Using an Expression Interpreter to Reason With Partial Terms

Lev Naiman
Department of Computer Science
University of Toronto
Toronto, Canada
Email: naiman@cs.toronto.edu

***Abstract* — Refinements of programming specifications often include partial terms and need to be handled using formal rules. The idea of an expression interpreter over character strings is presented as a candidate solution. The interpreter allows for reasoning with partial terms without requiring a meta-logic or a logic with more than two values. We show how the interpreter can be used to create generic and compact laws, which also allow simple reasoning about expressions syntactically. We argue that it is simple to integrate the interpreter within existing theorem provers.**

Keywords* — *logic; partial-terms;expression interpreter; theorem prover; two-valued logic

I. Introduction

In programming specifications and their refinements we commonly encounter partial terms. Partial terms are defined as expressions that fail to denote a value. A term t in a theory T is partial if there are no laws in T that apply to t. An example is where a function or an operator is applied to an argument outside of its domain, such as $1/0$. We also say that a formula e is unclassified in theory T if it is neither classified as a theorem or an anti-theorem. Such expressions are present in proofs of programs due to the partial functions and operators that are often used in specifications. Borrowing an example from [1], we might implement the difference function as follows (where the domain of *diff* is integers, and the assumed theory is arithmetic and first-order two-valued logic).

$$\mathit{diff}\, i\, j = \mathbf{if}\ i = j\ \mathbf{then}\ 0\ \mathbf{else}\ (\mathit{diff}\, i\, (j+1)) + 1\ \mathbf{fi} \tag{1}$$

We would like to prove

$$\forall i, j : int \cdot i \geq j \Rightarrow (\mathit{diff}\, i\, j) = i - j \tag{2}$$

but when trying to simplify this expression instantiated with 1 and 2 respectively for i and j we get

$$\begin{aligned} 1 \geq 2 \quad &\Rightarrow \quad (\mathit{diff}\, 1\, 2) = 1 - 2 \\ = \ \mathbf{F} \quad &\Rightarrow \quad (\mathit{diff}\, 1\, 2) = -1 \end{aligned} \tag{3}$$

and we cannot apply any laws at this point to simplify it further. A law would allow simplifying the expression to true, but it requires that both operands be boolean. The expression *diff* 1 2 is a partial term because no laws apply to it. For this reason we cannot use any law to conclude that $(\mathit{diff}\, 1\, 2) = -1$ is a boolean, even though it has the form $X = Y$. Tools that reason with such expressions must be based on formal rules in order to have confidence in their proofs. We propose a character-string interpreter to solve this problem.

The rest of the paper is organized as follows: in section II we examine the existing approaches in the literature to cope with partial terms. In section III we describe the background theories we use to define the interpreter in IV. Section V shows how the interpreter can be used to cope with partial terms. Section VI describes other benefits of the interpreter when constructing theories. Section VII describes how we can extend the definition of the interpreter to be more expressive.

II. Current Approaches to Partial Terms

One approach to resolve partial terms is to make all terms denote. Formally this means that for each partial term such as $x/0$, a law must exist saying which set of values that expression is a member of. This set of values is assumed to already be defined in the logic, as opposed to newly created values. In this case there could be a law defined saying that $\forall x : int \cdot x/0 : int$. This is the approach used in the programming theory of [2]. Such laws do not explicitly say what value a partial term is equal to, and this can cause certain peculiar and possibly unwanted results such as $0/0 = 0$ being a theorem.

$$\begin{aligned} & 0 \\ &= 0 \times (1/0) \\ &= 1 \times (0/0) \\ &= 0/0 \end{aligned} \tag{4}$$

This approach can be slightly modified and the value of partial terms can be fixed. However, this might cause some unwanted properties. In the case of division by zero a choice of 42 as used in [3] cannot be allowed due to inconsistency.

In [4] the authors point out that underspecification alone may cause problems. If we allow domains of single elements then these problems can go as far as inconsistency. The semantic model of our interpreter uses underspecification, but not exclusively. In some cases, similarly to LPF, the interpreter would leave some expressions unclassified. One way of finding a model for partial functions in set theory is the standard approach of mapping any unmapped element from the domain to a special value, usually called $\perp$ [5]. The denotational semantics for a generic law for equality are extended with

TABLE I
THREE-VALUED BOOLEAN OPERATORS

	T	**F**	$\bot$
$\neg$	**F**	**T**	$\bot$

	TT	**TF**	**FT**	**FF**	**T**$\bot$	$\bot$**T**	$\bot$**F**	**F**$\bot$	$\bot\bot$
$\vee$	**T**	**T**	**T**	**F**	**T**	**T**	$\bot$	$\bot$	$\bot$
$\wedge$	**T**	**F**	**F**	**F**	$\bot$	$\bot$	**F**	**F**	$\bot$

this value, and in this particular model $7/0 = 5/0$ would be a theorem (assuming strict equality). However, a user of a logic that includes the interpreter would not need to perform any calculations that concern this extra value.

