Applications of Languages with Self-Interpreters to Partial Terms and Functional Programming

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

***Abstract* —Those programming languages that contain self-interpreters have the added power of reflection, and allow dynamically controlling execution. In a logical language a complete self-interpreter is necessarily inconsistent. However, we demonstrate a logical language with a reasonably complete self-interpreter. We argue for its use as a simple formalism for reasoning about partial terms, and functional languages that allow both general recursion and dependent types. Since refinements of programming specifications often include partial terms, they need to be handled using formal rules. Likewise, we show formal rules for handling general recursion consistently in a simple language. Moreover, we demonstrate how to use an interpreter to reason about lazy evaluation. We argue that the interpreter can be integrated within theorem provers.**

Keywords* — *logic; partial-terms; theorem prover; two-valued logic; expression interpreter; functional programming; general recursion; lazy evaluation

I. Introduction

In this paper we argue that logics for programming must be able to cope with general recursion and partial terms, and that an interpreter [1] is a viable solution.

General recursive and partially recursive functions naturally occur in programs. This is an effect of recursively defined datatypes such as trees, and the computation paths that arise in sufficiently complex programs. Aside from necessary complexity while programming, partially recursive functions are often the result of computation that is non-terminating. While often this indicates programmer error is is undesirable, there are many cases where a non-terminating program is intentional. This includes any application code that waits and responds to user input, and some semi-decision procedures. Therefore, general recursion is not a property that should be excluded, but rather desirable in a functional language for the ease of use of programmers.

Despite their power and expressiveness, general and partially recursive functions pose challenges for a number of theories of programming and practical tools. The issues present themselves both in the difficulty of proof of their properties and with the possible partiality or at worst inconsistency that they introduce. For example, the language Gallina within the interactive theorem prover Coq [2] requires that all functions terminate. Non-terminating programs would introduce logical inconsistency. Likewise, theories of programming such as Morgan's Programming from Specifications [3] exclude non-terminating programs from its standard theory and require a proof of termination for all programs. In the Vienna Development Method [4] non-terminating computation corresponds to partial functions. In a well-typed or dependently typed language this partiality naturally arises when a function is required to produce a certain result for a pre-condition stronger than **true**.

The occurrence of partial terms are not limited to non-terminating functions. There are often cases where expressions within programs do not denote a value. For example the indexing of a sequence with a negative number results in an error in most programming languages. Formal reasoning about partial expressions often occurs when using a formal programming theory, even one that requires termination. Both issues of partial terms and general recursion pose a similar type of problems to a theory: they produce expressions to which no formal rules apply, and make proofs of intuitively simple theorems impossible. At worst, the result is inconsistency.

It is crucial for a formal program theory to consistently and elegantly deal with general recursion and partial terms, introducing the minimal amount of extra values and extra theory to do so. The interpreter formalism can be applied for reasoning about both of these features, and is an extension of the interpreter presented in [1].

A. General Recursion

For formal typed functional languages recursion is often restricted. This is because logically reasoning about functions with both constructive types and general recursion is inconsistent. For example, from the following definition:

$$f : nat \to nat \tag{1}$$

$$f = \lambda x : nat \cdot 1 + f\, x \tag{2}$$

It is immediately clear that $f\, n = f\, n + 1$, and since $f\, n : nat$ we have the contradiction that $1 = 0$. There are a number of methods to maintain consistency that either restrict recursion, require functions to be constructive, or require proof that both argument and result of a function is a value [5]. A constructive type theory is desirable in order to perform effective error checking statically. Dependent types are more expressive, and allow type information to encode invariants.

B. Partial Terms

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 [6], 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).

$$\mathit{diff}\, i\, j = \mathbf{if}\, i = j\, \mathbf{then}\, 0\, \mathbf{else}\, (\mathit{diff}\, i\, (j+1)) + 1\, \mathbf{fi} \tag{3}$$

We would like to prove

$$\forall i, j : \mathit{int} \cdot i \geq j \Rightarrow (\mathit{diff}\, i\, j) = i - j \tag{4}$$

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

$$\begin{array}{rcl} 1 \geq 2 & \Rightarrow & (\mathit{diff}\, 1\, 2) = 1 - 2 \\ = \mathbf{F} & \Rightarrow & (\mathit{diff}\, 1\, 2) = -1 \end{array} \tag{5}$$

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 Section 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 VIII 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 $\forall x : \mathit{int} \cdot x/0 : \mathit{int}$. This is the approach used in the programming theory of [3]. 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.

TABLE I
Three-Valued Boolean Operators

	T	**F**	$\perp$
$\neg$	**F**	**T**	$\perp$

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

$$\begin{array}{rl} & 0 \\ = & 0 \times (1/0) \\ = & 1 \times (0/0) \\ = & 0/0 \end{array} \tag{6}$$

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 [7] cannot be allowed due to inconsistency.

In [8], 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 $\perp$ [9]. The denotational semantics for a generic law for equality are extended with 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) [10], [11] 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) [12] 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 [13] include 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 [7] with more than two values. In these logics the truth table of boolean operators is usually extended as in Table I (where $\perp$ 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 $\perp$. 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) [14] there is a $\perp_t$ 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 [15]. 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.

For many of these non-classical logics the authors of [16] demonstrate a relationship, and how to transform undefined terms in one logic to another in a similar method to data-refinement.

