COMPUTING SCIENCE

Semantic Models for a Logic of Partial Functions

Cliff B. Jones and Matthew J. Lovert

TECHNICAL REPORT SERIES

No. CS-TR-1220 **September, 2010**

TECHNICAL REPORT SERIES

No. CS-TR-1220 **September, 2010**

Semantic Models for a Logic of Partial Functions

C. B. Jones, M. J. Lovert

Abstract

The Logic of Partial Functions (LPF) is used to reason about propositions that include terms that can fail to denote values. This paper provides semantics for LPF. A Structural Operational Semantics (SOS) provides an intuitive introduction; this is followed by a denotational semantics where the space of denotations is relations which provide an intuitive model of undefined terms. Finally, we illustrate how the denotational semantics can be used as a basis for proofs about propositions that include terms that can fail to denote.

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

Bibliographical details

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

Semantic Models for a Logic of Partial Functions
[By] C. B. Jones and M. J. Lovert
Newcastle upon Tyne: University of Newcastle upon Tyne: Computing Science, 2010.

(University of Newcastle upon Tyne, Computing Science, Technical Report Series, No. CS-TR-1220)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE
Computing Science. Technical Report Series. CS-TR-1220

Abstract

The Logic of Partial Functions (LPF) is used to reason about propositions that include terms that can fail to denote values. This paper provides semantics for LPF. A Structural Operational Semantics (SOS) provides an intuitive introduction; this is followed by a denotational semantics where the space of denotations is relations which provide an intuitive model of undefined terms. Finally, we illustrate how the denotational semantics can be used as a basis for proofs about propositions that include terms that can fail to denote.

About the authors

Cliff Jones is a Professor of Computing Science at Newcastle University. 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 now PI of the follow-on *Platform Grant* "Trustworthy Ambient Systems" (TrAmS) (also EPSRC). He is also PI on an EPSRC-funded project "Splitting (Software) Atoms Safely" and coordinates the "Methodology" strand of the EU-funded RODIN project. As well as his academic career, Cliff has spent over twenty years in industry. His fifteen 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 (and enjoyed the family atmosphere of Wolfson College). 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). 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.

Suggested keywords

NON-DENOTING TERMS
LOGIC OF PARTIAL FUNCTIONS
STRUCTURAL OPERATIONAL SEMANTICS
DENOTATIONAL SEMANTICS

Semantic Models for a Logic of Partial Functions

Cliff B. Jones and Matthew J. Lovert

School of Computing Science, Newcastle University, NE1 7RU, UK
{cliff.jones,matthew.lovert}@ncl.ac.uk

Abstract. The *Logic of Partial Functions (LPF)* is used to reason about propositions that include terms that can fail to denote values. This paper provides semantics for LPF. A *Structural Operational Semantics (SOS)* provides an intuitive introduction; this is followed by a *denotational semantics* where the space of denotations is relations which provide an intuitive model of undefined terms. Finally, we illustrate how the denotational semantics can be used as a basis for proofs about propositions that include terms that can fail to denote.

Keywords. Logic of Partial Functions; Non-denoting Terms; Structural Operational Semantics; Denotational Semantics

1 Introduction

Terms that can fail to denote proper values arise from partial operators such as head (of a list) and from applications of recursive functions; such terms occur frequently in program specifications [CJ91, Jon06, Fit07]. This raises the question of how reasoning about such terms can be conducted formally. To illustrate the issue, consider the following (deliberately partial over all integers) function [CJ91]:

$$zero : \mathbb{Z} \to \mathbb{Z}$$
$$zero(i) \triangleq \textbf{if } i = 0 \textbf{ then } 0 \textbf{ else } zero(i-1)$$

this function returns 0 when $i \geq 0$. However, when $i < 0$, $zero(i)$ will fail to denote a value of the expected type and thus a term such as $zero(-1)$ is referred to as a non-denoting (or "undefined") term. Now consider the following property of this function[1]:

$$\forall i \in \mathbb{Z} \cdot zero(i) = 0 \vee zero(-i) = 0 \tag{1}$$

[1] The function might look to be perversely partial but it –and Property 1– have been deliberately chosen to be as simple as possible to illustrate the issues around non-denoting terms (e.g. in Property 1, there is no obvious "guarding" predicate to be used on the left of an implication). In realistic applications, it is frequently difficult to spot the defined domain of a function: [CJ91,FJ08] use a function of two parameters where definedness depends on a relation between the two arguments.

A reasonable view of the *zero* function suggests that Property 1 is *true*; in particular, the disjunction is true for the least fixed point interpretation of the recursive definition of *zero*. It is clear that one of the disjuncts will fail to denote a value, with the exception of the case when $i = 0$, so the truth of Property 1 relies on the truth of disjunctions such as $zero(1) = 0 \vee zero(-1) = 0$ which reduces to[2] $0 = 0 \vee \bot_{\mathbb{Z}} = 0$. Since $zero(-1)$ does not, in the least-fixed point, denote an integer; with strict equality (undefined if either operand is undefined) this further reduces to $true \vee \bot_{\mathbb{B}}$ which makes no sense in FOPC.

Since we are interested in reasoning formally about such properties, we have decided to make use of a non-classical (three-valued) logic known as the *Logic of Partial Functions (LPF)* [BCJ84] which copes naturally with propositions over terms that can fail to denote. In LPF Property 1 is *true* and its proof presents no difficulty.

The objective for this paper is to provide semantics for LPF both through a *Structural Operational Semantics (SOS)* and a *denotational semantics*. Because decision procedures etc. can go underneath the notion of provability in the logic ($P \vdash Q$), it is important to fix the semantics of truth in a model ($P \models Q$). The second author of this paper is undertaking research on mechanised tool support for proofs involving non-denoting terms using LPF.

1.1 Outline

Section 2 provides a brief overview of approaches to coping with non-denoting terms which is followed by a detailed discussion on our preferred approach of LPF. Section 3 introduces the chosen expression constructs before presenting a *Structural Operational Semantics (SOS)* which defines their evaluation according to the semantics of LPF. Continuing with the semantics of LPF, Section 4 presents a *denotational semantics* which addresses shortfalls noted in Section 3.3. Section 5 illustrates how *proofs* about propositions that include terms that can fail to denote values can be based on our denotational semantics, and finally Section 6 highlights future work alongside some conclusions.

2 Approaches to Coping with Non-denoting Terms

When terms involve the application of partial functions and operators, they can fail to denote proper values. Over the years many different approaches have been suggested to handle non-denoting (undefined) terms; the reader is referred to [CJ90, CJ91, Jon06] for fuller surveys and citations to the original papers, but it is useful here to picture the range of options. Essentially, the issue is where undefinedness is "caught". For instance one could *insist* that all terms do denote something, for example, $zero(-1)$ should denote an arbitrary value in the range of the *zero* function, i.e. an arbitrary integer — pictorially:

[2] Where necessary we represent non-denoting terms as $\bot_{\mathbb{Z}}$ and non-denoting logical values as $\bot_{\mathbb{B}}$.

$$\forall i \in \mathbb{Z} \cdot \overbrace{zero(i)}^{\in\mathbb{Z}} = 0 \vee \overbrace{zero(-i)}^{\in\mathbb{Z}} = 0$$

this, however, raises questions like whether it is *true* that $zero(-1) = zero(-1)$ and whether $zero(-1) = zero(-2)$.

$=_\exists$	0	1	2	...	$\bot_\mathbb{Z}$
0	**true**	**false**	**false**	...	**false**
1	**false**	**true**	**false**	...	**false**
2	**false**	**false**	**true**	...	**false**
...	...	...	...	...	...
$\bot_\mathbb{Z}$	**false**	**false**	**false**	...	**false**

Fig. 1. The truth table for existential equality with integer operands.