The logic of partial terms (LPT) [6], [7] is an example of a logic that does not include the undefined constant. It does however include a definedness operator $\downarrow$. In this theory the specialization law $(\forall x \cdot A(x)) \Rightarrow A(v)$ requires that v be defined. The basic logic of partial terms (BPT) [8] is a modification of LPT, and relaxes the previous requirement for some laws. It allows for reasoning with non-terminating functional programs. Some logics such as [9] inlcude multiple notions of equality to be used in calculations. This may complicate the laws of quantifiers.

Another approach to deal with partial terms is a non-classical logic such as LPF [3] with more than two values. In these logics the truth table of boolean operators is usually extended as in table I (where $\bot$ represents an "undefined" value, and the column heads are both of the arguments to the operator). In this logic the expression $0/0 = 1$ would not be classified to one of the boolean values, but would rather be classified as $\bot$. Undefinedness is either resolved by the boolean operators or is carried up the tree of the expression. Some three valued logics have a distinct undefined value for each value domain, such as integers and booleans.

Three and more valued logics have varied useful applications. However, a drawback of using a logic with multiple truth values is that certain useful boolean laws no longer hold. This is particularly true of the law of the excluded middle, $\forall x : bool \cdot x \vee \neg x$, which in a three value logic can be modified to $\forall x : bool \cdot x \vee \neg x \vee undefined(x)$. In the Logic of Computable Functions (LCF) [10] there is a $\bot_t$ value for each type t, requiring the modification of several laws. Another issue of multiple valued logics is that not knowing the value of an expression seems to be pushed one level up; attempting to formalize these extra values will result in a semantic gap. There are always expressions that must remain unclassified for a theory to remain consistent.

A further method of dealing with partial terms is conditional, or short-circuit operators [11]. This approach is similar to those logics with three values, since it gives special treatment to partial terms. Boolean operators have an analogous syntax $a\,\mathbf{cor}\,b$, $a\,\mathbf{cand}\,b$, $a\,\mathbf{cimp}\,b$, etc. In these expressions if the first value is undefined, then the whole expression is undefined. These conditional operators are not commutative.

III. BACKGROUND THEORIES

We introduce two theories from [12] that we will use to define the interpreter.

A. Bunch Theory

A bunch is a collection of objects. It is different from a set, which is a collection of objects in a package. A bunch is instead just those objects, and a bunch of a single element is just the element itself. Every expression is a bunch, but not all bunches are elementary. Here are two bunch operators.

$$A\,,B \qquad \text{A union B} \tag{5}$$

$$A : B \qquad \text{A in B, or A included in B} \tag{6}$$

Operators such as a comma, colon, and equality apply to whole bunches, but some operators apply to their elements instead. In other words, they distribute over bunch union. For example

$$\begin{aligned} & 1 + (4, 7) \\ &= 1 + 4\ ,\ 1 + 7 \\ &= 5\,, 8 \end{aligned} \tag{7}$$

Bunch distribution is similar to a cross-product in set theory.

B. String Theory

A string is an indexed collection of objects. It is different from a list or ordered pair, which are indexed collections of objects in a package. A string of a single item is just that item. The simplest string is the empty string, called nil. Strings are joined together, or concatenated with the semicolon operator to form larger strings. This operator is associative but not commutative. The string $0;1$ has zero as the first item and one as the second. For a natural number n and a string S, $n * S$ means n copies of S. Let nat be the bunch of natural numbers. The copies operator is defined as follows.

$$0 * S = nil \tag{8}$$

$$\forall\, n : nat \cdot (n + 1) * S = n * S; S \tag{9}$$

Strings can be indexed, and their length can be obtained with the length operator ($\leftrightarrow$).

$$S_n \qquad \text{S at index n} \tag{10}$$

$$\leftrightarrow S \qquad \text{length of S} \tag{11}$$

A semicolon distributes over bunch union, as so does an asterisk in its left operand. Some examples of the operators defined are

$$\begin{aligned} \leftrightarrow (7;1;0) &= 3 \\ (7;1;0)_0 &= 7 \\ 1;(5,17);0 &= (1;5;0),(1;17;0) \\ 3 * (0;1) &= 0;1;0;1;0;1 \\ (0,1) * (0;1) &= 0 * (0;1), 1 * (0;1) = nil, 0;1 \end{aligned} \tag{12}$$

The prefix copies operator $*S$ is defined to mean $nat * S$, or informally the bunch of any number of copies of S. Finally,

we introduce characters, which we write with double quote marks such as “a”, “b”, etc. To include the open and close double-quote characters we escape them with a backslash: “\“”. Strings that contain exclusively character strings are sometimes abbreviated with a single pair of quotes: “abc” is short for “a”; “b”; “c”. If the bunch of all characters is called $char$, then the bunch of all two-character strings is $char; char$.