III. Background Theories

We introduce two theories from [17] 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. A number, character or boolean is an element. Every expression is a bunch, but not all bunches are elementary. Here are some bunch operators.

$$A \,,\, B \qquad \text{A union B} \tag{7}$$
$$A \,`\, B \qquad \text{A intersect B} \tag{8}$$
$$A : B \qquad \text{A in B, or A included in B} \tag{9}$$
$$¢A \qquad \text{cadinality of A} \tag{10}$$

If x is an element, then $¢x = 1$. The empty bunch, whose cardinality equals zero, is the constant $null$. The union of two elements x, y is not an element iff $x`y = null$. Both bunch union and intersection are symmetric, associative, and idempotent. The definition of bunches essentially gives algebraic properties to a comma as an operator. 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

$$\begin{aligned} & 1 + (4, 7) \\ &= 1 + 4 \ , \ 1 + 7 \\ &= 5 \,, 8 \end{aligned} \tag{11}$$

Bunch comprehension is denoted with the section sign §. For element x, bunches A and B, and predicate f, § is defined as follows:

$$(§v : null \cdot f\, v) \quad = \quad null \tag{12}$$
$$(§v : x \cdot f\, v) \quad = \quad \mathbf{if}\, f\, x \,\mathbf{then}\, x \,\mathbf{else}\, null \tag{13}$$
$$(§v : A, B \cdot f\, v) \quad = \quad (§v : A \cdot f\, v), (§v : B \cdot f\, v) \tag{14}$$

Where nat is the bunch of naturals, we define the notation $x, ..y$, read as "x to y" as

$$x, ..y \ = \ §i : nat \cdot x \leq i < y \tag{15}$$

Bunch distribution is similar to a cross-product in set theory. Sets do not distribute over bunch union, and set brackets can be placed around a bunch to form a set (which itself is an element). For example, $\{null\}$ is the empty set, and $¢\{null\} = 1$. Set comprehension $\{x : D | f\, x\}$ is an abbreviation for $\{§x : D \cdot f\, x\}$.

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.

$$0*S = nil \tag{16}$$
$$\forall\, n : nat \cdot (n + 1)*S = (n*S); S \tag{17}$$

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

$$S_n \qquad \text{S at index n} \tag{18}$$
$$\leftrightarrow S \qquad \text{length of S} \tag{19}$$

A semicolon distributes over bunch union, and so does an asterisk in the left operand. Similarly to bunches, a number, character, or boolean is an item. If x is an item, then $\leftrightarrow x = 1$, and only the string nil has length zero. The concatenation of two items is not an item. Note that $null$ is not an item, and that $\leftrightarrow null = null$. Operators and functions, whose domains include only items, distribute over concatenation. For example

$$(0; 2; 4) + 1 \ = \ 1; 3; 5 \tag{20}$$

Since a string S can be thought of as a function that maps natural numbers from $0, .. \leftrightarrow S$ to the items of S, quantifiers can be lifted to apply to strings.

$$\Sigma S \ = \ \Sigma n : 0, .. \leftrightarrow S \cdot S_n \tag{21}$$
$$\forall S \ = \ \forall n : 0, .. \leftrightarrow S \cdot S_n \tag{22}$$

Analogously to the bunch notation $x, ..y$ the notation $x; ..y$ for items x, y, z is defined as

$$x; ..x = nil \tag{23}$$
$$x; ..(x + 1) = x \tag{24}$$
$$(x; ..y) \ ; \ (y; ..z) = x; ..z \tag{25}$$

Similarly to the relationship between sets and bunches, strings can be pakcaged into lists. Lists are denoted with square brackets, and operators of lists are

$$[S] \qquad \text{List containing S} \tag{26}$$
$$[S] + [T] \ = \ [S; T] \qquad \text{List Concatenation} \tag{27}$$
$$[S]n \ = \ [S_n] \qquad \text{List Indexing} \tag{28}$$

Some examples of the operators defined are

$$\begin{aligned} \leftrightarrow (7; 1; 0) &= 3 \\ (7; 1; 0)_0 &= 7 \\ 1; (5, 17); 0 &= (1; 5; 0), (1; 17; 0) \\ 3{*}(0; 1) &= 0; 1; 0; 1; 0; 1 \\ (0, 1){*}(0; 1) &= 0{*}(0; 1), 1{*}(0; 1) = nil, 0; 1 \end{aligned} \tag{29}$$

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". Let 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. This built-in distributivity comes at a cost: the cost is that at times it is required to prove that some bunches are elements. Consider the bunch $bool$ defined as $bool = \mathbf{T}, \mathbf{F}$. Then we prove

$$\neg bool \tag{30}$$
$$= \neg(\mathbf{T}, \mathbf{F}) \tag{31}$$
$$= \mathbf{F}, \mathbf{T} \tag{32}$$
$$= bool \tag{33}$$

However, this does not mean that bunch theory is inconsistent. In order to simplify $bool = \neg bool$ to $\mathbf{F}$, we must know that $bool$ is an element.