Another approach would be to accept that for example $zero(-1) = \bot_\mathbb{Z}$, and employ non-strict relational operators to bring the problem of non-denoting terms under control; writing $\mathbb{Z}_\bot = \mathbb{Z} \cup \bot_\mathbb{Z}$:

$$\forall i \in \mathbb{Z} \cdot \overbrace{\underbrace{zero(i)}_{\in\mathbb{Z}_\bot} =_\exists 0}^{\in\mathbb{B}} \vee \overbrace{\underbrace{zero(-i)}_{\in\mathbb{Z}_\bot} =_\exists 0}^{\in\mathbb{B}}$$

notice that "existential equality" has been used here: its truth table is given in Figure 1; it is non-strict and thus non-computational. Clearly, this is distinct from the weak equality that must be used in the computation of *zero*. (The truth table for weak equality shows $\bot_\mathbb{B}$ when either operand is $\bot_\mathbb{Z}$.) The key disadvantage of this approach is precisely that a user who is reasoning about partial functions and/or operators has to think about two notions of equality[3] and this extends to all relational operators.

Moving now to non-classical logics (considering again strict relational operators), non-denoting terms can be brought under control by having the logical operators cope with the arising non-denoting logical values (so, accepting that for example $zero(-1)$ is a non-denoting term – $\bot_\mathbb{Z}$ – and that $zero(-1) = 0$ is a non-denoting logical value – $\bot_\mathbb{B}$):

$$\forall i \in \mathbb{Z} \cdot \overbrace{\underbrace{zero(i) = 0}_{\in\mathbb{B}_\bot} \vee \underbrace{zero(i) = 0}_{\in\mathbb{B}_\bot}}^{\in\mathbb{B}}$$

this is exactly how LPF captures undefinedness. This is our preferred approach to handling non-denoting terms and we present a detailed discussion of this approach in Section 2.1.

McCarthy's conditional operators also gives rise to a non-classical logic. This approach imposes a left-to-right evaluation, since conditional expressions are

[3] An interesting link between variant notions of equality and LPF proofs is examined in [FJ08].

strict in their first argument, so while $\mathit{true} \vee \bot_{\mathbb{B}}$ and $\bot_{\mathbb{B}} \vee \mathit{true}$ are truths in LPF, the latter is not in McCarthy's approach [Jon06]. The conditional operator approach deals adequately with expressions where there is a form of "guard" as in Property 2:

$$\forall i \in \mathbb{Z} \cdot i \geq 0 \;\Rightarrow\; \mathit{zero}(i) = 0 \tag{2}$$

but conditional logical operators do not enjoy the familiar equivalences of classical logic; so, for example, disjunction is not commutative nor does the contrapositive of an implication hold. Thus it is not obvious how to handle the contrapositive of Property 2:

$$\forall i \in \mathbb{Z} \cdot \neg\,(\mathit{zero}(i) = 0) \;\Rightarrow\; i < 0 \tag{3}$$

Property 1 also poses a problem in this approach. It is also interesting to note that it would appear that undefined values corrupt quantified expressions such that:

$$\exists i \in \{-1..1\} \cdot \mathit{zero}(-i) = 0$$

does not have the same truth value as:

$$\mathit{zero}(1) = 0 \vee \mathit{zero}(0) = 0 \vee \mathit{zero}(-1) = 0$$

etc. All of these issues are resolved in LPF.

A final approach to the problem of non-denoting terms is to *prohibit* the use of partial functions and operators:

$$\overbrace{\forall i \in \mathbb{Z} \cdot \mathit{zero}(i) = 0 \vee \mathit{zero}(-i) = 0}^{\text{verboten}}$$

In addition the simple implication of Property 2 can, of course, be rewritten over a restricted set as: $\forall i \in \mathbb{N} \cdot \mathit{zero}(i) = 0$ but this approach does not provide a pleasant treatment for our key disjunction property, and restricting types becomes messy for functions of more than one argument where the required "dependant type" needs a predicate of all arguments [CJ91].

2.1 The Logic of Partial Functions (LPF)

The remainder of this paper is concerned with the approach known as LPF; which 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].

LPF[4] copes with undefinedness by accepting that where one (or both) operands of propositional operators fail to denote they will not yield one of the

[4] A brief history of LPF: Three-valued truth tables for the propositional operators are given in [Kle52]; Peter Aczel supervised Koletsos' research [Kol76] in which he gives a proof theory for such a logic; Cliff Jones suggested that Jen Cheng [Che86] apply this to programming tasks [BCJ84]; the typed version of LPF is covered in [JM94].

logical values (*true* or *false*); the interpretation of quantifiers is extended in a compatible way. A shorthand for talking about this is to say that there is a third logical value: *undefined* ($\bot_{\mathbb{B}}$), but –for reasons that become clear below– we prefer Blamey's notion of "gaps" in the value space [Bla80]. From this point forward, $\bot_{\mathbb{B}}$ and $\bot_{\mathbb{Z}}$ are to be understood as ways of representing "gaps".

The truth tables in Figure 2 (presented in [Kle52]) illustrate the way in which the propositional operators in LPF have been extended to handle logical values which may fail to denote (conjunction is defined via negation and disjunction in the normal way). These truth tables provide the strongest possible extension of the familiar propositional operators and can be viewed as "describing a parallel lazy evaluation of the operands" [Fit07]: a result is delivered as soon as possible, for example *true* $\vee \bot_{\mathbb{B}}$ evaluates to *true* and this result will still be valid (there will be no contradiction) even if the second (non-denoting) operand later evaluates to *true* or to *false*, (i.e. the evaluation later completes). This fits with our liking for Blamey's notion of undefinedness as "gaps". The issue of how to handle such "gaps" is central to the semantics given in both Sections 3 and 4.

$\vee$	**true**	$\bot_{\mathbb{B}}$	**false**
true	**true**	**true**	**true**
$\bot_{\mathbb{B}}$	**true**	$\bot_{\mathbb{B}}$	$\bot_{\mathbb{B}}$
false	**true**	$\bot_{\mathbb{B}}$	**false**

$\Rightarrow$	**true**	$\bot_{\mathbb{B}}$	**false**
true	**true**	$\bot_{\mathbb{B}}$	**false**
$\bot_{\mathbb{B}}$	**true**	$\bot_{\mathbb{B}}$	$\bot_{\mathbb{B}}$
false	**true**	**true**	**true**

$\neg$	
true	**false**
$\bot_{\mathbb{B}}$	$\bot_{\mathbb{B}}$
false	**true**

Fig. 2. The LPF truth tables for disjunction, implication, and negation.

It is also worthwhile noting that these propositional operators enjoy the familiar equivalences of classical logic; so, for example, disjunction is commutative and the contrapositive of an implication holds.

The quantifiers of LPF are a natural extension of the propositional operators — viewing existential quantification as a disjunction and universal quantification as a conjunction as is standard in FOPC. 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 no witness value can be shown. This follows *mutatis mutandis* with universally quantified expressions.

Considering again the *zero* partial function from Section 1, when $i < 0$ an application of this function can be thought of as denoting an undefined value of the appropriate type (say $\bot_{\mathbb{Z}}$) but we again prefer a notion of "gaps" in the value space. Formally, in LPF [JM94], one reasons about (the least fixed point interpretation of) recursive functions such as *zero* function using two inference rules that can be generated automatically from the recursive definition of *zero*:

$$\boxed{\textit{zero-base}}\ \frac{i = 0}{\textit{zero}(i) = 0}$$

$$\boxed{\textit{zero-step}}\ \frac{i \in \mathbb{Z};\ i \neq 0;\ \textit{zero}(i-1) = k}{\textit{zero}(i) = k}$$

it is important here to note that the equality used throughout these inference rules is "strict" — see Section 1.

