of Symbolic Logic*, 55(3):1269–1291, 1990.

[13] William M. Farmer. Mechanizing the traditional approach to partial functions. In M. Kohlhase W. Farmer, M. Kerber, editor, *Proceedings of the Workshop on the Mechanization of Partial Functions*, pages 27–32, 1996.

[14] William M. Farmer and Joshua D. Guttman. A set theory with support for partial functions. *Studia Logica*, 66:59–78, 2000.

[15] 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.

[16] David Gries and Fred B. Schneider. 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.

[17] John Harrison. *Handbook of Practical Logic and Automated Reasoning.* Cambridge University Press, 2009.

[18] C. B. Jones. *Systematic Software Development using VDM.* Prentice Hall International, second edition, 1990.

[19] C. B. Jones, K. D. Jones, P. A. Lindsay, and R. Moore. *mural: A Formal Development Support System.* Springer-Verlag, 1991.

[20] C. B. Jones and M. J. Lovert. Semantic models for a logic of partial functions. *IJSI*, 5:55–76, 2011.

[21] C. B. Jones, M. J. Lovert, and L. J. Steggles. A semantic analysis of logics that cope with partial functions. In John Derrick et al., editors, *ABZ 2012*, volume 7316 of *Lecture Notes in Computer Science*, pages 252–265, June 2012.

[22] C. B. Jones, M. J. Lovert, and L. J. Steggles. Towards a mechanisation of a logic that copes with partial terms. Technical Report CS-TR-1314, Newcastle University, February 2012.

[23] C.B. Jones and C.A. Middelburg. A typed logic of partial functions reconstructed classically. *Acta Informatica*, 31(5):399–430, 1994.

[24] 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.

[25] 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.

[26] S. C. Kleene. *Introduction to Metamathematics*. Van Nostrad, 1952.

[27] 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.

[28] Matthew J. Lovert. *On the Mechanisation of the Logic of Partial Functions*. PhD thesis, Newcastle University, 2013.

[29] 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.

[30] O. Müller and K. Slind. Treating partiality in a logic of total functions. *The Computer Journal*, 40(10):640–652, 1997.

[31] 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.

[32] 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.

[33] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12:23–41, 1965.

[34] 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.

[35] Matthias Schmalz. *Formalizing the Logic of Event-B: Partial Functions, Definitional Extensions, and Automated Theorem Proving*. PhD thesis, ETH Zurich, 2012.

[36] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

[37] 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.

A Details of Proofs

A.1 Lemma 3

This section provides some illustrative proofs for Lemma 3 based on expanding $[\![e]\!]$.

Detailed proof of Lemma 3.(i). $e \equiv \neg\neg e$, for any formula e.

$\sigma[\![\neg\neg e]\!]\mathfrak{tt}$
$\quad\Leftrightarrow$ expand using cases for $\neg$
$\sigma[\![\neg e]\!]\mathfrak{ff}$
$\quad\Leftrightarrow$ expand using cases for $\neg$
$\sigma[\![e]\!]\mathfrak{tt}$

$\sigma[\![\neg\neg e]\!]\mathfrak{ff}$
$\quad\Leftrightarrow$ expand using cases for $\neg$
$\sigma[\![\neg e]\!]\mathfrak{tt}$
$\quad\Leftrightarrow$ expand using cases for $\neg$
$\sigma[\![e]\!]\mathfrak{ff}$

□

Detailed proof of Lemma 3.(ii). $\neg(e_1 \vee e_2) \equiv (\neg e_1) \wedge (\neg e_2)$