IV. Defining the Interpreter

An interpreter is very similar to a semantic valuation function, except that it does not require a universe of values. In addition, it will be extended to interpret a language that includes the interpreter symbol itself, which if naïvely done causes inconsistency. While standard notation does not explicitly distinguish logic from meta-logic (unless different operators are used), we put quotes around the interpreted language. Since the interpreter essentially encodes meta-logic, we would like to keep it simple in the sense that it should introduce as few new operators as possible. It should also preserve the properties of existing operators. In this way we both avoid a separate meta-language, and 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. Where as multiple level of meta-logic might require multiple sets of symbols, reasoning with the interpreter only requires adding more quotes. We take the idea of the character-string predicate of Hehner [18], and we extend it to be a general interpreter for any expression in our language. To maintain consistency we exclude the interpreter itself from the interpreted language in this section. 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 nat be the bunch of naturals. We have the following definitions

$$\begin{aligned} digit &= \text{“0”, “1”, “2”, “3”, “4”,} \\ &\quad \text{“5”, “6”, “7”, “8”, “9”} && (34) \\ var &= alpha; {*}alpha && (35) \\ num &= digit; {*}digit && (36) \\ uniops &= \text{“}\neg\text{”, “}-\text{”, “}\forall\text{”, “}\exists\text{”, “}\Sigma\text{”} && (37) \\ binops &= \text{“}=\text{”, “}\wedge\text{”, “}\vee\text{”, “}\Rightarrow\text{”, “}\Leftarrow\text{”,} && (38) \\ &\quad \text{“;”, “}-\text{”, “}+\text{”, “ ”, “}\in\text{”} \\ charstring &= \text{“\“”}; {*}char; \text{“\“”} && (39) \end{aligned}$$

We define our language $lang$ to be the following bunch of strings.

$$
\begin{aligned}
&var, num, string, \text{“}\mathbf{T}\text{”}, \text{“}\mathbf{F}\text{”} : lang \\
&\text{“}\langle\text{”}; var; \text{“ : ”}; lang; \text{“} \to \text{”}; lang; \text{“}\rangle\text{”} : lang \\
&\text{“(”}; lang; \text{“)”} : lang \\
&\text{“\{”}; lang; \text{“\}”} : lang \\
&\text{“(”}; uniops; lang; \text{“)”} : lang \\
&\text{“(”}; lang; binops; lang; \text{“)”} : lang
\end{aligned} \tag{40}
$$

Here we have defined a language that includes boolean algebra, numbers, logical quantifiers, functions, and strings. This language is fully bracketed for simpler laws, and non-bracketed expressions should be read as abbreviations. 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 (character string) 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}\ \text{“}\mathbf{T}\text{”} &= \mathbf{T} \\
\mathcal{I}\ \text{“}\mathbf{F}\text{”} &= \mathbf{F} \\
\forall\, s : num \cdot \forall\, d : digit \cdot \mathcal{I}(s; d) &= (\mathcal{I}\, s) \times 10 + (\mathcal{I}\, d) \\
\forall\, s, t : lang \cdot \mathcal{I}\,(\text{“}\langle a : \text{”}; s; \text{“} \to \text{”}; t; \text{“}\rangle\text{”}) &= \langle a : \mathcal{I}\, s \to \mathcal{I}\, t \rangle \\
\forall\, s : lang \cdot\ \mathcal{I}\,\text{“(”}; s; \text{“)”} &= \mathcal{I}\, s \ \wedge \\
\mathcal{I}\,\text{“\{”}; s; \text{“\}”} &= \{\mathcal{I}\, s\} \ \wedge \\
\mathcal{I}\,(\text{“}\neg\text{”}; s) &= \neg(\mathcal{I}\, s) \ \wedge \\
\mathcal{I}\,(\text{“} - \text{”}; s) &= -(\mathcal{I}\, s) \ \wedge \\
\mathcal{I}\,(\text{“}\forall\text{”}; s) &= \forall(\mathcal{I}\, s) \ \wedge \\
\mathcal{I}\,(\text{“}\exists\text{”}; s) &= \exists(\mathcal{I}\, s) \ \wedge \\
\mathcal{I}\,(\text{“}\Sigma\text{”}; s) &= \Sigma(\mathcal{I}\, s) \\
\forall s, t : lang \cdot \mathcal{I}\,(s; \text{“} = \text{”}; t) &= (\mathcal{I}\, s) \wedge (\mathcal{I} t) \ \wedge \\
\mathcal{I}\,(s; \text{“} \wedge \text{”}; t) &= (\mathcal{I}\, s) = (\mathcal{I}\, t) \ \wedge \\
\mathcal{I}\,(s; \text{“} \vee \text{”}; t) &= (\mathcal{I}\, s) = (\mathcal{I}\, t) \ \wedge \\
\mathcal{I}\,(s; \text{“} \Rightarrow \text{”}; t) &= (\mathcal{I}\, s) \Rightarrow (\mathcal{I}\, t) \ \wedge \\
\mathcal{I}\,(s; \text{“} \Leftarrow \text{”}; t) &= (\mathcal{I}\, s) \Leftarrow (\mathcal{I}\, t) \ \wedge \\
\mathcal{I}\,(s; \text{“;”}; t) &= (\mathcal{I}\, s); (\mathcal{I}\, t) \ \wedge \\
\mathcal{I}\,(s; \text{“} - \text{”}; t) &= (\mathcal{I}\, s) - (\mathcal{I}\, t) \ \wedge \\
\mathcal{I}\,(s; \text{“} + \text{”}; t) &= (\mathcal{I}\, s) + (\mathcal{I}\, t) \ \wedge \\
\mathcal{I}\,(s; \text{“} \in \text{”}; t) &= (\mathcal{I}\, s) \in (\mathcal{I}\, t) \ \wedge \\
\mathcal{I}\,(s; \text{“ ”}; t) &= (\mathcal{I}\, s)\ \ (\mathcal{I}\, t) \\
\forall\, s : {*char} \cdot \mathcal{I}\,(\text{“}\backslash\text{“”}; s; \text{“}\backslash\text{””}) &= s
\end{aligned} \tag{41}
$$

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.