In LPF, Property 2 of the *zero* function is *true* and is easily proved. In addition Property 1 is *true* in LPF and its proof is presented in Figure 3[5]. Moreover, Property 4 is also *true* in LPF:

from	$i \in \mathbb{Z}$	
1	$i \geq 0 \vee i < 0$	h1, $\mathbb{Z}$
2	**from** $i \geq 0$	
2.1	$\textit{zero}(i) = 0$	$\Rightarrow$-E-L(L, 2.h1)
	infer $\textit{zero}(i) = 0 \vee \textit{zero}(-i) = 0$	$\vee$-I-R(2.1)
3	**from** $i < 0$	
3.1	$-i \geq 0$	3.h1, $\mathbb{Z}$
3.2	$\textit{zero}(-i) = 0$	$\Rightarrow$-E-L(L, 3.1)
	infer $\textit{zero}(i) = 0 \vee \textit{zero}(-i) = 0$	$\vee$-I-L(3.2)
infer	$\textit{zero}(i) = 0 \vee \textit{zero}(-i) = 0$	$\vee$-E(1, 2, 3)

Fig. 3. Proof of the *zero* function disjunction Property 1.

$$\exists i \in \mathbb{Z} \cdot \textit{zero}(i) = 0 \tag{4}$$

Why is LPF not universally accepted? Clearly, there is a reluctance to adopt any non-classical logic. Specifically, one looks for those things that are unfamiliar in a non-standard logic. The only significant "surprise" in LPF is that the law of the excluded middle ($e \vee \neg\, e$) does not hold because the disjunction of two undefined Boolean values is still undefined — so $\textit{zero}(-1) = 0 \vee \neg\, (\textit{zero}(-1) = 0)$ is not considered to be a tautology.

For expressive completeness, a defined (δ) operator has been introduced into LPF: $\delta(e)$ returns *true* if e is defined (it is *true* or it is *false*). This gives LPF the deductive power of classical logic for defined expressions. In actual proofs about programs, the δ operator is rarely needed except that, due to the loss of the law of the excluded middle, the unrestricted deduction theorem:

$$\frac{e_1 \vdash e_2}{e_1 \ \Rightarrow\ e_2}$$

[5] Where the L used in this proof refers to the implication lemma of $i \geq 0 \ \Rightarrow\ \textit{zero}(i) = 0$, where i is an integer, and whose proof also follows with little difficulty. The inference rules used in this proof are the standard inference rules for LPF presented in [BFL$^+$94].

does not hold in LPF; a version of the deduction theorem that does hold in LPF adds an extra hypothesis stating that e_1 is defined:

$$\boxed{\Rightarrow\text{-}I}\ \frac{\delta(e_1);\, e_1 \vdash e_2}{e_1 \ \Rightarrow\ e_2}$$

In fact, the most weighty argument against the adoption of LPF is the body of research and engineering that has created automatic tools for classical logic. This is precisely why the second author of this paper is researching mechanised proof support tools for LPF.

3 Structural Operational Semantics (SOS)

The first semantic formalisation approach that we use to provide the semantics for LPF is an SOS specification (introduced by Gordon Plotkin [Plo81]). Our SOS specification provides an intuitive introduction to the semantics of LPF and how LPF addresses the issues of handling propositions that can include terms that fail to denote values. We feel it is beneficial to provide such a formalisation as doing so allows us to be clear about the semantics of the logic before we begin with a mechanisation of it. Additionally such a specification will provide a means of checking whether any mechanisation we implement is correct.

Before we introduce the rules which formalise the semantics of LPF for expression evaluation we first introduce the expression constructs that we are interested in providing the semantics for. In order to serve the stated purposes, it is clear that we need to present a language that includes quantified expressions and ways of introducing non-denoting terms — for instance through recursive functions.

3.1 Introducing Our Language

Our basic language includes logical expression constructs, where all of our expressions must be of the type Bool or of the type Int. This restriction is made to be able to simplify the semantic rules that follow but at the same time even with just these two types we can adequately describe the issues encountered with non-denoting terms.

A constant value is itself an expression. Other expressions in our language include using a valid identifier, arithmetic expressions ($-$ and $\div$), a relational (equality) expression, a conditional expression[6], propositional logic expressions (disjunction, negation, and the defined operator — δ)[7], quantified expressions (universal and existential)[8] and a function invocation expression.

[6] Useful for recursive function definitions.

[7] Conjunction and implication follow in virtually the same way as disjunction in the semantic rules that follow and as a result we do not present them in this paper. The same argument applies to why we only present the relational (equality) operator and not relational operators such as $>$, $\geq$, etc.

[8] All expressions have to be explicitly closed by quantifiers. In addition we only consider quantification over integers for simplicity.

A function in our language always takes a (single) integer argument and returns an integer result[9]; a function definition thus contains a parameter name and a resulting expression (an expression — that might include recursive calls to the function). Such functions have no free variables; the free variable of a result expressions can only be the parameter. A function invocation requires the name of the function and an argument to pass to the function.

For a function invocation expression we need to be able to access the called function in our semantic rules. To do this we create a map entitled Γ from function names to the functions themselves (VDM notation [Jon90] is used):

$$\Gamma = \mathit{Id} \xrightarrow{m} \mathit{Func}$$

where Γ is the set of all possible functions, and γ ($\gamma \subseteq \Gamma$) is used to represent a specific set of functions.

We are interested in ruling out ill-formed expressions so that we do not have to provide any semantics to expressions such as $\mathit{true} - 1$ and $\mathit{true} \vee 1$. In order to be able to perform type checks in our language we consider a map called *Types* that maps variable identifiers to the type of data that they can store.

$$\mathit{Type} = \textsc{Bool} \mid \textsc{Int}$$

$$\mathit{Types} = \mathit{Id} \xrightarrow{m} \mathit{Type}$$

We only intend to provide semantics for those expressions which are well-formed and thus satisfy the following criteria:

- A constant expression (of the type Bool or Int) is well-formed
- An identifier is well-formed if it is in the domain of a given *Types* map, and thus has an appropriate type (Bool or Int)
- An arithmetic expression is well-formed if both of its operands are well-formed and are both of the type Int, and the operator is either $-$ or $\div$
- A relational (equality) expression is well-formed if both of its operands are well-formed and of the type Bool
- Propositional logic expressions (disjunction, negation, and δ) are well-formed if all of their operands are well-formed and of the type Bool
- A conditional expression is well-formed if the expression condition is well-formed and of the type Bool, and the *true* and *false* expressions are both well-formed and of the type Int
- Quantification expressions (universal and existential) are well-formed if the quantified expression is well-formed and of the type Bool when the quantified variable is included in a given *Types* map and is constrained to be of the type Int

[9] We have chosen to simplify the semantics by limiting functions to the one argument, and constraining the parameter type and the functions return type to integers — this is of course trivial to change. This restricted form of function definitions still adequately allows us to highlight where the issues which we are faced with occur when reasoning about propositions over terms that can fail to denote.

– A function invocation is well-formed if the argument is well-formed and of the type INT, and the function to be called exists in a given γ map

In addition we only consider well-formed functions, and we consider them to be well-formed if the result expression is of the same type as the return type (INT) of the function.

3.2 Semantic Rules

Having introduced our expression constructs and having ruled out ill-formed expressions and function definitions from further consideration, we can now move on to our primary task of defining the semantics of LPF. We begin with an SOS specification.

All expressions in our language that reduce to a constant value are defined. Such values cannot be reduced any further. The constant values present in our language are the Boolean values *true* and *false*, and the integers ($\cdots, -1, 0, 1, \cdots$). If an expression can be evaluated to a member of one of these two sets then it is fully evaluated (no more evaluation can occur) and we refer to this as the evaluated expression denoting a value. For instance, the expression 0 is denoting, the expression $zero(0)$ denotes 0 and is thus denoting, but while the argument of $zero(-1)$ is denoting such an expression can never denote a member of one of these two sets of values and thus this expression is not denoting — it is an undefined expression.