$\sigma[\![\neg(e_1 \vee e_2)]\!]\mathfrak{tt}$
$\quad\Leftrightarrow$ expand using cases for $\neg$
$\sigma[\![e_1 \vee e_2]\!]\mathfrak{ff}$
$\quad\Leftrightarrow$ expand using cases for $\vee$
$\sigma[\![e_1]\!]\mathfrak{ff} \wedge_{\mathbb{B}} \sigma[\![e_2]\!]\mathfrak{ff}$
$\quad\Leftrightarrow$ using definition of $\neg$
$\sigma[\![\neg e_1]\!]\mathfrak{tt} \wedge_{\mathbb{B}} \sigma[\![\neg e_2]\!]\mathfrak{tt}$
$\quad\Leftrightarrow$ using definition of $\wedge$ (Definition 2)
$\sigma[\![(\neg e_1) \wedge (\neg e_2)]\!]\mathfrak{tt}$

$\sigma[\![\neg(e_1 \vee e_2)]\!]\mathfrak{ff}$
$\quad\Leftrightarrow$ expand using cases for $\neg$
$\sigma[\![e_1 \vee e_2]\!]\mathfrak{tt}$
$\quad\Leftrightarrow$ expand using cases for $\vee$
$\sigma[\![e_1]\!]\mathfrak{tt} \vee_{\mathbb{B}} \sigma[\![e_2]\!]\mathfrak{tt}$
$\quad\Leftrightarrow$ using definition of $\neg$
$\sigma[\![\neg e_1]\!]\mathfrak{ff} \vee_{\mathbb{B}} \sigma[\![\neg e_2]\!]\mathfrak{ff}$
$\quad\Leftrightarrow$ using definition of $\wedge$ (Definition 2)
$\sigma[\![(\neg e_1) \wedge (\neg e_2)]\!]\mathfrak{ff}$

□

A.2 Lemma 7

Detailed proof of Lemma 7. Re (i) $\Delta(\exists x \cdot e) \equiv \forall x \cdot \Delta e \vee \exists x \cdot (e \wedge \Delta e)$
Using the abbreviation $\sigma^i = \sigma \dagger \{x \mapsto i\}$ we expand as follows:

$\sigma[\![\Delta \exists x \cdot e]\!]\mathtt{tt}$
$\quad \Leftrightarrow$ expand using cases for Δ
$\sigma \in \mathbf{dom}\,[\![\exists x \cdot e]\!]$
$\quad \Leftrightarrow$ expand using cases for $\exists$
$\exists i \cdot \sigma^i[\![e]\!]\mathtt{tt} \vee \forall i \cdot \sigma^i[\![e]\!]\mathtt{ff}$

$\sigma[\![\forall x \cdot \Delta e \vee \exists x \cdot (e \wedge \Delta e)]\!]\mathtt{tt}$
$\quad \Leftrightarrow$ expand using cases for $\vee$ and commute
$\sigma[\![\exists x \cdot (e \wedge \Delta e)]\!]\mathtt{tt} \vee \sigma[\![\forall x \cdot \Delta e]\!]\mathtt{tt}$
$\quad \Leftrightarrow$ expand using cases for $\exists$ and $\forall$
$\exists i \cdot (\sigma^i[\![e \wedge \Delta e]\!]\mathtt{tt}) \vee \forall i \cdot \sigma^i[\![\Delta e]\!]\mathtt{tt}$
$\quad \Leftrightarrow$ expand using cases for $\wedge$
$\exists i \cdot (\sigma^i[\![e]\!]\mathtt{tt} \wedge \sigma^i[\![\Delta e]\!]\mathtt{tt}) \vee \forall i \cdot \sigma^i[\![\Delta e]\!]\mathtt{tt}$
$\quad \Leftrightarrow$ expand using cases for Δ (twice)
$\exists i \cdot (\sigma^i[\![e]\!]\mathtt{tt} \wedge \sigma^i \in \mathbf{dom}\,[\![e]\!]) \vee \forall i \cdot \sigma^i \in \mathbf{dom}\,[\![e]\!]$