Bunch and string theory are used because they allow for compact language definitions. For example, denoting the collection of naturals greater than zero in set theory can be done by writing $\{n : nat | n > 0\}$. In bunch theory it can be written as $nat + 1$. We can of course define an addition operator that distributes over the contents of a set, but the benefit of bunch theory (and analogously string theory) is that no such duplication is necessary.

IV. Defining the Interpreter

We would like a simple method to reason with partial terms that introduces as few new operators as possible, and which preserves the properties of existing operators. We would also like to avoid a separate meta-language, and to do all reasoning within a single logic. In the literature authors often use one set of symbols for the meta-logic operators and another for the object logic. We use character strings instead both for clarity, and in the case where we wish to use the logic to study itself. We take the idea of the character-string predicate of Hehner [13], and we expand it to be a general interpreter for any expression in our language. To maintain consistency we exclude the interpreter itself from the interpreted language. The interpreter, which we call $\mathcal{I}$ is an operator which applies to character strings and produces an expression. The interpreter can be thought of as unquoting a string. We first define our language as a bunch of character strings.

Let *char* be the bunch of all character symbols, let *alpha* be the bunch of character symbols in the English alphabet, and let *digit* be the bunch of digit character. We define our language *lang* to be the following bunch of strings.

$$
\begin{aligned}
α *alpha; *“\prime” = var \\
&digit; *digit = num \\
&binops = “\wedge”, “\vee”, “=”, “\Rightarrow”, “\Leftarrow”, “;”, “-”, “\ ” \\
&var, num, “\mathbf{T}”, “\mathbf{F}” : lang \\
&“\langle”; var; “:”; lang; “\rightarrow”; lang; “\rangle” : lang \\
&“(”; lang; “)” : lang \\
⟨ binops; lang : lang \\
&“\neg”, “-”, “\forall”, “\exists”; lang : lang \\
&“\backslash“”; *char; “\backslash”” : lang
\end{aligned}
\tag{13}
$$

Here we have defined a language that includes boolean algebra, numbers, logical quantifiers, functions, and strings. The language is defined similarly to how a grammar for a language would be given. Function syntax is $\langle v : D \to B \rangle$, where the angle brackets denote the scope of the function, and v is the introduced variable of type D. We treat quantifiers as operators that apply to functions. The quantifiers $\exists$ and $\forall$ give boolean results. When we use more standard notation such as $\forall v : domain \cdot body$ we mean it as an abbreviation for $\forall \langle v : D \to B \rangle$.

The interpreter is intuitively similar to a program interpreter: it turns passive data into active code. Our interpreter turns a text that represents an expression into the expression itself. The interpreter is defined very closely to how $lang$ was defined. The laws are as follows.

$$
\begin{aligned}
&\mathcal{I}“\mathbf{T}” = \mathbf{T} \\
&\mathcal{I}“\mathbf{F}” = \mathbf{F} \\
&\forall\, s : (digit; *digit) \cdot \forall\, d : digit \cdot \mathcal{I}(s; d) = (\mathcal{I}s) \times 10 + (\mathcal{I}d) \\
&\forall\, dom, body : lang \cdot \\
&\mathcal{I}(“\langle a :”; dom; “\rightarrow”; body; “\rangle”) = \langle a : \mathcal{I}dom \to \mathcal{I}body \rangle \\
&\forall\, s : lang \cdot \mathcal{I}“(”; s; “)” = \mathcal{I}s \quad \wedge \\
&\qquad \mathcal{I}(“\neg”; “(”; s; “)”) = \neg(\mathcal{I}s) \quad \wedge \\
&\qquad \mathcal{I}(“\forall”; “(”; s; “)”) = \forall(\mathcal{I}s) \quad \wedge \\
&\qquad \mathcal{I}(“\exists”; “(”; s; “)”) = \exists(\mathcal{I}s) \quad \wedge \\
&\qquad \mathcal{I}(“-”; “(”; s; “)”) = -(\mathcal{I}s) \\
&\forall s, t : lang \cdot \mathcal{I}(“(”; s; “)”; “\wedge”; “(”; t; “)”) = (\mathcal{I}s) \wedge (\mathcal{I}t) \quad \wedge \\
&\qquad \mathcal{I}(“(”; s; “)”; “=”; “(”; t; “)”) = (\mathcal{I}s) = (\mathcal{I}t) \quad \wedge \\
&\qquad \mathcal{I}(“(”; s; “)”; “\Rightarrow”; “(”; t; “)”) = (\mathcal{I}s) \Rightarrow (\mathcal{I}t) \wedge \\
&\qquad \mathcal{I}(“(”; s; “)”; “\Leftarrow”; “(”; t; “)”) = (\mathcal{I}s) \Leftarrow (\mathcal{I}t) \wedge \\
&\qquad \mathcal{I}(“(”; s; “)”; \ “;” \ ; “(”; t; “)”) = (\mathcal{I}s); (\mathcal{I}t) \quad \wedge \\
&\qquad \mathcal{I}(“(”; s; “)”; “-”; “(”; t; “)”) = (\mathcal{I}s) - (\mathcal{I}t) \quad \wedge \\
&\qquad \mathcal{I}(“(”; s; “)”; “\ ”; “(”; t; “)”) = (\mathcal{I}s) \ \ (\mathcal{I}t) \\
&\forall\, s : *char \cdot \mathcal{I}(“\backslash“”; s; “\backslash””) = s
\end{aligned}
\tag{14}
$$