We also need to introduce a map –that we refer to as a memory store– which maps the variable identifiers to the values they store at runtime:

$$Value = \mathbb{B} \mid \mathbb{Z}$$

$$\Sigma = Id \xrightarrow{m} Value$$

where Σ is the set of all possible memory stores in our language and σ ($\sigma \in \Sigma$) is used to represent a specific memory store.

Our SOS specification is presented as a series of inference rules which define the valid expression evaluations that can occur for the expression constructs we are considering. This SOS provides an intuitive understanding but is itself problematic when it comes to the quantified expressions. These semantic rules define the LPF semantics for expression evaluation; for a comparison between the semantic rules that are required to define LPF and those that are required to define FOPC, the reader is referred to Appendix A.

The semantic relation that we use to model the process of expression evaluation is[10]:

$$\xrightarrow{e} : \mathcal{P}((Expr \times \Sigma) \times Expr)$$

[10] We do not include γ in our semantic relation until later when we define the semantics for function invocations. This is purely for simplicity since γ is not used directly in any of the earlier semantic rules which we present.

where required, we use $\xrightarrow{e}*$ for the reflexive, transitive closure of $\xrightarrow{e}$.

Our first semantic rule simply returns the value to which an identifier is mapped in a given memory store.

$$\boxed{\textit{Id-E}}\ \frac{id \in Id}{(id, \sigma) \xrightarrow{e} \sigma(id)}$$

The following semantic rules define the evaluation of arithmetic expressions. The operands a and b must be reduced as much as possible (to constant values) before a result can be returned, i.e. eliminating the operator from the expression. The reader should be aware that the choice of which rule is evaluated is non-deterministic; there is no notion of fairness in the SOS rules.

$$\boxed{\textit{Arith-L}}\ \frac{op \in \{-, \div\}; (a, \sigma) \xrightarrow{e} a'}{(a\ op\ b, \sigma) \xrightarrow{e} a'\ op\ b}$$

$$\boxed{\textit{Arith-R}}\ \frac{op \in \{-, \div\}; (b, \sigma) \xrightarrow{e} b'}{(a\ op\ b, \sigma) \xrightarrow{e} a\ op\ b'}$$

$$\boxed{\textit{Arith-E1}}\ \frac{a \in \mathbb{Z}; b \in \mathbb{Z}}{(a + b, \sigma) \xrightarrow{e} [\![-]\!](a, b)}$$

$$\boxed{\textit{Arith-E2}}\ \frac{a \in \mathbb{Z}; b \in \mathbb{Z}; b \neq 0}{(a \div b, \sigma) \xrightarrow{e} [\![\div]\!](a, b)}$$

where $[\![op]\!]$ provides a semantic function for the syntactic object op — thus $[\![op]\!]$ is the expected result from evaluating op on its two operands.

Non-denoting terms arise from the arithmetic semantics given above with expressions that reduce to something of the form $i \div 0$. Notice that the *Arith-E2* semantic rule is one of the places that gives rise to "gaps" in the value space.

The following set of semantic rules define weak (strict) equality [FJ08] which returns a result only if both operands denote values; otherwise, the relational (equality) expression will fail to denote a value of the expected type. The truth table for weak equality is discussed in Section 1.

$$\boxed{\textit{Equality-L}}\ \frac{(a, \sigma) \xrightarrow{e} a'}{(a = b, \sigma) \xrightarrow{e} a' = b}$$

$$\boxed{\textit{Equality-R}}\ \frac{(b, \sigma) \xrightarrow{e} b'}{(a = b, \sigma) \xrightarrow{e} a = b'}$$

$$\boxed{\textit{Equality-E}}\ \frac{a \in \mathbb{Z}; b \in \mathbb{Z}}{(a = b, \sigma) \xrightarrow{e} [\![=]\!](a, b)}$$

Considering the *Equality-E* semantic rule if both a and b are not fully reduced (evaluated) to integer values then a result from an equality expression in question

will never be returned thus making the equality expression non-denoting. The reader should now notice how non-denoting terms that are operands to such strict relational operators can lead to non-denoting logical values.

The semantic rules that we present all represent a small-step semantics unless stated otherwise. The small-step semantics allow for interleaving of steps in different expression branches as can be seen from the rules for the arithmetic operators and from the relational equality operator. It is less important to have such interleaving for the arithmetic operators and the strict relational equality operator as both operands must denote anyway, but it is important for logical operators such as disjunction since they have to cope with the "gaps" that can occur. This is because in LPF we can return a result even in the presence of "gaps" in operands, as long as there is enough information available from evaluating the other operand. For example, $\bot_{\mathbb{B}} \vee \mathit{true}$ can be evaluated to *true* even though the first operand has not been fully evaluated[11]. Considering the first operand of the previous example as containing a term that will never denote (e.g. arising from our function invocation e.g. $\mathit{zero}(-1) = 0$ — see later), without such interleaving being able to occur we may start to evaluate the first operand, and with a big-step semantics we could not stop evaluation without evaluating this operand to a constant Boolean value (which it will never denote). The following set of semantic rules illustrates the evaluation of the disjunction logical operator according to the truth table presented in Figure 2.

$$\boxed{\mathit{Or\text{-}L}}\ \frac{(a,\sigma) \xrightarrow{e} a'}{(a \vee b,\sigma) \xrightarrow{e} a' \vee b}$$

$$\boxed{\mathit{Or\text{-}R}}\ \frac{(b,\sigma) \xrightarrow{e} b'}{(a \vee b,\sigma) \xrightarrow{e} a \vee b'}$$

$$\boxed{\mathit{Or\text{-}E1}}\ \frac{}{(\mathbf{true} \vee b,\sigma) \xrightarrow{e} \mathbf{true}}$$

$$\boxed{\mathit{Or\text{-}E2}}\ \frac{}{(a \vee \mathbf{true},\sigma) \xrightarrow{e} \mathbf{true}}$$

$$\boxed{\mathit{Or\text{-}E3}}\ \frac{}{(\mathbf{false} \vee \mathbf{false},\sigma) \xrightarrow{e} \mathbf{false}}$$

The two rules *Or-E*1 and *Or-E*2 can be seen as "coping with gaps" in that they can return a value even if one of their operands fails to denote.

The choice of which rule is used is non-deterministic; there is no notion of fairness, so we have no control over which rule is used. Ideally we would like each operand to be evaluated in parallel and then have an elimination rule return a result once enough information is available from at least the one evaluated

[11] It could be that this operand could be fully evaluated or that this operand will fail to denote a value.

operand. Alternatively we could simulate this parallel evaluation by performing one evaluation step on the left hand operand and then one evaluation step on the right hand operand and then repeating this process until enough information is available to be able to apply an elimination rule (to complete the evaluation of a disjunction expression).

The fact that we have no control over when and what semantic rule is evaluated could be problematic. One may always evaluate the left hand operand and never the right hand operand. Alternatively one may evaluate the left hand operand to *true* and then try to evaluate the right hand operand continuously (with multiple applications of a rule), and this right hand operand may not denote (see function application later) and thus the disjunction expression may never denote a Boolean value. Additionally there are other similar evaluations that are possible with these rules that could cause no result to be returned even if a result could be returned according to our understanding of the LPF truth table for disjunction.