The two expressions can be proved equivalent, a (pair of) pedantic natural deduction proof(s) would be:

	from $\exists i \cdot \sigma^i[\![e]\!]\mathtt{tt} \vee \forall i \cdot \sigma^i[\![e]\!]\mathtt{ff}$	
1	**from** $\exists i \cdot \sigma^i[\![e]\!]\mathtt{tt}$	
1.1	$\exists i \cdot \sigma^i[\![e]\!]\mathtt{tt} \wedge \sigma^i \in \mathbf{dom}\,[\![e]\!]$	
	infer $\exists i \cdot (\sigma^i[\![e]\!]\mathtt{tt} \wedge \sigma^i \in \mathbf{dom}\,[\![e]\!]) \vee \forall i \cdot \sigma^i \in \mathbf{dom}\,[\![e]\!]$	$\vee$-I(1.1)
2	**from** $\forall i \cdot \sigma^i[\![e]\!]\mathtt{ff}$	
	infer $\exists i \cdot (\sigma^i[\![e]\!]\mathtt{tt} \wedge \sigma^i \in \mathbf{dom}\,[\![e]\!]) \vee \forall i \cdot \sigma^i \in \mathbf{dom}\,[\![e]\!]$	$\vee$-I(h2)
	infer $\exists i \cdot (\sigma^i[\![e]\!]\mathtt{tt} \wedge \sigma^i \in \mathbf{dom}\,[\![e]\!]) \vee \forall i \cdot \sigma^i \in \mathbf{dom}\,[\![e]\!]$	$\vee$-E(h,1,2)

	from $\exists i \cdot (\sigma^i[\![e]\!]\mathtt{tt} \wedge \sigma^i \in \mathbf{dom}\,[\![e]\!]) \vee \forall i \cdot \sigma^i \in \mathbf{dom}\,[\![e]\!]$	
1	**from** $\exists i \cdot (\sigma^i[\![e]\!]\mathtt{tt} \wedge \sigma^i \in \mathbf{dom}\,[\![e]\!])$	
1.1	$\exists i \cdot \sigma^i[\![e]\!]\mathtt{tt}$	$\wedge$-Er(h1)
	infer $\exists i \cdot \sigma^i[\![e]\!]\mathtt{tt} \vee \forall i \cdot \sigma^i[\![e]\!]\mathtt{ff}$	$\vee$-I(1.1)
2	**from** $\forall i \cdot \sigma^i \in \mathbf{dom}\,[\![e]\!]$	
	infer $\exists i \cdot \sigma^i[\![e]\!]\mathtt{tt} \vee \forall i \cdot \sigma^i[\![e]\!]\mathtt{ff}$	$\mathbf{dom}\,[\![e]\!] = \mathbb{B}$
	infer $\exists i \cdot \sigma^i[\![e]\!]\mathtt{tt} \vee \forall i \cdot \sigma^i[\![e]\!]\mathtt{ff}$	$\vee$-E(h,1,2)

And:

$\sigma[\![\Delta \exists x \cdot e]\!]\mathit{ff}$
$\quad \Leftrightarrow$ expand using cases for Δ
$\sigma \notin \mathbf{dom}\,[\![\exists x \cdot e]\!]$
$\quad \Leftrightarrow$ expand using cases for $\exists$
$\neg\, \exists i \cdot \sigma^i[\![e]\!]\mathit{tt} \wedge \neg\, \forall i \cdot \sigma^i[\![e]\!]\mathit{ff}$
$\quad \Leftrightarrow$ de Morgan (twice)
$\forall i \cdot \neg\, \sigma^i[\![e]\!]\mathit{tt} \wedge \exists i \cdot \neg\, \sigma^i[\![e]\!]\mathit{ff}$