To save space we leave out the interpretation of each digit. For scopes the introduced variable must be an identifier, and the expression $\mathcal{I}a$ in that position would not satisfy this requirement. We instead have a law for only the identifier a, and other identifiers can be obtained through an application of a renaming law. We add character brackets to these laws in order to avoid precedence issues.

Note that we defined $lang$ as a bunch of texts, and not the expressions themselves. When these texts are interpreted, the results are expressions or values in the language. The text “2” is in $lang$, but not the value 2. The interpreter is similar to a function of strings and distributes over bunch union. It is of course possible to have a logical language to parallel the texts in $lang$; all the expressions in the language which do not contain $\mathcal{I}$ can then be denoted as $\mathcal{I}\, lang$. In this paper we leave out some operators from $lang$, such as the ones in bunch theory.

The interpreter is similar to a traditional semantic valuation function, with a few differences. First, the interpreter is a way of encoding meta-logic within the logic itself; no extra meta-logic is required. Second, the interpreter does not necessarily map every string in the language to a value. Rather, we later introduce generic laws that reason with such expressions directly. Lastly, we will show how the interpreter can be included in the interpreted language without inconsistency.

A. Variables

One significant change that we allow in our logic is for variables. The interpreter needs to refer to an infinite collection of strings that represent variables, and there is no simple way to refer to all variables themselves. We would like the interpretation of a string representing a variable to be the variable that is represented:

$$\begin{aligned} &\mathcal{I}\text{“}a\text{”} = a \\ &\mathcal{I}\text{“}b\text{”} = b \\ &\cdots \end{aligned} \tag{15}$$

We instead say that a variable with the name a is an abbreviation for $\mathcal{I}$“a”, and similarly for all other variable names. Although in our initial definition we excluded the interpreter from the interpreted strings, we later show in VII how we can extend our language to safely include the interpreter.

There is an important consequence of making variable syntax more expressive: function application and variable instantiation is no longer a decidable procedure in general. This is because deciding whether two variable strings are equal is now as difficult as all of proving. However, this does not pose a problem for the implementation of function application along with the interpreter in a theorem prover. The simple solution is that whenever we see an interpreter in the body, we simply do not apply the function. We argue that this rarely hinders the use of the interpreter, since users can do all calculation in the sub-language that does not include the interpreter, exactly as before. In the case where reasoning with the interpreter is desired, standard proof obligations can be generated and discharged.

We finish this section by noting that we could have simplified the definition considerably if we had a fully parenthesized prefix language. All operator interpretation could be compressed to a single law, and some bracket characters removed.

V. Resolving Partial Terms With the Interpreter

Our solution to reasoning with partial terms is neither at the term or propositional level. We rather say that some operators, such as equality or bunch inclusion are generic. For example, here are two of the generic laws for equality.

$$\forall a, b : lang \cdot \mathcal{I}\,(a; \text{“}=\text{”}; b) : bool \qquad \text{Boolean Equality} \tag{16}$$

$$\forall a : \mathcal{I}\,lang \cdot a = a \qquad \text{Reflexivity} \tag{17}$$

The first law says that any equality is a boolean expression, similarly to the Excluded Fourth Law in LPF which implies an equality is either true, false or undefined [1]. The arguments can be any expressions in the interpreted language. For a simple formal example of the use of the law we continue with the difference example.

$$\begin{aligned} &\mathbf{F} \Rightarrow \mathit{diff}\,1\,2 = -1 && \text{Bool Base Law} \\ &\quad \text{Type Checking Proof Obligation} \\ &\quad (\mathit{diff}\,1\,2 = -1) : \mathit{bool} && \text{Interpreter laws} \\ &\quad = \mathcal{I}\text{“}\mathit{diff}\,1\,2 = -1\text{”} : \mathit{bool} && \text{String Assoc.} \\ &\quad = \mathcal{I}(\text{“}\mathit{diff}\,1\,2\text{”}; \text{“}=\text{”}; \text{“}-1\text{”}) : \mathit{bool} && \text{Bool Equality} \\ &\quad = \mathbf{T} \\ &= \mathbf{T} \end{aligned} \tag{18}$$

As we can see in the example, since the interpreter unquotes expressions, using it in proofs is usually just the reverse process.

