Int J Software Informatics, Volume 5, Issue 1-2 (2011), Part I, pp. 55–76
International Journal of Software and Informatics, ISSN 1673-7288

E-mail: ijsi@iscas.ac.cn
http://www.ijsi.org
Tel: +86-10-62661040

Semantic Models for a Logic of Partial Functions

Cliff B. Jones and Matthew J. Lovert

(School of Computing Science, Newcastle University, NE1 7RU, 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 two semantic descriptions for LPF. A *Structural Operational Semantics (SOS)* gives an intuitive introduction; this is followed by a *denotational semantics* where relations are chosen for denotations offering 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.

Key words: logic of partial functions; non-denoting terms; structural operational semantics; denotational semantics

Jones CB, Lovert MJ. Semantic models for a logic of partial functions. *Int J Software Informatics*, **Vol.5, No.1-2 (2011), Part I: 55–76. http://www.ijsi.org/1673-7288/5/i76.htm**

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[12,13,17]. 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 from Ref.[12]:

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

this function returns 0 when $i \geqslant 0$. 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)]:

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

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*. However it is clear that, with the exception of the case when $i = 0$,

Corresponding author: Cliff B. Jones, Email: cliff.jones@ncl.ac.uk
Received 2010-09-09; Accepted 2010-12-20; Final revised version 2011-01-17.

[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: Refs.[12,14] use a function of two parameters where definedness depends on a relation between the two arguments.

one of the disjuncts will fail to denote a value; so the truth of Property 1 relies on the truth of disjunctions such as $zero(1) = 0 \vee zero(-1) = 0$; since $zero(-1)$ does not –in the least-fixed point– denote an integer[2)], the property further reduces to $0 = 0 \vee \bot_{\mathbb{Z}} = 0$. With strict equality (undefined if either operand is undefined) this further reduces to $true \vee \bot_{\mathbb{B}}$ which makes no sense in classical logic since the truth tables only define propositional operators for proper values in $\mathbb{B}$.

There are a variety of approaches to 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)*[3] 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 second author of this paper is undertaking research on mechanised tool support for proofs in LPF involving non-denoting terms. Because decision procedures etc. can go underneath the notion of provability in a logic ($P \vdash Q$), it is important to fix the semantics of truth in a model ($P \models Q$). This paper provides semantics for LPF using both a *Structural Operational Semantics (SOS)* and a *denotational semantics*. The former gives an intuitive understanding of our approach to "gaps" but Section 3.3 indicates problems with making the SOS completely formal; these problems are overcome in the denotational semantics which is also used as a basis for representative proofs.

Section 2 provides a brief overview of approaches to coping with non-denoting terms which is followed by a detailed discussion 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 the 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 Refs.[10, 11, 12, 17, 21, 25, 27] 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:

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

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

Another approach is 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;

---

2) Both semantic descriptions below avoid the need for a representation of undefined but, where we wish to illustrate a point, we write non-denoting terms as $\bot_{\mathbb{Z}}$ and non-denoting logical values as $\bot_{\mathbb{B}}$.

writing $\mathbb{Z}_\perp = \mathbb{Z} \cup \perp_\mathbb{Z}$:

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

notice that "existential equality" has been used here: its truth table is given in Fig.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 $\perp_\mathbb{B}$ when either operand is $\perp_\mathbb{Z}$.) The key disadvantage of this approach is precisely that a user who is reasoning about partial functions and/or operators has to think about multiple notions of equality[3)] and this extends to all relational operators.

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

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

Resuming the assumption that relational operators are strict, non-denoting terms can be brought under control by having the logical operators cope with non-denoting logical values. Such approaches are then using non-classical logics. So, accepting that for example $zero(-1)$ is a non-denoting term and that $zero(-1) = 0$ is a non-denoting logical value, the following illustration shows how the non-classical logic LPF captures undefinedness:

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

LPF is our preferred approach to handling non-denoting terms and Section 2.1 presents a more detailed discussion.