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 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.

Note that unlike a semantic valuation function the interpreter does not necessarily map every string in the language to a value. Rather, we later introduce generic laws that reason with these partial terms 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. We say that a variable with the name a is an abbreviation for $\mathcal{I}\text{“}a\text{”}$, and similarly for all other variable names. Although in our initial definition we excluded the interpreter from the interpreted strings, we later show in Section VIII 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 do not apply the function; it is treated as a syntactic variable that can only be replaced by its interpretation. We argue that this rarely hinders the use of the interpreter, since in the sub-language that does not include the interpreter users can do all calculations 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 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.

$$\forall a, b : lang \cdot \mathcal{I}\,(a; \text{“} = \text{”}; b) : bool \qquad \text{Boolean Equality} \tag{42}$$

$$\forall a : \mathcal{I}\, lang \cdot a = a \qquad \text{Reflexivity} \tag{43}$$

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 [6]. 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.

(44)

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

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

$$\forall A, B : \mathcal{I}\,lang \cdot (\{A\} = \{B\}) = (A = B) \tag{45}$$

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 the law as

$$\forall f : (+, \times, =) \cdot \forall a, b : nat \cdot = (f\,a\,b)(f\,b\,a) \tag{46}$$

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

$$
\begin{aligned}
\forall f : \text{“}+\text{”}, \text{“}\times\text{”}, \text{“}=\text{”} \cdot \forall a, b : lang \cdot & \qquad (47) \\
\mathcal{I}\,(a, b) : nat \Rightarrow \mathcal{I}\,(a; f; b) = \mathcal{I}\,(b; f; a) &
\end{aligned}
$$

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

$$\forall f : \text{“}+\text{”}, \text{“}\times\text{”}, \text{“}\wedge\text{”}, \text{“}\vee\text{”}, \text{“}=\text{”} \cdot \forall a, b : lang \cdot \tag{48}$$

$$\mathcal{I}\,(a; f; b) = \mathcal{I}\,(b; f; a) \tag{49}$$

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. Some further abbreviations can be particularly useful:

$$\forall v \cdot P \;=\; \forall v : \mathcal{I}\,lang \cdot P \tag{50}$$

$$\exists v \cdot P \;=\; \exists v : \mathcal{I}\,lang \cdot P \tag{51}$$

$$\Sigma v \cdot P \;=\; \Sigma v : \mathcal{I}\,lang \cdot P \tag{52}$$

$$\S v \cdot P \;=\; \S v : \mathcal{I}\,lang \cdot P \tag{53}$$

$$\langle v \to P \rangle \;=\; \langle v : \mathcal{I}\,lang \to P \rangle \tag{54}$$

It appears as if the language has unrestricted quantification, comprehension, and domain-less functions. This is useful for generic quantification and generic functions. Of course, the expressions in the domains of the quantifiers and function above must not include the interpreter.

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 then

$$(\forall a : D \cdot P) = P \tag{55}$$

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

$$
\begin{aligned}
&\exists i : (0, .. \leftrightarrow P) \cdot P_i = \text{“}a\text{”} \quad \wedge & (56) \\
&\quad \neg \exists s, t, D, pre, post : {*char} \cdot \\
&\qquad (pre; \text{“}\langle a : \text{”}; D; \text{“}\to\text{”}; s; P_i; t; \text{“}\rangle\text{”}; post) = P \\
&\qquad \vee\, (pre; \text{“}\langle\text{”}; P_i; D; \text{“}\to\text{”}; s; \text{“}\rangle\text{”}; post) = P
\end{aligned}
$$

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; $free$ “a” P

says that a is free in P. The caveat for the quantifier law is formalized as

$$\neg(free\ \text{“}a\text{”}\ P) \quad \Rightarrow \quad \mathcal{I}\,(\text{“}\forall a:\text{”}; D; \text{“}\cdot\text{”}; P) = \mathcal{I}\,P \quad (57)$$

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

VII. Fixed-Points

Quines are self-reproducing expressions; their interpretation is equal to themselves. Fixed-points of the interpreter are then Quines, formally satisfying

$$\mathcal{I}\,Q \;=\; Q \quad (58)$$

A Quine under this definition need not be a program. The following expression is a Quine [17]:

$$\text{“}\backslash\text{“}\backslash\text{“}[0; 2{*}(0, ..15)]\backslash\text{”}[0; 2{*}(0, ..15)]\text{”} \quad (59)$$

Of course, if we have $Q =$ “Q” then it is trivially a Quine, and therefore the definition of Quines is often restricted to expressions with no free variables.