A. Implementation

In general, implementing laws that use the interpreter in a theorem prover is non-trivial. This is because it is difficult to determine if unification alone is sufficient to check if a law applies. We deliberately wrote two equality laws differently to illustrate a couple cases where this task can be made easy. If the only place the interpreter appears in a law is the expression $\mathcal{I}\,lang$ in the domain of a variable, it can be treated as a generic type. Type checking can be done by scanning to see that the interpreter does not appear in any instantiated expression with a generic type. In the case of the second law, instantiating the variables and parsing yields a valid expression without any further computation.

VI. Metalogical Reasoning Within the Logic

There are several benefits of defining the interpreter and using it to create laws. One such benefit is the creation of generic laws, where type-checking for variables is not necessary. The removal of type-checking is not only beneficial for simplicity, partiality, and efficiency, but some operators are meant to be truly generic. For example, the left operand of the set-membership operator ($\in$) can be any expression in the language, and set brackets can be placed around any expression. By including the interpreter in the logic these laws are expressed with full formality. For sets, an example would be

$$\forall A, B : \mathcal{I}\,lang \cdot (\{A\} = \{B\}) = (A = B) \tag{19}$$

Another benefit is compact laws. For example, we wish to define a generic symmetry law for natural arithmetic in our logic. If we had a prefix notation then we could have written it like this.

$$\forall f : (+, \times, =) \cdot \forall a, b : nat \cdot = (f\,a\,b)(f\,b\,a) \tag{20}$$

Using the interpreter we can create a law in a similar fashion for non-prefix notation.

$$\begin{aligned} &\forall f : \text{“}+\text{”}, \text{“}\times\text{”}, \text{“}=\text{”} \cdot \forall a, b : lang \cdot \\ &\qquad \mathcal{I}\,(a, b) : nat \Rightarrow \mathcal{I}\,(a; f; b) = \mathcal{I}\,(b; f; a) \end{aligned} \tag{21}$$

This law can be made completely generic and include more than arithmetic operators. It even becomes simpler to write.

$$\forall f : \text{“}+\text{”}, \text{“}\times\text{”}, \text{“}\wedge\text{”}, \text{“}\vee\text{”}, \text{“}=\text{”} \cdot \forall a, b : lang \cdot \tag{22}$$
$$\mathcal{I}\,(a; f; b) = \mathcal{I}\,(b; f; a) \tag{23}$$

These sorts of laws allow us to capture an idea like associativity or commutativity in a compact way, and can be easily extended by concatenating to the operator text.

One of the most useful features of the interpreter is reasoning about the syntactic structure of an expression without requiring a meta-logic. These laws include function application and several programming laws. Some laws have caveats, such as requiring that in some expressions certain variables or operators do not appear. For example, there is a quantifier law for $\forall$ that says if the variable a does not appear free in P.

$$(\forall a : D \cdot P) = P \tag{24}$$

We would like to formalize this caveat. It is straight forward to write a program that checks variable or operator appearance in a string (respecting scope). We formalize a specification of the “no free variable” requirement using the interpreter. For simplicity, assume that variables are single characters, and strings are not in the interpreted language. For a string P in our language and a variable named a we specify

$$\begin{aligned}&\exists i : (0..\leftrightarrow P) \cdot P_i = \text{“}a\text{”} \quad \wedge \\ &\quad \neg\exists s, t, D, pre, post : *char \cdot \\ &\qquad (pre; \text{“}\langle a : \text{”}; D; \text{“}\rightarrow\text{”}; s; P_i; t; \text{“}\rangle\text{”}; post) = P \\ &\qquad \vee\, (pre; \text{“}\langle\text{”}; P_i; D; \text{“}\rightarrow\text{”}; s; \text{“}\rangle\text{”}; post) = P\end{aligned} \tag{25}$$

This specification says that a is free in P. The first part says that there is an index i in P at which a appears. The second part says that a is not local. Let $free$ denote this specification parameterized for an expression and a variable; $free\ \text{“}a\text{”}\ P$ says that a is free in P. The caveat for the quantifier law is formalized as

$$\neg(free\ \text{“}a\text{”}\ P) \quad \Rightarrow \quad \mathcal{I}\,(\text{“}\forall a : \text{”}; D; \text{“}\cdot\text{”}; P) = \mathcal{I}\,P \tag{26}$$

In a similar manner we can avoid including axiom schemas in some theories and have just a single axiom. The notation allows us to refer to all variables in an expression.