McCarthy's conditional operators (discussed in Ref.[23]) also give rise to a non-classical logic. This approach imposes a left-to-right evaluation, since conditional expressions are strict in their first argument. The conditional operator approach deals adequately with expressions where there is a form of "guard" as in Property 2:

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

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:

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

[3)] An interesting link between variant notions of equality and LPF proofs is examined in Ref.[14].

Property 1 also poses a problem in this approach since, while $true \vee \bot_{\mathbb{B}}$ and $\bot_{\mathbb{B}} \vee true$ are truths in LPF, the latter is not in McCarthy's approach (cf. Ref. [17]).

It is also interesting to note that it would appear that undefined values corrupt quantified expressions such that:

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

does not have the same truth value as:

$$zero(1) = 0 \vee zero(0) = 0 \vee zero(-1) = 0$$

since the undefinedness of $zero(-1)$ makes the whole existentially quantified expression undefined. All of these issues are resolved in LPF.

An extreme approach to the problem of non-denoting terms is to *prohibit* the use of partial functions and operators. Taken this generally, the rule would be unworkable but various notions of (roughly) left-to-right guarding can cope with Property 2. Unfortunately, this is much more difficult to extend to:

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

Finally, the simple implication of Property 2 can, of course, be rewritten over a restricted set as: $\forall i \in \mathbb{N} \cdot zero(i) = 0$ but this approach does not provide a pleasant treatment for the disjunction in Property 1 and restricting types becomes messy for functions of more than one argument where the required "dependant type" needs a predicate of all arguments — see Ref.[12].

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)*[4,13,16].

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 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[5]. From this point forward, $\bot_{\mathbb{B}}$ and $\bot_{\mathbb{Z}}$ are to be understood as ways of representing "gaps".

The truth tables in Fig.2 (presented in Ref.[19]) 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"[13]: 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)

4) A brief history of LPF: Three-valued truth tables for the propositional operators are given in Ref.[19]; Peter Aczel supervised Koletsos' research[20] in which he gives a proof theory for such a logic; Cliff Jones suggested that Jen Cheng[10] apply this to programming tasks[3]; the typed version of LPF is covered in Ref.[15].

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**

Figure 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 an infinite disjunction (in the worst case) and universal quantification as an infinite conjunction as is standard in First Order Predicate Calculus (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. Similar comments apply, *mutatis mutandis*, for 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[15], one reasons about (the least fixed point interpretation of) recursive functions such as the *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 in 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 (in, for example, the style of Ref.[4]) is presented in Fig.3[5)]. Moreover, Property 4 is also *true* in LPF:

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

	from $i \in \mathbb{Z}$	
1	$i \geqslant 0 \vee i < 0$	h1, $\mathbb{Z}$
2	**from** $i \geqslant 0$	
2.1	$zero(i) = 0$	$\Rightarrow$-E-L(Lem(h1), 2.h1)
	infer $zero(i) = 0 \vee zero(-i) = 0$	$\vee$-I-R(2.1)
3	**from** $i < 0$	
3.1	$-i \geqslant 0$	h1, 3.h1, $\mathbb{Z}$
3.2	$-i \in \mathbb{Z}$	h1, 3.1, $\mathbb{Z}$
3.3	$zero(-i) = 0$	$\Rightarrow$-E-L(Lem(3.2), 3.1)
	infer $zero(i) = 0 \vee zero(-i) = 0$	$\vee$-I-L(3.3)
	infer $zero(i) = 0 \vee zero(-i) = 0$	$\vee$-E(1, 2, 3)

Figure 3. A proof of Property 1

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 $zero(-1) = 0 \vee \neg (zero(-1) = 0)$ is not considered to be a tautology.

For expressive completeness, a defined (δ) operator is available in LPF: $\delta(e)$ returns *true* if e is defined (it is *true* or it is *false*). This gives LPF exactly 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:

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

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.

[5)] Where the Lem used in this proof refers to the implication lemma $i \geqslant 0 \;\Rightarrow\; zero(i) = 0$, where i is an integer, and whose proof also follows with little difficulty.