VIII. Including the Interpreter

The definitions above exclude the interpreter itself from the interpreted language to maintain consistency. Gödel's First Incompleteness Theorem implies that it is not possible to define the interpreter to be both consistent and complete [19], [20]. A simpler proof of Gödel's theorem by [18] shows why a straight-forward inclusion of the interpreter by the laws

$$\text{“}\mathcal{I}\text{”}; lang\ :\ lang \quad (60)$$

$$\forall s : lang \cdot \mathcal{I}\ \text{“}\mathcal{I}\text{”}; s \;=\; \mathcal{I}\,s \quad (61)$$

is inconsistent. In that paper the interpreter was defined to be a mapping $lang \to bool$, which is not the case without definitions. Nonetheless, since both completeness and consistency cannot be achieved at once, any consistent logic that includes the interpreter will necessarily include strings that cannot be consistently interpreted, and hence be incomplete. Specifically, let rus be the expression $\{\S x : \mathcal{I}\,lang \cdot \neg(x \in x)\}$. Then rus cannot have its string representation interpreted consistently. The proof is as follows, and is similar to Russell's paradox.

$$\begin{aligned} & rus \in rus \quad (62)\\ &= rus \in \{\S x : \mathcal{I}\,lang \cdot \neg(x \in x)\}\\ &= rus : (\S x : \mathcal{I}\,lang \cdot \neg(x \in x))\\ &= \neg(rus \in rus) \wedge rus : \mathcal{I}\,lang\\ &= \neg(rus \in rus) \wedge \{\S x : \mathcal{I}\,lang \cdot \neg(x \in x)\} : \mathcal{I}\,lang\\ &= \neg(rus \in rus) \end{aligned}$$

Since $rus \in rus$ is boolean and an element, the proof above shows a contradiction in the logic. The interpreter is therefore incomplete for expressions such as rus, but all expressions for which the interpreter is incomplete include the interpreter.

However, as [18] 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 [2] allow reflection as a proving technique. Reflection is a proof technique that allows a program (written in the functional language of Coq) to reason about expressions in Coq syntactically (at a meta-level). For example, a tactic for normalizing variable ordering in arithmetic equations would need to reason about expressions syntactically. It improves the performance of proof search considerably by eliminating the need for a number of applications of a commutative law.

A simple interpreter can be defined for arithmetic expressions in Coq in a straight-forward manner: the semantics of an expression would parallel its syntactic definition. However, such an interpreter must be well-typed, while the interpreter in this paper may fail to denote a value. It is non-trivial to define an interpreter in Coq whose interpreted language includes the interpreter itself.

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 ∞. The domain of both $\mathcal{T}$ and $\mathcal{T}_{\mathcal{I}}$ is nat and their range is $xnat$. We define a parsing function from strings in the language to a tree data structure as follows:

$$\forall v : var, num, string, \text{“}\mathbf{T}\text{”}, \text{“}\mathbf{F}\text{”} \cdot \mathbf{parse}\,v = \mathbf{graft}\,v\,nil$$

$$\forall s, t : lang \cdot \forall v : var \cdot \forall bin : binops \cdot \forall uni : uniops \cdot$$

$$\begin{aligned} \mathbf{parse}\ uni; v &= \mathbf{graft}\ uni\ (\mathbf{parse}\,s)\\ \mathbf{parse}\ s; bin; t &= \mathbf{graft}\ bin\ (\mathbf{parse}\,s); (\mathbf{parse}\,t)\\ \mathbf{parse}\ \text{“(”}; s; \text{“)”} &= \mathbf{graft}\ \text{“()”}\ (\mathbf{parse}\,s)\\ \mathbf{parse}\ \text{“}\langle\text{”}; v; \text{“:”}; s; \text{“}\to\text{”}; t; \text{“}\rangle\text{”} &= \mathbf{graft}\ \text{“}\langle\rangle\text{”}\ (\mathbf{parse}\,s); (\mathbf{parse}\,t) \end{aligned}$$

We can measure interpretation time recursively by defining the following timing functions.

$$
\begin{aligned}
&\mathcal{T}\, nil = \mathcal{T}_{\mathcal{I}}\, nil = 0\\
&\mathcal{T}\, s = \mathcal{T}\,(\mathbf{parse}\, s)\\
&\mathcal{T}\,\mathbf{graft}\, op\, subtrees \quad = \quad \mathbf{if}\; op = \text{“}\mathcal{I}\text{”}\; \mathbf{then}\\
&\qquad\qquad\qquad\qquad 1 + \Sigma\mathcal{T}_{\mathcal{I}}\,(subtrees)\\
&\qquad\qquad\qquad \mathbf{else}\\
&\qquad\qquad\qquad\qquad \Sigma\mathcal{T}(subtrees)\\
&\mathcal{T}_{\mathcal{I}}\,\mathbf{graft}\, op\, subtrees \quad = \quad \mathbf{if}\; op = \text{“}\mathcal{I}\text{”}\; \mathbf{then}\\
&\qquad\qquad\qquad\qquad 1 + \Sigma\mathcal{T}_{\mathcal{I}}\,(subtrees)\\
&\qquad\qquad\qquad \mathbf{else\ if}\; op : var\\
&\qquad\qquad\qquad\qquad \mathcal{T}\,(\mathbf{parse}\,(\mathcal{I}\, op))\\
&\qquad\qquad\qquad \mathbf{else}\\
&\qquad\qquad\qquad\qquad \Sigma\mathcal{T}_{\mathcal{I}}(subtrees)
\end{aligned}
$$