The following set of semantic rules defines the evaluation of the negation logical operator. If a is evaluated to *true* or to *false* then invert it, otherwise attempt to keep evaluating a to see if eventually it will become defined.

$$\boxed{\textit{Not-A}}\ \frac{(a,\sigma) \xrightarrow{e} a'}{(\neg a,\sigma) \xrightarrow{e} \neg a'}$$

$$\boxed{\textit{Not-E1}}\ \frac{}{(\neg \mathbf{true},\sigma) \xrightarrow{e} \mathbf{false}}$$

$$\boxed{\textit{Not-E2}}\ \frac{}{(\neg \mathbf{false},\sigma) \xrightarrow{e} \mathbf{true}}$$

The defined (δ) construct as mentioned earlier is defined to return *true* only if its argument is defined. Taking $\delta(a)$, if a can be evaluated to *true* or to *false* then return *true* as a is defined, otherwise a is not denoting (but we can attempt to continue to evaluate a to see if it will eventually denote a Boolean value). The evaluation of this construct is illustrated in the following set of semantic rules.

$$\boxed{\textit{Defined-A}}\ \frac{(a,\sigma) \xrightarrow{e} a'}{(\delta(a),\sigma) \xrightarrow{e} \delta(a')}$$

$$\boxed{\textit{Defined-E1}}\ \frac{}{(\delta(\mathbf{true}),\sigma) \xrightarrow{e} \mathbf{true}}$$

$$\boxed{\textit{Defined-E2}}\ \frac{}{(\delta(\mathbf{false}),\sigma) \xrightarrow{e} \mathbf{true}}$$

We now consider the task of defining the rules for the quantifier expressions[12]. We start with universal quantification. The following semantic rule states that

[12] Here we define the quantification rules using big-step semantics, whereas the rest of our semantic rules are defined using small-step semantics. The small-step semantics allow interleaving of the steps in different expression branches.

if there exists an integer i –which when applied to the expression e causes e to evaluate to *false*– then return *false*, even if the expression e fails to denote with certain values of i. Clearly the choice of the value for i is important.

$$\boxed{\textit{Forall-F}}\ \frac{\exists i \in \mathbb{Z} \cdot (e, \sigma \dagger \{t \mapsto i\}) \xrightarrow{e}* \mathbf{false}}{(\forall t \cdot e, \sigma) \xrightarrow{e} \mathbf{false}}$$

For the following semantic rule, it is necessary that for every integer substituted for i, e must evaluate to *true*.

$$\boxed{\textit{Forall-T}}\ \frac{\forall i \in \mathbb{Z} \cdot (e, \sigma \dagger \{t \mapsto i\}) \xrightarrow{e}* \mathbf{true}}{(\forall t \cdot e, \sigma) \xrightarrow{e} \mathbf{true}}$$

We are using quantifiers to define the quantifier above and this is the core issue resolved in Section 4. For now, think of the use of the existential quantifier above the line as shorthand for an infinite disjunction (using the LPF disjunction logical operator already introduced), and the use of the universal quantifier above the line as shorthand for an infinite conjunction (both over the set of integers).

The next set of semantic rules illustrates the evaluation of the existential quantifier.

$$\boxed{\textit{Exists-T}}\ \frac{\exists i \in \mathbb{Z} \cdot (e, \sigma \dagger \{t \mapsto i\}) \xrightarrow{e}* \mathbf{true}}{(\exists t \cdot e, \sigma) \xrightarrow{e} \mathbf{true}}$$

$$\boxed{\textit{Exists-F}}\ \frac{\forall i \in \mathbb{Z} \cdot (e, \sigma \dagger \{t \mapsto i\}) \xrightarrow{e}* \mathbf{false}}{(\exists t \cdot e, \sigma) \xrightarrow{e} \mathbf{false}}$$

Notice that both quantifiers can result in "gaps".

Later we introduce the semantic rules for the evaluation of function invocations. In order to be able to allow for recursive functions to be defined it is necessary to have a conditional expression.

$$\boxed{\textit{Cond-A}}\ \frac{(e, \sigma) \xrightarrow{e} e'}{(e\ ?\ t : s, \sigma) \xrightarrow{e} e'\ ?\ t : s}$$

$$\boxed{\textit{Cond-E1}}\ \frac{}{(\mathbf{true}\ ?\ t : s, \sigma) \xrightarrow{e} t}$$

$$\boxed{\textit{Cond-E2}}\ \frac{}{(\mathbf{false}\ ?\ t : s, \sigma) \xrightarrow{e} s}$$

The *Cond-A* semantic rule describes the small-step semantics for evaluating the condition expression in our conditional expression construct. If this expression can be evaluated to a Boolean value (the expression is defined), then one of our two elimination semantic rules (*Cond-E*1 or *Cond-E*2) can be applied — either simply replaces the conditional expression construct with the appropriate sub-expression (t or s).

Up until now non-denoting terms ("gaps") have only been able to be introduced through applying the division operator in the obvious way. We now present another way of introducing such "gaps" in the first place through the use of our function invocation expression. First we must update our semantic relation that we use to model the process of expression evaluation to include Γ. There is no way in our semantics to update Γ.

$$\xrightarrow{e} : \mathcal{P}((Expr \times \Sigma \times \Gamma) \times Expr)$$

The *FuncCall-A* semantic rule represents the small-step semantics for evaluating the argument expression to be passed to a function. This rule is to be utilised until the argument expression has been reduced to a constant value.

$$\boxed{FuncCall\text{-}A} \frac{(arg, \sigma, \gamma) \xrightarrow{e} arg'}{(id(arg), \sigma, \gamma) \xrightarrow{e} id(arg')}$$

Any argument used in a function invocation must denote, otherwise we do not evaluate the function. Once (if) the argument expression has been reduced to a constant value we can now attempt to evaluate the function's result expression.

$$\boxed{FuncCall\text{-}E} \frac{arg \in \mathbb{Z}; (\gamma(id).result, \sigma \dagger \{\gamma(id).param \mapsto arg\}, \gamma) \xrightarrow{e} * res}{(id(arg), \sigma, \gamma) \xrightarrow{e} res}$$

Notice how this last rule represents a big step semantics in that the result of the function is computed in one go. Ideally we would like a small-step semantics to allow for the possibility of interleaving of steps in different expression branches. We now modify the *FuncCall-E* semantic rule to allow for such circumstances[13].

$$\boxed{FuncCall\text{-}E} \frac{arg \in \mathbb{Z}}{(id(arg), \sigma, \gamma) \xrightarrow{e} FuncInter(\gamma(id).result, \gamma(id).param, arg)}$$

Here we make use of another expression construct that combines the necessary information from a function invocation and from the function being called itself. The new expression construct *FuncInter* (which represents a function invocation under evaluation) contains the result expression from the function (e.g. a conditional expression) as well as the function's parameter identifier and the value passed into the function. This expression construct is used to allow for the current state of the result to be stored (alongside the parameter data) so that the evaluation can resume from where it left off previously if any interleaving of the steps in expression branches occurs.

The following two semantic rules define the rest of the small-step semantics for evaluating a function invocation. The first semantic rule is used to make a

[13] No change needs to be made to the *FuncCall-A* semantic rule since this already represents a small-step semantics.

further step in evaluating the result of the function each time it is applied[14]. The second semantic rule returns the result of the function invocation once (if) the function's result expression has been evaluated to an integer value.

$$\boxed{\textit{FuncInter-A}}\ \frac{(res, \sigma \dagger \{paramid \mapsto param\}, \gamma) \xrightarrow{e} res'}{(FuncInter(res, paramid, param), \sigma, \gamma) \xrightarrow{e} FuncInter(res', paramid, param)}$$

$$\boxed{\textit{FuncInter-E}}\ \frac{res \in \mathbb{Z}}{(FuncInter(res, paramid, param), \sigma, \gamma) \xrightarrow{e} res}$$

3.3 Discussion

The reader should have noticed that in the semantic rules we are using quantifiers to define quantifiers. This is not acceptable because if the meta-language interpretation of the quantifiers changes then so does the implied semantics. Section 4 provides an alternative to what has been presented in this section.