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[28]). Our SOS specification provides an intuitive introduction to the semantics of LPF and how it 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 since 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 proving 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 for which we are interested in providing the semantics. 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 numerous 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 (useful for recursive function definitions), propositional logic expressions (disjunction, negation, and the defined operator — δ)[6], quantified expressions (universal and existential)[7] and a function application expression.

A function in our language always takes a (single) integer argument and returns an integer result[8]; a function definition thus contains a parameter name and a resulting expression (that might include recursive calls to the function). We intend that the only variable that can be used within a function's result expression is its parameter. A function invocation expression requires the name of the function to be called and an argument to pass to the function.

In the semantic rules for a function invocation, we need to be able to access the function definition that is being called. To achieve this, we use a map Γ from function names to their corresponding function definitions:

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

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

---

[6] 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. Similarly, we only present the relational (equality) operator and not relational operators such as $>$, $\geqslant$, etc.

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

[8] This restricted form of function definition allows us to highlight the issues in reasoning about propositions over terms that can fail to denote. Lifting the restrictions is of course trivial.

A record is used to represent a single function definition (VDM notation[16] is used throughout):

$$\begin{array}{lll} Func & :: & param \ : \ Id \\ & & result \ : \ Expr \end{array}$$

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

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

and (notice that BOOL is a primitive token naming the type $\mathbb{B}$):

$$Type = \text{BOOL} \mid \text{INT}$$

We only intend to provide semantics for those expressions which are well-formed in the sense that they satisfy the following criteria[9)]:

- 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 INT
- Propositional logic expressions (disjunction, negation, and δ) are well-formed if all of their operand(s) 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* sub-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
- A function invocation expression is well-formed if the argument is well-formed and of the type INT, and the function to be called exists in the given γ map

In addition we only tackle well-formed function definitions and we consider a function definition to be well-formed if its result expression is well-formed (based on the above criteria) and of the same type as the return type of the function (INT). We intend that only the parameter of a function should be used as a variable within a function; thus a function's result expression should be checked to see if it is well-formed with only the functions parameter name included as a variable within a given *Types* map.

9) These context conditions are given formally in Ref. [22] where use is also made of abstract syntax.

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 using 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; whereas, 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 need to use a map –that we refer to as a memory store– which maps the variable identifiers to the values they store:

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

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

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. A memory store (σ) is a global object in our language in the sense that the associations between identifiers and their values cannot be changed in expression evaluation.

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; the rules to define FOPC are straightforward modifications.

The semantic relation that we use to model expression evaluation is[10)]:

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

where required, we use $\xrightarrow{E}$ for the reflexive, transitive closure of $\xrightarrow{e}$. The presence of Σ only on the left of the $\xrightarrow{e}$ relation shows clearly that there is no notion of side-effects that would change the store but the lack of uniformity means that a conventional transitive closure cannot be used. We can however define our own:

$$(e_1, \sigma) \xrightarrow{E} e_2 \;\Leftrightarrow\; e_1 = e_2 \vee \exists e_i \in Expr \cdot (e_1, \sigma) \xrightarrow{e} e_i \wedge (e_i, \sigma) \xrightarrow{E} e_2$$

The semantic rules use 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 selected arithmetic operators and from the rules for the relational equality operator below among others. It is important to have such interleaving for logical operators such as disjunction since they have to cope with the

[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.

"gaps" that can occur. This point is explained below when the semantic rules for disjunction are introduced.

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 computed (eliminating the operator from the expression). 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 $[\![op]\!](a, b)$ is the expected result from evaluating op on its two operands.

Non-denoting terms arise from this arithmetic semantics with expressions that reduce to something of the form $i \div 0$. Notice that the *Arith-E*2 semantic rule is one of the places that give rise to "gaps" in the value space.

To illustrate how expressions are evaluated in our language, consider the example presented in Fig.4, where the expression we are evaluating is $x - x$, with a (global) memory store (σ) containing the variable x mapping to the value 3.