VII. Including the Interpreter

So far we have excluded the interpreter itself from the interpreted language to maintain consistency. Gödel’s First Incompleteness Theorem implies that we could never define our interpreter to be both consistent and complete [14], [15]. Let the § symbol denote bunch comprehension, and be treated as a quantifier; when applied to a function it returns a bunch. Then there are expressions such as $\{\S x : \mathcal{I}\,lang \cdot \neg(x \in x)\}$ whose string representation we cannot interpret (interpreting this expression in particular causes Russell’s paradox). A simpler proof of Gödel’s theorem by [13] shows why a straight-forward inclusion of the interpreter is inconsistent. However, as [13] also suggests, any logic can be completely described by another. This point is intuitively manifested in the fact that all expressions that cannot be interpreted include the interpreter itself. In a sense, we relegate all issues of partiality in our logic to involve only the interpreter.

However, we can weaken the restriction on the interpreter being excluded from the language. The motivation for including the interpreter is to reason about languages that allow this sort of self reference. In practice, theorem provers such Coq [16] allow reflection as a proving technique. We would like to use the interpreter as a simple way of reasoning about termination and consistency of definitions. The key insight is that a mathematical function disregards computation time. The domain $xnat$ is the naturals extended with ∞. We can measure computation time recursively by defining the following timing function.

$$\mathcal{T} = \langle s : lang \rightarrow \mathbf{result}\ r : xnat \cdot \tag{27}$$

$$\begin{aligned}&\mathbf{var}\ f : (char \rightarrow bool) := \langle t : lang \rightarrow \text{“}\backslash\text{“”}; t; \text{“}\backslash\text{””} = s\rangle \cdot \\ &\mathbf{if}\ \exists f\ \mathbf{then}\ r := 1\ \mathbf{else}\end{aligned}$$

$$\begin{aligned}&\mathbf{var}\ f : (lang \rightarrow bool) := \langle t : lang \rightarrow \text{“}\neg\text{”}; t = s\rangle \cdot \\ &\mathbf{if}\ \exists f\ \mathbf{then}\ r := 1 + \mathcal{T}(\S f)\ \mathbf{else}\end{aligned}$$

$$\begin{aligned}&\mathbf{var}\ f : (lang \rightarrow lang \rightarrow bool) := \\ &\qquad \langle t, t' : lang \rightarrow t; \text{“}\wedge\text{”}; t' = s\rangle \cdot \\ &\mathbf{if}\ \exists f\ \mathbf{then}\ r := 1 + \mathcal{T}(\S t : lang \cdot \exists t' : lang \cdot f\,t\,t') \\ &\qquad + \mathcal{T}(\S t' : lang \cdot \exists t : lang \cdot f\,t\,t')\ \mathbf{else} \\ &\quad \vdots \\ &\mathbf{var}\ f : (lang \rightarrow bool) := \langle t : lang \rightarrow \text{“}\mathcal{I}\text{”}; t = s\rangle \cdot \\ &\mathbf{if}\ \exists f\ \mathbf{then}\ r := 1 + \mathcal{T}(\mathbf{if}\ \S f : var \wedge \mathcal{I}(\S f) : lang \\ &\qquad \mathbf{then}\ \mathcal{I}(\S f)\ \mathbf{else}\ \S f\ \mathbf{fi})\ \mathbf{else} \\ &\quad r := \infty \\ &\mathbf{fi} \\ &\rangle\end{aligned}$$

This function is in a way parallel to how an interpretation works, except that it counts time. The time in question is the number of law applications needed to simplify an expression to have no interpreter symbol in it. At each if-statement the function checks for the occurrence of a certain piece of syntax, and the vertical ellipsis would include a similar check for the rest of the syntax. The special part of this function is when we see the interpreter symbol. If the interpreter was applied to a string representing a variable, and that variable’s value is a string in the language, we recurse on its value. If the interpreter is applied to any other expression, we recurse on

that expression's string representation. For example, if we have

$$Q = \text{“}\neg\mathcal{I}Q\text{”} \tag{28}$$

then we calculate

$$\begin{aligned} &\mathcal{T}Q \\ &= \mathcal{T}\text{“}\neg\mathcal{I}Q\text{”} \\ &= 1 + \mathcal{T}\text{“}\mathcal{I}Q\text{”} \\ &= 1 + \mathcal{T}Q \end{aligned} \tag{29}$$