4 Set Theoretic Definition

This section carries the intuition of the foregoing SOS definition over to a denotational semantics by providing a set theoretic definition of the values that are denoted by expressions. Here the "gaps" that arise from partial terms and propositional expressions are handled 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 is directly prompted by thinking of the operational semantics of Section 3 as defining $\xrightarrow{e}$ as a relation. The choice of relational denotations is the essential difference that distinguishes what is done here from both [JM94] and [And99]. So:

$$\mathcal{E}\colon \mathcal{P}((Expr \times \Sigma \times \Gamma) \times Value)$$

which is defined here in parts:

$$\mathcal{E} = \mathcal{E}id \cup \mathcal{E}or \cup \mathcal{E}not \cup \mathcal{E}defined \cup \mathcal{E}equality \cup \mathcal{E}arith \cup \mathcal{E}cond \cup \mathcal{E}forall \cup \mathcal{E}exists \cup \mathcal{E}funccall$$

Access to named values presents no difficulties (because in Section 3.1 we ruled out any unknown names):

$$\mathcal{E}id = \{((v, \sigma, \gamma), \sigma(v)) \mid v \in Id\}$$

[14] The parameter is included in the memory store during the evaluation of the function's result, but notice that the updated memory store is not returned by the semantic rule. Only the updated result expression along with the parameter information to (possibly) be used to update the memory store in the same way later is returned by this semantic rule. This is to achieve the necessary variable scoping since interleaving of steps in different expression branches is allowed and is necessary for LPF.

The way in which "gaps" arise can be seen clearly for disjunctions (cf. *Or-E*1 and *Or-E*2 of Section 3.2):

$$\begin{array}{l}\mathcal{E}\,or = \\ \quad \{((a \vee b, \sigma, \gamma), \mathbf{true}) \mid ((a, \sigma, \gamma), \mathbf{true}) \in \mathcal{E}\} \cup \\ \quad \{((a \vee b, \sigma, \gamma), \mathbf{true}) \mid ((b, \sigma, \gamma), \mathbf{true}) \in \mathcal{E}\} \cup \\ \quad \{((a \vee b, \sigma, \gamma), \mathbf{false}) \mid ((a, \sigma, \gamma), \mathbf{false}) \in \mathcal{E} \wedge ((b, \sigma, \gamma), \mathbf{false}) \in \mathcal{E}\}\end{array}$$

Straightforwardly:

$$\begin{array}{l}\mathcal{E}\,not = \\ \quad \{((\neg a, \sigma, \gamma), \mathbf{false}) \mid ((a, \sigma, \gamma), \mathbf{true}) \in \mathcal{E}\} \cup \\ \quad \{((\neg b, \sigma, \gamma), \mathbf{true}) \mid ((b, \sigma, \gamma), \mathbf{false}) \in \mathcal{E}\}\end{array}$$

and:

$$\begin{array}{l}\mathcal{E}\,defined = \\ \quad \{((\delta(a), \sigma, \gamma), \mathbf{true}) \mid (a, \sigma, \gamma) \in \mathbf{dom}\,\mathcal{E}\}\end{array}$$

$$\begin{array}{l}\mathcal{E}\,cond = \\ \quad \{((e \ ?\ t :\ s, \sigma, \gamma), res) \mid ((e, \sigma, \gamma), \mathbf{true}) \in \mathcal{E} \wedge ((t, \sigma, \gamma), res) \in \mathcal{E}\} \cup \\ \quad \{((e \ ?\ t :\ s, \sigma, \gamma), res) \mid ((e, \sigma, \gamma), \mathbf{false}) \in \mathcal{E} \wedge ((s, \sigma, \gamma), res) \in \mathcal{E}\}\end{array}$$

$$\begin{array}{l}\mathcal{E}\,arith = \\ \quad \{((a\ op\ b, \sigma, \gamma), [\![op]\!](a', b')) \mid \\ \qquad (op \in \{-, \div\}) \wedge ((a, \sigma, \gamma), a') \in \mathcal{E} \wedge ((b, \sigma, \gamma), b') \in \mathcal{E}\}\end{array}$$

Remembering that weak equality is strict gives:

$$\begin{array}{l}\mathcal{E}\,equality = \\ \quad \{((a = b, \sigma, \gamma), [\![=]\!](a', b')) \mid ((a, \sigma, \gamma), a') \in \mathcal{E} \wedge ((b, \sigma, \gamma), b') \in \mathcal{E}\}\end{array}$$

The semantics for quantifiers needs to ensure that "gaps" are handled by non-denoting propositional expressions being absent from the domain of $\mathcal{E}$:

$$\begin{array}{l}\mathcal{E}\,forall = \\ \quad \{((\forall t \cdot e, \sigma, \gamma), \mathbf{true}) \mid \{((e, \sigma \dagger \{t \mapsto i\}, \gamma), \mathbf{true}) \mid i \in \mathbb{Z}\} \subseteq \mathcal{E}\} \cup \\ \quad \{((\forall t \cdot e, \sigma, \gamma), \mathbf{false}) \mid \mathbf{false} \in \mathbf{rng}\,(\{(e, \sigma \dagger \{t \mapsto i\}, \gamma) \mid i \in \mathbb{Z}\} \lhd \mathcal{E})\}\end{array}$$

$$\begin{array}{l}\mathcal{E}\,exists = \\ \quad \{((\exists t \cdot e, \sigma, \gamma), \mathbf{true}) \mid \mathbf{true} \in \mathbf{rng}\,(\{(e, \sigma \dagger \{t \mapsto i\}, \gamma) \mid i \in \mathbb{Z}\} \lhd \mathcal{E})\} \cup \\ \quad \{((\exists t \cdot e, \sigma, \gamma), \mathbf{false}) \mid \{((e, \sigma \dagger \{t \mapsto i\}, \gamma), \mathbf{false}) \mid i \in \mathbb{Z}\} \subseteq \mathcal{E}\}\end{array}$$

The definition of $\mathcal{E}\,funccall$ is recursive but the traditional least fixed point interpretation of partial functions in denotational semantics can be applied to relations so that $((zero(1), \sigma, \gamma), 0) \in \mathcal{E}$ but $(zero(-1), \sigma, \gamma) \notin \mathbf{dom}\,\mathcal{E}$.

Thus:

$$\begin{array}{l}\mathcal{E}\,funccall = \\ \quad \{((f(arg), \sigma, \gamma), res) \mid \\ \qquad ((arg, \sigma, \gamma), arg') \in \mathcal{E} \wedge \\ \qquad\qquad ((\gamma(f).result, \sigma \dagger \{\gamma(f).param \mapsto arg'\}, \gamma), res) \in \mathcal{E}\}\end{array}$$

5 Proofs

This section illustrates how the semantics of Section 4 can be used as a basis for direct proofs about logical expressions. This does not, of course, indicate that proofs about expressions should be conducted in terms of the denotational semantics — the natural deduction style of Figure 3 is far preferable; the interest is that having based the semantics on simple set theory, even direct proofs are straightforward.

It is first useful to record that the definition of $\mathcal{E}$ is deterministic:

Lemma 1 $((e, \sigma, \gamma), r_1) \in \mathcal{E} \wedge ((e, \sigma, \gamma), r_2) \in \mathcal{E} \;\Rightarrow\; r_2 = r_1$

this follows from the fact that there is exactly one rule for each type of *Expr* and that although $\mathcal{E}or$, $\mathcal{E}not$, $\mathcal{E}cond$, $\mathcal{E}forall$ and $\mathcal{E}exists$ are defined with set unions, the domains of the relations never overlap.

As an illustrative example, the universally quantified disjunction of Section 1 could have been used; but for brevity, rather than look at Property 1, it is assumed that recursive predicates could be added to our language — e.g.

$$pos : \mathbb{Z} \to \mathbb{B}$$
$$pos(i) \;\triangleq\; \textbf{if } i = 0 \textbf{ then true else } pos(i-1)$$

and the logical expression of interest is:

$$\forall i \in \mathbb{Z} \cdot pos(i) \vee pos(-i) \tag{5}$$

Property 5 has been chosen because it presents some difficulty to the other approaches discussed in Section 2, but as we have already mentioned poses little difficulty in LPF.