$\sigma = \{x \mapsto 3\}$
$(x - x, \sigma) \xrightarrow{e} 3 - x$ *Arith-L, Id-E*
$(3 - x, \sigma) \xrightarrow{e} 3 - 3$ *Arith-R, Id-E*
$(3 - 3, \sigma) \xrightarrow{e} 0$ *Arith-E*1
thus:
$(x - x, \sigma) \xrightarrow{E} 0$ $\xrightarrow{E}$

Figure 4. An example of how an expression can be evaluated in our language

The following set of semantic rules define weak (strict) equality[14] which returns a result only if both operands denote values; otherwise, the relational (equality) expression will fail to denote a value. The truth table for weak equality is discussed in Section 1.

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

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

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

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 purpose of providing a small-step semantics is to allow for the interleaving of steps in different expression branches 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 could start to evaluate this 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 Fig.2.

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

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

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

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

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

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; we have no control over which rule is used. Ideally when evaluating a disjunction expression we would like

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

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 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 iterating 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 invocation 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 must 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 have so far coped with "gaps" in propositional calculus, we would now like to extend this to cover quantified expressions. Here we define the quantification rules using big-step semantics; this could be avoided but the required configurations become rather messy. We start with universal quantification. The following semantic

rule states that 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.

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

For the following semantic rule, it is necessary that for every integer i which when applied to the expression e causes e to evaluate to *true*.

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

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 in the *Forall-F* semantic rule as shorthand for an infinite disjunction (using the LPF disjunction logical operator already introduced), and the use of the universal quantifier above the line in the *Forall-T* semantic rule as shorthand for an infinite conjunction (both over the set of integers). This is perfect for our intuition but means that the semantics is really only "semi-formal".

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

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

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

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 (for which the concrete syntax $e\ ?\ t\ :\ s$ is chosen).

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

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

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

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" through the use of our function invocation expression. First we must update our semantic relation to model the process of expression evaluation to include Γ. Fortunately, 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.

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

Any argument used in a function invocation must denote, otherwise we do not evaluate the function.

There are two approaches to evaluating the expression in a function: it is easy to understand a big step semantics so this is shown first; but it does not cope with undefined in the way we want so some technical machinery is erected to provide a small step semantics.

Once (if) the argument expression has been reduced to a constant value we can attempt to evaluate the function's result expression[12)].

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

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)].

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

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.

[12)] Where $\gamma(id)$ retrieves a function definition (represented as a *Func* record) from the given γ map corresponding to the function name id, and the following *.result* and *.param* are used to retrieve the selected data from the function definition under question.

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

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 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.

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

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

The purpose of using the *FuncInter* expression construct is to allow for the current state of the result to be stored (alongside the parameter data) so that the evaluation of a functions result can resume from where it left off previously if any interleaving of the steps in expression branches occurs. This allows us to provide a small-step semantics, so for instance given an expression such as $zero(1) = 0 \vee zero(\text{-}1) = 0$, if we start evaluating the non-denoting operand of the disjunction operator $(zero(\text{-}1) = 0)$ we can make the one step in evaluating this operand using our small-step function semantic rules presented above. After that one evaluation step has been performed it is then possible for the other (denoting in this case) operand to be evaluated and thus a result (*true* in this case) could eventually be returned. This may not be possible with the original *FuncCall-E* semantic rule which made a big-step in evaluating the result of a function. The use of *FuncInter* in our semantics for function invocation is how our approach differs from the approach presented in Ref.[18].

The overall value v of an expression e is found by:

$$(e, \{\,\}, \gamma_0) \xrightarrow{e} v$$

where γ_0 contains any required function definitions and the empty starting store is used because all expressions are assumed to be closed by universal quantification; the context conditions ensure that only bound identifiers are referenced during evaluation.

To illustrate how an expression containing a function invocation is evaluated in our language consider the example presented in Fig.5, where the expression we are evaluating is $x - zero(y - 1)$, given a function definition (stored in γ) of $zero(i) \triangleq i = 0\ ?\ 0 :\ zero(i - 1)$, where i is an integer.