$\sigma[\![\forall x \cdot \Delta e \vee \exists x \cdot (e \wedge \Delta e)]\!]\mathit{ff}$
$\quad \Leftrightarrow$ expand using cases for $\vee$ (and commute)
$\sigma[\![\exists x \cdot e \wedge \Delta e]\!]\mathit{ff} \wedge \sigma[\![\forall x \cdot \Delta e]\!]\mathit{ff}$
$\quad \Leftrightarrow$ expand using cases for $\exists$ and $\forall$
$\forall i \cdot \sigma^i[\![e \wedge \Delta e]\!]\mathit{ff} \wedge \exists i \cdot \sigma^i[\![\Delta e]\!]\mathit{ff}$
$\quad \Leftrightarrow$ expand using cases for $\wedge$
$\forall i \cdot (\sigma^i[\![e]\!]\mathit{ff} \vee \sigma^i[\![\Delta e]\!]\mathit{ff}) \wedge \exists i \cdot \sigma^i[\![\Delta e]\!]\mathit{ff}$
$\quad \Leftrightarrow$ expand using cases for Δ (twice)
$\forall i \cdot (\sigma^i[\![e]\!]\mathit{ff} \vee \sigma^i \notin \mathbf{dom}\,[\![e]\!]) \wedge \exists i \cdot \sigma^i \notin \mathbf{dom}\,[\![e]\!]$

The two expressions can be proved equivalent, a (pair of) pedantic natural deduction proof(s) would be:

	from $\forall i \cdot \neg\, \sigma^i[\![e]\!]\mathit{ff} \wedge \exists i \cdot \neg\, \sigma^i[\![e]\!]\mathit{ff}$	
1	$\quad \forall i \cdot \neg\, \sigma^i[\![e]\!]\mathit{ff}$	$\wedge$-El(h)
2	$\quad \exists i \cdot \neg\, \sigma^i[\![e]\!]\mathit{ff}$	$\wedge$-Er(h)
3	$\quad \forall i \cdot \sigma^i[\![e]\!]b \;\Rightarrow\; b \in \mathbb{B}$	$[\![e]\!]$
4	$\quad \forall i \cdot \sigma^i[\![e]\!]\mathit{ff} \vee \sigma^i \notin \mathbf{dom}\,[\![e]\!]$	1, 3
5	$\quad \exists i \cdot \sigma^i \notin \mathbf{dom}\,[\![e]\!]$	1, 2
	infer $\forall i \cdot (\sigma^i[\![e]\!]\mathit{ff} \vee \sigma^i \notin \mathbf{dom}\,[\![e]\!]) \wedge \exists i \cdot \sigma^i \notin \mathbf{dom}\,[\![e]\!]$	$\wedge$-I(4,5)

	from $\forall i \cdot (\sigma^i[\![e]\!]\mathit{ff} \vee \sigma^i \notin \mathbf{dom}\,[\![e]\!]) \wedge \exists i \cdot \sigma^i \notin \mathbf{dom}\,[\![e]\!]$	
1	$\quad \forall i \cdot \sigma^i[\![e]\!]\mathit{ff} \vee \sigma^i \notin \mathbf{dom}\,[\![e]\!]$	$\wedge$-El(h)
2	$\quad \exists i \cdot \sigma^i \notin \mathbf{dom}\,[\![e]\!]$	$\wedge$-Er(h)
3	$\quad \forall i \cdot \neg\, \sigma^i[\![e]\!]\mathit{tt}$	1
4	$\quad \exists i \cdot \neg\, \sigma^i[\![e]\!]\mathit{ff}$	2
	infer $\forall i \cdot \neg\, \sigma^i[\![e]\!]\mathit{ff} \wedge \exists i \cdot \neg\, \sigma^i[\![e]\!]\mathit{ff}$	$\wedge$-I(3, 4)

Re (ii) $\Delta(\forall x \cdot e) \equiv \forall x \cdot \Delta e \vee \exists x \cdot (\neg\, e \wedge \Delta e)$
Follows from the abbreviation

$$
\begin{aligned}
&[\![\Delta \forall x \cdot e]\!] \\
&\quad = [\![\Delta \neg\, \exists x \cdot \neg\, e]\!] \\
&\quad = [\![\Delta \exists x \cdot \neg\, e]\!] \\
&\quad = [\![\forall x \cdot \Delta \neg\, e \vee \exists x \cdot (\neg\, e \wedge \Delta \neg\, e)]\!] \\
&\quad = [\![\forall x \cdot \Delta e \vee \exists x \cdot (\neg\, e \wedge \Delta e)]\!]
\end{aligned}
$$

□