The following results follow easily from the denotational semantics. Firstly, the recursively defined predicate *pos* is defined (and delivers *true*) over the natural numbers.

Lemma 2 $\{((pos(i), \{i \mapsto n\}, \gamma), \textbf{true}) \mid n \in \mathbb{N}\} \subseteq \mathcal{E}$

This can be proved by a simple inductive argument:

$$((pos(i), \{i \mapsto 0\}, \gamma), \textbf{true}) \in \mathcal{E}$$

by $\mathcal{E}id$, $\mathcal{E}arith$, $\mathcal{E}cond$ and $\mathcal{E}funccall$; then:

$$\forall n \in \mathbb{N}_1 \cdot \\ ((pos(i), \{i \mapsto n-1\}, \gamma), \textbf{true}) \in \mathcal{E} \;\Rightarrow\; ((pos(i), \{i \mapsto n\}, \gamma), \textbf{true}) \in \mathcal{E}$$

follows from $\mathcal{E}cond$ and $\mathcal{E}funccall$. So, by induction:

$$\forall n \in \mathbb{N} \cdot ((pos(i), \{i \mapsto n\}, \gamma), \textbf{true}) \in \mathcal{E}$$

and the required lemma follows from the single valued property (Lemma 1).

From this it is easy to show that the disjunction $(pos(i) \vee pos(-i))$ is defined over all integers.

Lemma 3 $\{(((pos(i) \vee pos(-i)), \{i \mapsto m\}, \gamma), \mathbf{true}) \mid m \in \mathbb{Z}\} \subseteq \mathcal{E}$

Observe that Lemma 2 gives both:

$$\{((pos(i), \{i \mapsto n\}, \gamma), \mathbf{true}) \mid n \in \mathbb{N}\} \subseteq \mathcal{E}$$
$$\{((pos(i), \{i \mapsto -n\}, \gamma), \mathbf{true}) \mid n \in (\mathbb{Z} - \mathbb{N})\} \subseteq \mathcal{E}$$

and by $\mathcal{E}or$:

$$\begin{array}{l}\{(((pos(i) \vee pos(-i)), \{i \mapsto m\}, \gamma), \mathbf{true}) \mid m \in \mathbb{Z}\} = \\ \quad (\{((pos(i), \{i \mapsto n\}, \gamma), \mathbf{true}) \mid n \in \mathbb{N}\} \cup \\ \quad\quad \{((pos(i), \{i \mapsto -n\}, \gamma), \mathbf{true}) \mid n \in (\mathbb{Z} - \mathbb{N})\})\end{array}$$

This lets us conclude the required result about the universally quantified (over integers) expression.

Theorem 4 $\{(((\forall i \in \mathbb{Z} \cdot pos(i) \vee pos(-i)), \{\,\}, \gamma), \mathbf{true})\} \subseteq \mathcal{E}$

This is an immediate consequence of Lemma 3 and $\mathcal{E}forall$.

It would also be straightforward to prove:

$$p(42) \;\Rightarrow\; \exists i \in \mathbb{Z} \cdot p(i)$$

which is interesting because it ought follow immediately from an existential introduction but appears not to be safe in all logics (e.g. McCarthy's).

6 Conclusions

Over the course of this paper we have formalised the semantics of LPF for the evaluation of numerous expression constructs first using an SOS specification and then through the use of a denotational semantics, where the latter overcomes a problem that presented itself in our SOS specification. In addition we have illustrated how our denotational semantics definition can be used as a basis for proofs about propositions over terms that can fail to denote.

Non-denoting terms arise frequently in program specifications [CJ91, Jon06, Fit07] which raises the question of how proofs about such terms can be conducted formally. In addition since a large body of research and engineering has created tools for classical logic and approaches to coping with non-denoting terms having attracted much research over the years (e.g. [McC67,Bla80,BCJ84,Owe85,Che86, CJ91,Jon06,Kra06,Fit07]), we feel further research on mechanised proof support tools for LPF will be of significant benefit.

One of the reasons for carrying out this work was to provide a formalisation of a non-classical logic; in particular to formalise how LPF copes with propositions that may contain potentially non-denoting terms, for instance from the application of partial (recursive) functions and operators. Our formalisations

were created with the intent of mechanising support for proofs in LPF. The formalisations provided in this paper not only allow us to be more confident that we fully understand the semantics of LPF before we begin with a mechanisation, but they also provide a means of checking whether any mechanisation of LPF which we come up with is correct.

As in most cases there is much more work to be done. The major task ahead of us is mechanising support for proofs in LPF. This can be broken down into two separate subclasses of problems for further research. The first is to research how fundamental techniques such as *unification* and *resolution* work with such a non-classical logic. The second activity relates to actually implementing mechanised tool support for proofs about propositions over terms that can fail to denote values.

One way of implementing mechanised tool support for proofs in LPF would be to implement a theorem prover for LPF from scratch. However, we take the view that is also taken in Chapter 4 "Designing a Theorem Prover" of [AGM92] that it may be a better idea to try to captialise on an existing tool and build our extension on that. The current options that we see include extending an existing proof assistant (notably *Isabelle* [NPW02]) or adapting a term-rewriting tool (notably *Maude* [C$^+$07]) to include support for LPF. We think using a generic theorem proving assistant such as *Isabelle* would be an ideal choice of tool for us to extend to provide a logical framework for LPF. We also like the idea of using a term-rewriting system such as *Maude* due to the similarity between the inference rules (like those for the *zero* function presented in Section 2.1) and the term-rewriting rules.

Acknowledgements

The content of this paper has benefited greatly from discussions with Jason Steggles and Joey Coleman. The authors of this paper also gratefully acknowledge the funding for their research from an EPSRC PhD Studentship and EPSRC grants: TrAmS and AI4FM.

References

[AGM92] S. Abramsky, Dov M. Gabbay, and S. E. Maibaum, editors. *Handbook of logic in computer science (vol. 2): background: computational structures*. Oxford University Press, Inc., New York, NY, USA, 1992.

[And99] D. Andrews, editor. *A Formal Definition of VDM-SL*. Department of Mathematics & Computer Science, Leicester University, 1999.

[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.

[Bla80] S. R. Blamey. *Partial Valued Logic*. PhD thesis, Oxford University, 1980.

[C+07] Manuel Clavel et al., editors. *All about maude - a high-performance logical framework, how to specify, program and verify systems in rewriting logic*, volume 4350 of *LNCS*. Springer, 2007.

[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.

[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.

[FJ08] J. S. Fitzgerald and C. B. Jones. The connection between two ways of reasoning about partial functions. *IPL*, 107(3–4):128–132, 2008.

[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.

[Kle52] S. C. Kleene. *Introduction to Metamathematics*. Van Nostrad, 1952.

[Kol76] G. Koletsos. Sequent calculus and partial logic. Master's thesis, Manchester University, 1976.

[Kra06] Alexander Krauss. Partial recursive functions in higher-order logic. In *Int. Joint Conference on Automated Reasoning (IJCAR 2006), LNCS*, pages 589–603. Springer-Verlag, 2006.

[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.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A proof assistant for higher-order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[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.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Technical report, Aarhus University, 1981.

[Sto77] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

APPENDIX

A First Order Predicate Calculus (FOPC) Rules for Comparison

Here we present several semantic rules and then the denotational semantics which define the evaluation of our chosen expression constructs according to the semantics of FOPC. This is done for those expression constructs where the evaluation in FOPC differs from that of LPF. This affects the disjunction logical operator as well as the two quantification logical expression constructs.

Consider again the truth table for the disjunction logical operator in LPF and then compare this with the truth table for the disjunction logical operator in FOPC, presented in Figure 4. We note again that in LPF in certain circumstances we can return a result even if an operand is not yet fully evaluated or it fails to denote a value, as long as enough information is already available from evaluating the other operand. However, in FOPC there is a need for all operands (logical expressions) to be defined before a result can be returned. FOPC has no meaning for non-denoting logical values [Jon06, FJ08].