[14] The parameter is included in the memory store during the evaluation of the functions result, but notice that the updated memory store is not returned by the semantic rule. After the one evaluation step has been made through the *FuncInter-A* semantic rule the update made to the given memory store is effectively undone. 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 (in the form of a *FuncInter* expression construct). This is to achieve the necessary variable scoping since interleaving of steps in different expression branches is allowed and is necessary for LPF.

$\sigma = \{x \mapsto 2, y \mapsto 1\}$
$\gamma = \{zero \mapsto Func(i, i = 0\ ?\ 0 :\ zero(i-1))\}$
$(x - zero(y-1), \sigma, \gamma) \xrightarrow{e} x - zero(1-1)$ *Arith-R*, *FuncCall-A*, *Arith-L*, *Id-E*
$(x - zero(1-1), \sigma, \gamma) \xrightarrow{e} x - zero(0)$ *Arith-R*, *FuncCall-A*, *Arith-E*1
$(x - zero(0), \sigma, \gamma) \xrightarrow{e} 2 - zero(0)$ *Arith-L*, *Id-E*
$(2 - zero(0), \sigma, \gamma) \xrightarrow{e} 2 - FuncInter(i = 0\ ?\ 0 :\ zero(i-1), i, 0)$ *Arith-R*, *FuncCall-E*
$(2 - FuncInter(i = 0\ ?\ 0 :\ zero(i-1), i, 0), \sigma, \gamma) \xrightarrow{e}$
$2 - FuncInter(0 = 0\ ?\ 0 :\ zero(i-1), i, 0)$ *Arith-R*, *FuncInter-A*, *Cond-A*, *Equality-L*, *Id-E*
$(2 - FuncInter(0 = 0\ ?\ 0 :\ zero(i-1), i, 0), \sigma, \gamma) \xrightarrow{e}$
$2 - FuncInter(true\ ?\ 0 :\ zero(i-1), i, 0)$ *Arith-R*, *FuncInter-A*, *Cond-A*, *Equality-E*
$(2 - FuncInter(true\ ?\ 0 :\ zero(i-1), i, 0), \sigma, \gamma) \xrightarrow{e} 2 - FuncInter(0, i, 0)$ *Arith-R*, *FuncInter-A*, *Cond-E*1
$(2 - FuncInter(0, i, 0), \sigma, \gamma) \xrightarrow{e} 2 - 0$ *Arith-R*, *FuncInter-E*
$(2 - 0, \sigma, \gamma) \xrightarrow{e} 2$ *Arith-E*1

Figure 5. An example of how an expression containing a function invocation can be evaluated in our language

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. It is also worth noting that the semantic rules for the quantifiers contain infinitely many premises since we quantify over the set of integers. In the case of proof systems this is referred to as "semi-formal". 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 modelled by choosing *relations* as the space of denotations. This is in contrast to the use of partial functions as is classical in denotational semantics[30]. 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 Ref.[15] and Ref.[2] (or, originally[24]).

In order to facilitate comparison with Section 3, its order of presentation is followed particularly in postponing the treatment of functions/Γ; in a pure denotational description this might have been presented first. So:

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

which is defined here in parts:

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

Access to named values presents no difficulties (Section 3.1 rules out reference to unknown names):

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

Straightforwardly:

$$\mathcal{E}arith = \{((a\ -\ b, \sigma, \gamma), [\![-]\!](a', b')) \mid ((a, \sigma, \gamma), a') \in \mathcal{E} \wedge ((b, \sigma, \gamma), b') \in \mathcal{E}\} \cup \{((a \div b, \sigma, \gamma), [\![\div]\!](a', b')) \mid ((a, \sigma, \gamma), a') \in \mathcal{E} \wedge ((b, \sigma, \gamma), b') \in \mathcal{E} \wedge b' \neq 0\}$$

Remembering that weak equality is strict gives:

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

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

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

Again, straightforwardly:

$$\mathcal{E}not = \{((\neg a, \sigma, \gamma), \mathbf{false}) \mid ((a, \sigma, \gamma), \mathbf{true}) \in \mathcal{E}\} \cup \{((\neg a, \sigma, \gamma), \mathbf{true}) \mid ((a, \sigma, \gamma), \mathbf{false}) \in \mathcal{E}\}$$

and:

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

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

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

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

The semantics for the conditional expression are also straightforward:

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

It is useful to record that the definition of $\mathcal{E}$ is deterministic (or "functional"):

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 $\mathcal{E}arith$, $\mathcal{E}or$, $\mathcal{E}not$, $\mathcal{E}forall$, $\mathcal{E}exists$ and $\mathcal{E}cond$ are defined with set unions, the domains of the relations only overlap in obvious cases such as $true \vee true$ where the results are the same.

The definition of clauses of $\mathcal{E}$ above are straightforward in the sense that they are over the structure of *Expr*; as with the operational semantics in Section 3, the context conditions ensure that only bound variables are referenced (thus $\sigma_0 = \{\,\}$). The definition of $\mathcal{E}funccall$ is more interesting because it is recursive; moreover, the function environment Γ should contain semantic –rather than syntactic– objects. Thus:

$$\Gamma = Id \xrightarrow{m} \mathcal{P}(Value \times Value)$$

which is constructed (assume *funs* contains all of the function definitions) by:

$$\gamma_0 = \{id \mapsto \{(i,v) \mid ((funs(id).result, \{funs(id).param \mapsto i\}, \gamma_0), v) \in \mathcal{E}) \mid i \in \mathbb{Z}\} \mid id \in \mathbf{dom}\, funs\}$$

As in the standard denotational style, the recursion over γ_0 is a lifting of the recursion on $\mathcal{E}$ but to be sure that this makes sense it is necessary to give the ordering over function denotations. This is straightforward since the definition is monotonic with respect to set containment.

Having developed denotations for the function definitions, function application expressions are now simpler than in the SOS:

$$\mathcal{E}funccall = \{((f(arg),\sigma,\gamma),res) \mid ((arg,\sigma,\gamma),arg') \in \mathcal{E} \wedge (arg',res) \in \gamma(f)\}$$

This has achieved what was expected (with the *zero* function as in Section 1) which is that $((zero(1),\sigma,\gamma),0) \in \mathcal{E}$ but $(zero(-1),\sigma,\gamma) \notin \mathbf{dom}\,\mathcal{E}$. It is, however, interesting to note that the fixed point construction of the function denotations can be thought of as a bottom up construction of what is done in the SOS by abandoning infinite expansion of the function calls.

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 Fig.3 is far preferable; the interest is that having based the semantics on simple set theory, even direct proofs are straightforward.

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.

$$pos : \mathbb{Z} \to \mathbb{B}$$

$$pos(i) \;\triangleq\; \mathbf{if}\; i = 0\; \mathbf{then}\; \mathbf{true}\; \mathbf{else}\; pos(i-1)$$

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

This can be proved by a simple inductive argument:

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

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

$$\begin{array}{l}\forall n \in \mathbb{N}_1 \cdot \\ \quad ((pos(i), \{i \mapsto n-1\}, \gamma), \mathbf{true}) \in \mathcal{E} \\ \quad \Rightarrow\ ((pos(i), \{i \mapsto n\}, \gamma), \mathbf{true}) \in \mathcal{E}\end{array}$$

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

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

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:

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

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.

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. 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 we come up with is correct.

One way of implementing mechanised tool support for proofs in LPF would be to implement a theorem prover for LPF from scratch. However, we share the view that is put forward in Chapter 4 "Designing a Theorem Prover" of Ref. [1] that it may be a better idea to try to captialise on an existing tool and build our extension on top of that. The current options that we see include extending an existing proof assistant (notably *Isabelle*[26]) or adapting a term-rewriting tool (notably *Maude*[9]) to include support for LPF. Our preferred option is to utilise the generic proof assistant *Isabelle* and to embed LPF within this tool. For this task we envisage that the SOS specification we presented within Section 3 will be of great use.