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

$$Q = \text{“}\neg\mathcal{I}\, Q\text{”} \tag{63}$$

then we calculate

$$
\begin{aligned}
&\mathcal{T}\, Q\\
&= \mathcal{T}\ \text{“}\neg\mathcal{I}\, Q\text{”}\\
&= \mathcal{T}\ \text{“}\mathcal{I}\, Q\text{”}\\
&= \mathcal{T}_{\mathcal{I}}\ \text{“}Q\text{”}\\
&= 1 + \mathcal{T}\, Q
\end{aligned} \tag{64}
$$

and therefore $\mathcal{T}Q = \infty$ since $\mathcal{T}Q : xnat$. For any string that does not include the interpreter the time is linear in the size of the string; 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:

$$\forall s : lang \cdot \mathcal{T}\, s < \infty \;\Rightarrow\; \mathcal{I}\ \text{“}\mathcal{I}\text{”}; s = \mathcal{I}\, s \tag{65}$$

As an example of calculating with the interpreter, consider an expression normalizer $\mathcal{N}$ that linearizes an associative expression. For a sample input of "$a + ((b + c) + d)$" it would output "$a + (b + (c + d))$". For the language $l_{\mathcal{N}}$ of expressions containing variables, brackets and plus, $\mathcal{N}$ is a total function. A partial specification of its behaviour is

$$\forall s : l_{\mathcal{N}} \cdot \mathcal{I}\,(\mathcal{N}\, s) \;=\; \mathcal{I}\, s \tag{66}$$

We want to normalize the expression $(a + \mathcal{I}\, t) + (b + c)$. The sub-part t of the expression that is input to $\mathcal{N}$ might be unknown, such as if it came from a stream communicating with a different process. However, if we know that $t : l\mathcal{N}$, and that the variables in string t do not contain a, b, c, then it is reasonable to prove that $(a + \mathcal{I}\, t) + (b + c) \;=\; a + (b + (c + \mathcal{I}\, t))$ using the definition of the normalizer. The calculation would be

$$
\begin{aligned}
&(a + \mathcal{I}\, t) + (b + c)\\
&= \mathcal{I}\ \text{“}(a + \mathcal{I}\, t) + (b + c)\text{”}\\
&= \mathcal{I}\,(\mathcal{N}\ \text{“}(a + \mathcal{I}\, t) + (b + c)\text{”})\\
&= \mathcal{I}\ \text{“}a + (b + (c + \mathcal{I}\, t))\text{”}\\
&= a + (b + (c + \mathcal{I}\, t))
\end{aligned} \tag{67}
$$

Although the proof appears simple, a proof obligation required to justify the steps is that the interpretation time of $\mathcal{I}\, t$ must be finite. We argue that this example is representative of real computations that can be reasoned about using the interpreter.

The requirement to prove finite interpretation time in order to evaluate the interpretation of a string in the language is similar to the concepts of partial and total correctness proofs in program theories [3], [15]. One is a proof about the result of execution, and the other about the execution time. Many programming theories require finding either a bounded-decreasing function of the input to a program, or fixed-points for loops [21] [3]. Other functional and proof languages restrict the language itself, often constructively. In effect, that is excluding those strings from the language that cannot be built constructively. Instead of interpretation time, we can define constructively the interpretable strings which include the interpreter. Let those strings be called $ilang$, whose definition parallels the definition of $lang$, except that for variables:

$$\forall v : var \cdot \mathcal{I}\, v : ilang \Rightarrow v : ilang \tag{68}$$

And then we add that

$$\text{“}\mathcal{I}\text{”}; ilang \;:\; ilang \tag{69}$$

$$\text{“}\mathcal{I}\text{”}; ilang \;:\; lang \tag{70}$$

We prove that for all strings in $ilang$ the interpretation time is finite by induction. The base case is strings in $terminal$, for which $\mathcal{T}$ is zero. For strings in unary, binary or bracketing operators $\mathcal{T}$ is the sum of the interpretation time of the operands. For strings pre-fixed with the interpreter, $\mathcal{T}$ is one plus $\mathcal{T}_{\mathcal{I}}$ of the operand, which by the induction hypothesis is finite. Finally, only variables whose interpretation is in $ilang$ may be included, so for $v : var$ we have $\mathcal{T}_{\mathcal{I}}\, v \;=\; \mathcal{T}\,(\mathcal{I}\, v)$

which is finite by the inductive hypothesis, which concludes the proof ■.

Therefore, no expressive power was gained with a restricted language as opposed to demanding proof of finite interpretation time. We argue that it is preferable not to restrict the language to finite interpretation time, because reasoning about strings with infinite interpretation time can be of the same use as reasoning about infinite computations. The antecedent requiring finite interpretation time is sufficient for consistency. However, not all strings in the language whose interpretation time is infinite cause an inconsistency.

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.

IX. General Recursion in Functional Languages and Lazy Evaluation

The Trellys project [22][5] aims to create a functional language with general recursion and dependent types. The current approach is to separate the logical language from the computational language, because otherwise the result is inconsistency. The logical language and computational language share some parts, such as some shared data-types, but it is only the computational language which may have general unrestricted recursion. However, the interpreter can be applied to decouple timing from the language definition and allow the logic and computational language to be the same.