and therefore $\mathcal{T}Q = \infty$ since $\mathcal{T}Q : xnat$. For any string that does not include the interpreter the time is linear; this can be proven by structural induction over *lang* if we add an induction axiom along with the construction axioms we defined earlier. We should only interpret an expression that includes the interpreter if the execution time of the interpretation is finite. If it is infinite or cannot be determined, then there is a potential for inconsistency had we decided to interpret it regardless. We can add the interpreter to the interpreted language as follows:

$$\forall s : lang \cdot \mathcal{T}s < \infty \Rightarrow \mathcal{I}\text{“}\mathcal{I}\backslash\text{“”}; s; \text{“}\backslash\text{””} = \mathcal{I}s \tag{30}$$

This also implies that interpreting variables in their unabbreviated form is also safe. If we have $s : var$ then:

$$\begin{aligned} &\mathcal{T}\text{“}\mathcal{I}\text{”}; s \\ &= 1 + \mathcal{T}s \\ &= 1 + 1 \\ &= 2 \end{aligned} \tag{31}$$

In general, proving a finite execution time is the halting problem. When reasoning about logics it may be useful to include the interpreter in the interpreted language. For many practical purposes it can be left out.

VIII. Proof of Consistency

To prove the interpreter consistent we will find a model in set theory. Characters are implemented as natural numbers, having

$$\text{“0”} = 0,\ \ldots\ \text{“9”} = 9,\ \text{“}a\text{”} = 10,\ \ldots\ \text{“}z\text{”} = 25,\ \ldots \tag{32}$$

Strings are implemented as ordered pairs in the standard way.

$$a; b = \{\{a\}, \{a, b\}\} \tag{33}$$

The interpreter is a mapping from the set of all strings in our language $lang$ to the class of all sets. $\mathcal{I} \subseteq lang \times Sets$. It is assumed that all other theories (functions, boolean algebra, numbers) are implemented in set theory in the standard way. For this reason partial functions might be implemented using another special value that all remaining domain elements will be mapped to. We will not delve into the implementation of functions and other theories, since once they are implemented in set theory, they are included in the class $Sets$.

We must prove that there exists a function $\mathcal{I}$ such that the interpreter axioms are true. The recursion theorem will be used to prove this [5]. The theorem states that given a set X, an element a of X, and a function $f : X \to X$ there exists a unique function F such that

$$F\,0 = a \tag{34}$$

$$\forall n : nat \cdot F\,(n+1) = f(F\,n) \tag{35}$$

Since $\mathcal{I} \subseteq lang \times Sets$ it is necessary to first find a function from $lang$ to the naturals; this is an enumeration of the strings in $lang$. Let $charNum$ be the total number of characters in $char$. Character string comparison for strings s, t is defined as

$$(s > t) = strNum(s) > strNum(t) \tag{36}$$

$$\begin{aligned} strNum = \langle S : *char \to \textbf{if}\ S = nil\ \textbf{then}\ 0 \\ \textbf{else}\ S_0 + charNum \times S_{1..\leftrightarrow S}\ \textbf{fi}\rangle \end{aligned} \tag{37}$$

The enumeration function $enum$ of strings in $lang$ is defined as

$$enum = (g^{-1}) \tag{38}$$

$$\begin{aligned} g = \langle n : nat \to \textbf{if}\ n = 0\ \textbf{then}\ (MIN\ s : lang \cdot s) \\ \textbf{else}\ (MIN\ s : (\S t : lang \cdot t > g(n-1)) \cdot s)\ \textbf{fi}\rangle \end{aligned} \tag{39}$$

The function $strNum$ assigns a unique number to each string. Some character strings are not in $lang$, and we desire an enumeration free from gaps. The function g assigns a unique string in $lang$ to each natural as follows: zero is mapped to the first string in the language, and each subsequent number is mapped to the next smallest string. Since g is one-to-one, we define $enum$ as its inverse. We define function F for a given state in the model with finite single-character variables as

$$\begin{aligned} &F\,0 = \{0; 0\} \\ &\vdots \\ &F\,9 = \{9; 9\} \cup F\,8 \end{aligned} \tag{40}$$

$$\begin{aligned} &\text{For all } s : char \text{ let } m = enum(\text{“}\backslash\text{“”}; s; \text{“}\backslash\text{””}) \text{ in} \\ &F\,m = \{m; s\} \cup F\,(m-1) \end{aligned} \tag{41}$$

$$\begin{aligned} &F\,(n+1) = \{(n+1); (H\,(n+1)\,(F\,n))\} \cup F\,n \\ &(H \text{ is defined below}) \end{aligned} \tag{42}$$