$\vee$ -*lpf*	**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**

$\vee$ -*fopc*	**true**	**false**
true	**true**	**true**
false	**true**	**false**

Fig. 4. A comparison of the LPF truth table and the FOPC truth table for disjunction.

The semantic relation that we use to model the process of expression evaluation is:

$$\xrightarrow{e} : \mathcal{P}((Expr \times \Sigma) \times Expr)$$

The modified semantic rules for the disjunction logical operator follow.

$$\boxed{FOPC\text{-}Or\text{-}L} \quad \frac{(a, \sigma) \xrightarrow{e} a'}{(mk\text{-}Or(a, b), \sigma) \xrightarrow{e} mk\text{-}Or(a', b)}$$

$$\boxed{FOPC\text{-}Or\text{-}R} \quad \frac{(b, \sigma) \xrightarrow{e} b'}{(mk\text{-}Or(a, b), \sigma) \xrightarrow{e} mk\text{-}Or(a, b')}$$

$$\boxed{FOPC\text{-}Or\text{-}E1} \quad \frac{}{(mk\text{-}Or(\mathbf{true}, \mathbf{true}), \sigma) \xrightarrow{e} \mathbf{true}}$$

$$\boxed{FOPC\text{-}Or\text{-}E2} \quad \frac{}{(mk\text{-}Or(\mathbf{true}, \mathbf{false}), \sigma) \xrightarrow{e} \mathbf{true}}$$

$$\boxed{\textit{FOPC-Or-E3}}\ \frac{}{(\textit{mk-Or}(\mathbf{false}, \mathbf{true}), \sigma) \xrightarrow{e} \mathbf{true}}$$

$$\boxed{\textit{FOPC-Or-E4}}\ \frac{}{(\textit{mk-Or}(\mathbf{false}, \mathbf{false}), \sigma) \xrightarrow{e} \mathbf{false}}$$

Above we have to make sure that before an elimination rule (*FOPC-Or-E*1 to *FOPC-Or-E*4) can be applied we have fully reduced (evaluated) each of the two operands to a constant (Boolean) value. This is problematic with non-denoting terms and one of our reasons for utilising LPF, since we are interested in reasoning about propositions that include terms that can fail to denote values.

We now turn our attention to the semantic rules which are required to define the evaluation of the quantification expression constructs in FOPC.

$$\boxed{\textit{FOPC-Forall-F}}\ \frac{\begin{array}{c}\forall j \in \mathbb{Z} \cdot ((e, \sigma \dagger \{t \mapsto j\}) \xrightarrow{e} \mathbf{true} \vee \\ (e, \sigma \dagger \{t \mapsto j\}) \xrightarrow{e} \mathbf{false}); \\ \exists i \in \mathbb{Z} \cdot (e, \sigma \dagger \{t \mapsto i\}) \xrightarrow{e} \mathbf{false}\end{array}}{(\forall t \cdot e, \sigma) \xrightarrow{e} \mathbf{false}}$$

$$\boxed{\textit{FOPC-Forall-T}}\ \frac{\forall i \in \mathbb{Z} \cdot (e, \sigma \dagger \{t \mapsto i\}) \xrightarrow{e} \mathbf{true}}{(\forall t \cdot e, \sigma) \xrightarrow{e} \mathbf{true}}$$

$$\boxed{\textit{FOPC-Exists-F}}\ \frac{\forall i \in \mathbb{Z} \cdot (e, \sigma \dagger \{t \mapsto i\}) \xrightarrow{e} \mathbf{false}}{(\exists t \cdot e, \sigma) \xrightarrow{e} \mathbf{false}}$$

$$\boxed{\textit{FOPC-Exists-T}}\ \frac{\begin{array}{c}\forall j \in \mathbb{Z} \cdot ((e, \sigma \dagger \{t \mapsto j\}) \xrightarrow{e} \mathbf{true} \vee \\ (e, \sigma \dagger \{t \mapsto j\}) \xrightarrow{e} \mathbf{false}); \\ \exists i \in \mathbb{Z} \cdot (e, \sigma \dagger \{t \mapsto i\}) \xrightarrow{e} \mathbf{true}\end{array}}{(\exists t \cdot e, \sigma) \xrightarrow{e} \mathbf{true}}$$

The difference between the evaluation of the quantified expressions in LPF and in FOPC reside in the *FOPC-Forall-F* and the *FOPC-Exists-T* rules. In FOPC we must show that the quantified expression is defined for every bound value (for every integer).

For the denotational semantics consider:

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

disjunction is modified to:

$$\begin{array}{l}\mathcal{E}\,or = \\ \quad \{((\textit{mk-Or}(a, b), \sigma, \gamma), \mathbf{true}) \mid \\ \qquad ((a, \sigma, \gamma), \mathbf{true}) \in \mathcal{E} \wedge ((b, \sigma, \gamma), \mathbf{true}) \in \mathcal{E}\} \cup \\ \quad \{((\textit{mk-Or}(a, b), \sigma, \gamma), \mathbf{true}) \mid \\ \qquad ((a, \sigma, \gamma), \mathbf{true}) \in \mathcal{E} \wedge ((b, \sigma, \gamma), \mathbf{false}) \in \mathcal{E}\} \cup \\ \quad \{((\textit{mk-Or}(a, b), \sigma, \gamma), \mathbf{true}) \mid \\ \qquad ((a, \sigma, \gamma), \mathbf{false}) \in \mathcal{E} \wedge ((b, \sigma, \gamma), \mathbf{true}) \in \mathcal{E}\} \cup \\ \quad \{((\textit{mk-Or}(a, b), \sigma, \gamma), \mathbf{false}) \mid \\ \qquad ((a, \sigma, \gamma), \mathbf{false}) \in \mathcal{E} \wedge ((b, \sigma, \gamma), \mathbf{false}) \in \mathcal{E}\}\end{array}$$

and the quantified expression cases are modified to:

$$
\begin{array}{l}
\mathcal{E}\mathit{forall} = \\
\quad \{((\forall t \cdot e, \sigma, \gamma), \mathbf{true}) \mid \\
\qquad \{((e, \sigma \dagger \{t \mapsto i\}, \gamma), \mathbf{true}) \mid i \in \mathbb{Z}\} \subseteq \mathcal{E}\} \cup \\
\quad \{((\forall t \cdot e, \sigma, \gamma), \mathbf{false}) \mid \\
\qquad \mathbf{false} \in \mathbf{rng}\,(\{(e, \sigma \dagger \{t \mapsto i\}, \gamma) \mid i \in \mathbb{Z}\} \lhd \mathcal{E}) \wedge \\
\qquad (\{(e, \sigma \dagger \{t \mapsto i\}, \gamma) \mid i \in \mathbb{Z}\} \subseteq \mathbf{dom}\,\mathcal{E})\}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}\mathit{exists} = \\
\quad \{((\exists t \cdot e, \sigma, \gamma), \mathbf{true}) \mid \\
\qquad \mathbf{true} \in \mathbf{rng}\,(\{(e, \sigma \dagger \{t \mapsto i\}, \gamma) \mid i \in \mathbb{Z}\} \lhd \mathcal{E}) \wedge \\
\qquad (\{(e, \sigma \dagger \{t \mapsto i\}, \gamma) \mid i \in \mathbb{Z}\} \subseteq \mathbf{dom}\,\mathcal{E})\} \cup \\
\quad \{((\exists t \cdot e, \sigma, \gamma), \mathbf{false}) \mid \\
\qquad \{((e, \sigma \dagger \{t \mapsto i\}, \gamma), \mathbf{false}) \mid i \in \mathbb{Z}\} \subseteq \mathcal{E}\}
\end{array}
$$