We claim that the interpreter can be used to simply and expressively reason about functional programming languages, and to define one with general recursion in a simple way. To do this, the interpretation of a functional program requires a finite execution time, and we decouple these two properties using the interpreter. Consider the following definition for the simple language of lambda calculus $\lambda lang$.

$$var : \lambda lang \tag{71}$$

$$\text{“(”}; \lambda lang; \text{“ ”}; \lambda lang; \text{“)”} \ : \ \lambda lang \tag{72}$$

$$\text{“}\lambda\text{”}; var; \text{“}\cdot\text{”}; \lambda lang \ : \ \lambda lang \tag{73}$$

Although consistent within its own domain, lambda calculus is not consistent when combined with other theories such as boolean algebra and arithmetic. However, even without requiring typed lambda terms, there are programming languages such as Python that implement consistently lambda terms within the programming language. This suggests that by restricting the evaluation of β-reductions to only those with finite execution time can resolve the inconsistency. For $v : var$ and $s, t : \lambda lang$ we define interpretation for lambda calculus as follows:

$$\mathcal{I}_\lambda \text{“}\lambda v \cdot\text{”}; s \ = \ \lambda v \cdot \mathcal{I}_\lambda \, s \tag{74}$$

$$\mathcal{I}_\lambda \text{“(”}; s; \text{“)”} \ = \ \mathcal{I}_\lambda \, s \tag{75}$$

$$\mathcal{T}_\lambda \, (s; \text{“ ”}; t) < \infty \ \Rightarrow \quad \mathcal{I}_\lambda \, s; \text{“ ”}; t \ = \ (\mathcal{I}_\lambda \, s) \, (\mathcal{I}_\lambda \, t) \tag{76}$$

In the same way as before, variables are abbreviation of the interpretation of strings in var, so $\mathcal{I}_\lambda$ “a” is an abbreviation of a. We assume the existence of a function β that performs β-reductions on strings in $\lambda lang$. For $v : var$ and $s, t : \lambda lang$ the timing function for lambda calculus is:

$$\mathcal{T}_\lambda \, v \ = \ 0 \tag{77}$$

$$\mathcal{T}_\lambda \text{“}\lambda\text{”}; v; \text{“}\cdot\text{”}; s \ = \ \mathcal{T}_\lambda \, s \tag{78}$$

$$\mathcal{T}_\lambda \text{“(”}; s; \text{“ ”}; t; \text{“)”} \ = \ 1 + \mathcal{T}_\lambda \, (\beta \, s \, t) \tag{79}$$

The proof of inconsistency for untyped lambda calculus shows that the β-reduction of $(\lambda x \cdot \neg(x \ x)) \ (\lambda x \cdot \neg(x \ x))$ is equal to its negation. However, the definition of $\mathcal{I}_\lambda$ requires finite interpretation time for β-reduction, which is not the case for this term:

$$\mathcal{T}_\lambda \text{“}((\lambda x \cdot \neg(x \ x)) \ (\lambda x \cdot \neg(x \ x)))\text{”} \tag{80}$$

$$= 1 + \mathcal{T}_\lambda \, \beta \, (\text{“}\lambda x \cdot \neg(x \ x)\text{”} \ \text{“}\lambda x \cdot \neg(x \ x)\text{”}) \tag{81}$$

$$= 1 + \mathcal{T}_\lambda \text{“}\neg((\lambda x \cdot \neg(x \ x)) \ (\lambda x \cdot \neg(x \ x)))\text{”} \tag{82}$$

$$= 1 + \mathcal{T}_\lambda \text{“}(\lambda x \cdot \neg(x \ x)) \ (\lambda x \cdot \neg(x \ x))\text{”} \tag{83}$$

And therefore the interpretation time is equal to ∞ as before. We have decoupled interpretation from execution, and made the theory consistent. The interpretation time was mapped to the underlying computation model, albeit slightly abstracted: β-reduction costs time 1, and all else is free.

In general, for any logic whose inconsistency is due purely to operators that take infinite time to compute, it can be used as a consistent computation logic with no change. In essence, the computation logic of a theory makes operators incomplete for infinite executions. For this reason lambda calculus can be used as a computation logic with no change.

Furthermore, interpretation time allows a flexible timing policy. If instead the computation model is an Oracle Turing Machine, then the cost of β-reduction becomes finite even for infinite computations, and the theory becomes an inconsistent computation logic. The timing function can be further generalized to consider valuation of inputs. This is particularly useful for reasoning about lazy evaluation of functional programs, because a lazy language can use the exact same logic as its non-lazy counter-part with the only difference being their interpretation time. A lazy language can allow for infinite data-types such as infinite lists, or inductively defined infinite data-types. Consider the following definition of an infinite list of naturals:

$$natlist \ = \ 0; (1 + natlist) \tag{84}$$

which can be equivalently defined without explicit recursion using a Y-combinator as

$$natlist = (\lambda f \cdot (f \ f)) \ (\lambda s \cdot 0; (1 + (s \, s))) \tag{85}$$

and that we wish to evaluate $natlist_{42}$. As currently defined, $\mathcal{T}_\lambda$ is a lazy timing policy, and $\mathcal{T}_\lambda \, (natlist_{42}) \ = \ 42$. An eager timing policy results in infinite execution time, and is achieved by the simple change to the timing function:

$$\mathcal{T}_\lambda \text{“(”}; s; \text{“ ”}; t; \text{“)”} \ = \ 1 + (\beta \, (\mathcal{T}_\lambda \, s) \, (\mathcal{T}_\lambda \, t)) \tag{86}$$