At each argument n function F is a mapping of all previous numbers to their corresponding expressions, in addition to the current one. The base elements are the variables, numbers and strings. Function H constructs expressions using the operators in our language from previous expressions. It is defined as

follows.

$$
\begin{aligned}
&H\,k\,I = \qquad (43)\\
&\{S : Sets | \exists n, m : \mathbf{dom}(I) \cdot g\,k = (g\,n); \text{“}+\text{”}; (g\,m)\\
&\qquad\qquad \wedge S = I\,n + I\,m\} \cup\\
&\{S : Sets | \exists n, m : \mathbf{dom}(I) \cdot g\,k = (g\,n); \text{“}\wedge\text{”}; (g\,m)\\
&\qquad\qquad \wedge S = I\,n \wedge I\,m\} \cup\\
&\{S : Sets | \exists n : \mathbf{dom}(I) \cdot g\,k = \text{“}\neg\text{”}; (g\,n) \wedge S = \neg I\,n\} \cup\\
&\vdots
\end{aligned}
$$

Like the timing function, the vertical ellipsis represents a similar treatment for other operators and is used to save space. Since g is one-to-one, only a single set in this union will have an element in it. In other words, each number is mapped to a single expression (but not vice-versa). Finally, the interpreter is implemented as follows.

$$\mathcal{I} = \langle s : lang \to (F\ (enum\,s))\,(enum\,s)\rangle \qquad (44)$$

IX. Conclusion

We have presented the formalism of an expression interpreter for the purpose of reasoning with partial terms. Our technique requires no separate meta-logic, and we believe that our encoding of expressions as character strings is simple and transparent. The use of the interpreter allows proofs with partial terms to proceed in a fully formal fashion classically; that is, with just the standard boolean algebra. We show how the interpreter can be used to create generic and compact laws, which also allow syntactic reasoning about expressions. We also argue that the incorporation of the interpreter in theorem provers is simple, since the parsing that is required for its use is an efficient linear-time algorithm.

References

[1] C. B. Jones and C. A. Middelburg, “A typed logic of partial functions reconstructed classically,” *ACTA*, vol. 31, no. 5, pp. 399–430, 1994.

[2] C. C. Morgan, *Programming from specifications, 2nd Edition*. Upper Saddle River, NJ, USA: Prentice Hall, 1994.

[3] C. B. Jones, M. J. Lovert, and L. J. Steggles, “A semantic analysis of logics that cope with partial terms,” in *ABZ*, ser. LNCS, J. Derrick, J. A. Fitzgerald, S. Gnesi, S. Khurshid, M. Leuschel, S. Reeves, and E. Riccobene, Eds., vol. 7316. Springer, 2012, pp. 252–265.

[4] C. B. Jones, “Partial functions and logics: A warning,” *IPL*, vol. 54, no. 2, pp. 65–67, 1995.

[5] W. Just and M. Weese, *Discovering Modern Set Theory. I*. American Mathematical Society, 1996, vol. 8.

[6] M. Beeson, *Foundations of Constructive Mathematics*. New York, NY, USA: Springer-Verlag, 1985.

[7] ——, “Lambda logic,” in *Automated Reasoning: Second International Joint Conference, IJCAR 2004*. Springer, 2004, pp. 4–8.

[8] R. F. Stärk, “Why the constant ’undefined’? logics of partial terms for strict and non-strict functional programming languages,” *J. Funct. Program.*, vol. 8, no. 2, pp. 97–129, 1998.

[9] R. D. Gumb, “The lazy logic of partial terms,” *JSYML*, vol. 67, no. 3, pp. 1065–1077, 2002.

[10] M. J. C. Gordon, R. Milner, and C. P. Wadsworth, *Edinburgh LCF*, ser. Lecture Notes in Computer Science. Springer, 1979, vol. 78.

[11] D. Gries, *The Science of Programming*. New York: Springer-Verlang, 1981.

[12] E. C. R. Hehner, *A Practical Theory of Programming*. New York: Springer, 1993. [Online]. Available: http://www.cs.toronto.edu/ hehner/aPToP/

[13] ——, “Beautifying gödel,” pp. 163–172, 1990.

[14] K. Gödel, “Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme,” *Monatshefte für Mathematik und Physik*, vol. 38, no. 1, pp. 173–198, 1931.

[15] R. Zach, “Kurt gödel and computability theory,” in *Logical Approaches to Computational Barriers*, ser. Lecture Notes in Computer Science, A. Beckmann, U. Berger, B. Löwe, and J. Tucker, Eds. Springer Berlin Heidelberg, 2006, vol. 3988, pp. 575–583.

[16] T. C. D. Team, “The coq proof assistant reference manual,” 2009.