General Acknowledgements

The content of this paper has benefited greatly from discussions with Jason Steggles and Joey Coleman. The authors would also like to thank the referee for the comments we received which we hope have helped clarify our explanation. The authors of this paper also gratefully acknowledge the funding for their research from an EPSRC PhD Studentship and EPSRC grants: TrAmS and AI4FM.

Acknowledgements to Manfred Broy from Cliff Jones

It is a great pleasure to be able to contribute to this *Festschrift* edition and to add here a few words of thanks to Manfred Broy. My evolving research agenda has overlapped with Manfred's for several decades. Our parallel interests include data types, concurrency and understanding what computer systems are actually required. I'll say a few words about each of these topics.

Back when Manfred was in Passau, I recall many happy arguments about the relative merits of what I describe as property-oriented and model-oriented descriptions of data types. Manfred was, of course, a strong proponent of "algebraic descriptions" whereas I had consistently used models to describe complex systems. I can remember at that time challenging him with issues of both partiality (to which I return below) and non-determinacy.

Nor did we entirely agree on the best way to tackle the intriguing problems around reasoning about concurrent programs. We certainly shared the view that this was a

major issue and I was delighted that Ketil Stølen went from my Manchester group to work with Manfred in Munich; their book[6] is a wonderful example of two good scientists benefiting from each other's contribution. Another fine scientist who moved from my group to Manfred's is Tobias Nipkow who has ploughed his own distinct and important research furrow. As a mutual friend, Tobias provides another bridge between myself and Munich.

Another topic about which I fully share Manfred's concerns is that formalists must concede that knowing how to develop perfect programs *from* specifications is, sadly, not enough. However difficult this is; however much more comfortable it is to live entirely in a world of formal symbols; we have to accept that, if the wrong system is built, the result will at best be unhappy customers and at worst deaths of people around the system. We must research what can be done with formal approaches to help reduce these dangers. In this vein, I wrote the postscript to my VDM book for the first (1986) edition against the backdrop of a US President who thought that vague desires to "shoot down any commie missiles" would serve as a specification for the "Star Wars" (SDI) programme.

It will probably not have escaped the reader's notice that, although I have suggested Manfred and I have been interested in the same problems, we do not always agree on the best solution to those problems. The degree of disagreement varies from topic to topic, but what is more important is the way in which this brings out another crucial facet of my experience with Manfred. One of the delightful things about Manfred is that I have always enjoyed the disagreements — our differing views have always been clarified by good discussions — and there has never been a hint that our differences could ever damage the friendship between us.

Turning now to our links on the specific issues in this paper, I confess to initially feeling apologetic when I switched from my intended topic to that of reasoning about partial functions. It quickly became clear that I had no need: looking through my files of Manfred's work, I found[29] which tackles the problem of undefined terms in the context of Dijkstra's "calculational logic". When I mentioned this to Martin Wirsing, he pointed me to further relevant papers[7,8]. I think it is true that if one tried to translate Manfred's approach into a purely logic-based framework that one would find it hard to avoid viewing partial functions as relations. This interpretation will no doubt be a topic of discussion for our next meeting which is likely to be under the aegis of IFIP's WG2.3 group whose meetings have provided a regular possibility to meet. With his customary hospitality, Manfred has also hosted meetings more often than most members.

On a more personal note, it has been a pleasure to know Manfred's lovely growing family; with his wife Karin, we share a love of fine restaurants and good wine.

References

[1] Abramsky S, Gabbay DM, Maibaum SE, eds. Handbook of Logic in Computer Science (vol. 2): Background: Computational Structures. Oxford University Press, Inc., New York, NY, USA, 1992.

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

[3] Barringer H, Cheng JH, Jones CB. A logic covering undefinedness in program proofs. Acta Informatica, 1984, 21: 251–269.

[4] Bicarregui J, Fitzgerald J, Lindsay P, Moore R, Ritchie B. Proof in VDM: A Practitioner's Guide. FACIT. Springer-Verlag, 1994. ISBN 3-540-19813-X.
[5] Blamey SR. Partial Valued Logic [Ph.D. Thesis]. Oxford University, 1980.
[6] Broy M, Stølen K. Specification and Development of Interactive Systems. Springer-Verlag, 2001.
[7] Broy M, Wirsing M. Partial functions and abstract data types. Bulletin of the European Association of Theoretical Computer Science, 1980, 11: 34–41.
[8] Broy M, Wirsing M. Algebraic definition of a functional programming language and its semantic models. RAIRO, 1983, 17(2): 137–162.
[9] Clavel M, Durán F, Eker S, Lincoln P, Martí-Oliet N, Meseguer J, Talcott C. All about Maude – a high-performance logical framework, how to specify, program and verify systems in rewriting logic. LNCS 4350, Springer, 2007.
[10] Cheng JH. A Logic for Partial Functions [Ph.D. Thesis]. University of Manchester, 1986.
[11] Cheng JH, Jones CB. On the usability of logics which handle partial functions. Technical Report UMCS-90-3-1, Manchester University, February 1990. Preprint of [12]
[12] Cheng JH, Jones CB. On the usability of logics which handle partial functions. In: Morgan C, Woodcock JCP, eds. 3rd Refinement Workshop. Springer-Verlag, 1991. 51–69.
[13] Fitzgerald JS. The typed logic of partial functions and the vienna development method. In: Bjørner D, Henson MC, eds. Logics of Specification Languages, EATCS Texts in Theoretical Computer Science. Springer, 2007. 427–461.
[14] Fitzgerald JS, Jones CB. The Connection between Two Ways of Reasoning about Partial Functions. IPL, 2008, 107(3-4): 128–132.
[15] Jones CB, Middelburg CA. A typed logic of partial functions reconstructed classically. Acta Informatica, 1994, 31(5): 399–430.
[16] Jones CB. Systematic Software Development using VDM. Prentice Hall International. Second, 1990.
[17] Jones CB. Reasoning About Partial Functions in the Formal Development of Programs. In: Proc. of AVoCS'05. 145, 3–25. Elsevier, Electronic Notes in Theoretical Computer Science, 2006.
[18] Kahn G. Natural semantics. STACS'87: Proc. Fourth Annual Symposium on Theoretical Aspects of Computer Science. Springer-Verlag, 1987. 22–39.
[19] Kleene SC. Introduction to Metamathematics. Van Nostrad, 1952.
[20] Koletsos G. Sequent Calculus and Partial Logic [MS Thesis]. Manchester University, 1976.
[21] Krauss A. Partial recursive functions in higher-order logic. International Joint Conference on Automated Reasoning (IJCAR 2006). LNCS, Springer-Verlag, 2006. 589–603.
[22] Lovert MJ. A Semantic Model for a Logic of Partial Functions. In: Pierce K, Plat N, Wolff S, eds. School of Computing Science Technical Report. CS-TR-1224, 33–45. Newcastle University, 2010.
[23] McCarthy J. A basis for a mathematical theory for computation. In: Braffort P, Hirschberg D, eds. Computer Programming and Formal Systems. North-Holland Publishing Company, 1967. 33–70.
[24] Monahan BQ. A type model for VDM. LNCS 252, Spring-Verlag, 1987. 210–236.
[25] Müller O, Slind K. Treating Partiality in a Logic of Total Functions. The Computer Journal, 1997, 40(10): 640–652.
[26] Nipkow T, Paulson LC, Wenzel M. Isabelle/HOL — A proof assistant for higher-order logic. LNCS 2283, Springer, 2002.
[27] Owe O. 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.
[28] Plotkin GD. A Structural Approach to Operational Semantics. Technical report, Aarhus University, 1981.
[29] Schieder B, Broy M. Adapting calculational logic to the undefined. The Computer Journal, 1999, 42(2).
[30] Stoy JE. Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. MIT Press, 1977.