X. Proof of Consistency

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

$$\text{“0”} = 0,\ \ldots \text{“9”} = 9, \text{“}a\text{”} = 10,\ \ldots \text{“}z\text{”} = 25,\ \ldots \quad (87)$$

Strings are implemented as ordered pairs in the standard way.

$$a; b = \{\{a\}, \{a, b\}\} \quad (88)$$

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 [9]. 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

$$F\,0 = a \quad (89)$$

$$\forall n : nat \cdot F\,(n+1) = f(F\,n) \quad (90)$$

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

$$(s > t) = strNum(s) > strNum(t) \quad (91)$$

$$strNum = \langle S : {*char} \to \textbf{if}\ S = nil\ \textbf{then}\ 0 \quad (92)$$
$$\textbf{else}\ S_0 + charNum \times S_{1,..\leftrightarrow S}\ \textbf{fi}\rangle$$

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

$$enum = (g^{-1}) \quad (93)$$

$$g = \langle n : nat \to \textbf{if}\ n = 0\,\textbf{then}\ (MIN\,s : lang \cdot s) \quad (94)$$
$$\textbf{else}\,(MIN\,s : (\S t : lang \cdot t > g(n-1)) \cdot s)\,\textbf{fi}\rangle$$

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

$$F\,0 = \{0; 0\} \quad (95)$$
$$\vdots$$
$$F\,9 = \{9; 9\} \cup F\,8$$

For all $s : char$ let $m = enum(\text{“}\backslash\text{“”}; s; \text{“}\backslash\text{””})$ in

$$F\,m = \{m; s\} \cup F\,(m-1) \quad (96)$$

$$F\,(n+1) = \{(n+1); (H\,(n+1)\,(F\,n))\} \cup F\,n \quad (97)$$

(H is defined below)

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.

$$H\,k\,I = \quad (98)$$
$$\{S : Sets | \exists n, m : \textbf{dom}(I) \cdot g\,k = (g\,n); \text{“}+\text{”}; (g\,m)$$
$$\wedge\, S = I\,n + I\,m\} \cup$$
$$\{S : Sets | \exists n, m : \textbf{dom}(I) \cdot g\,k = (g\,n); \text{“}\wedge\text{”}; (g\,m)$$
$$\wedge\, S = I\,n \wedge I\,m\} \cup$$
$$\{S : Sets | \exists n : \textbf{dom}(I) \cdot g\,k = \text{“}\neg\text{”}; (g\,n) \wedge S = \neg I\,n\} \cup$$
$$\vdots$$

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.

$$\mathcal{I} = \langle s : lang \to (F\ (enum\,s))\,(enum\,s)\rangle \quad (99)$$

XI. Conclusion

We have presented the formalism of an expression interpreter for the purpose of encoding meta-logic within a logic. This allows effective reasoning with partial terms and about functional languages. 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 reasonable, since the parsing that is required for its use is an efficient linear-time algorithm. We show how the interpreter can be used to implement a logic for functional programming that allows general recursion while maintaining consistency.

The encoding permits a flexible timing policy of execution, which also allows reasoning about lazy evaluation.

REFERENCES

[1] L. Naiman, "Using an expression interpreter to reason with partial terms," in *COMPUTATION TOOLS 2013: the fourth international conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking*, ser. IARIA '11, 2011, pp. 37–43.

[2] The Coq development team. (2004) The coq proof assistant reference manual. LogiCal Project. Version 8.0. [Online]. Available: http://coq.inria.fr. Access: 2013-01-20.

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

[4] C. B. Jones, *Systematic Software Development Using VDM (2Nd Ed.)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1990.

[5] G. Kimmell, A. Stump, H. D. Eades, III, P. Fu, T. Sheard, S. Weirich, C. Casinghino, V. Sjöberg, N. Collins, and K. Y. Ahn, "Equational reasoning about programs with general recursion and call-by-value semantics," in *Proceedings of the Sixth Workshop on Programming Languages Meets Program Verification*, ser. PLPV '12. New York, NY, USA: ACM, 2012, pp. 15–26.

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

[7] 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.

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

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

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

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

[12] 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.

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

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

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

[16] J. Woodcock and V. Bandur, "Unifying theories of undefinedness in utp," in *UTP*, ser. LNCS, B. Wolff, M.-C. Gaudel, and A. Feliachi, Eds., vol. 7681. Springer, 2012, pp. 1–22.

[17] E. C. R. Hehner, *A Practical Theory of Programming*. New York: Springer, 1993. [Online]. Available: http://www.cs.toronto.edu/ hehner/aPToP/. Access: 2014-05-27

[18] ——, "Beautifying gödel," pp. 163–172, 1990.

[19] 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.

[20] 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.

[21] C. A. R. Hoare and J. He, *Unifying Theories of Programming*. New York: Prentice Hall, 1998.

[22] V. Sjöberg, C. Casinghino, K. Y. Ahn, N. Collins, H. D. E. III, P. Fu, G. Kimmell, T. Sheard, A. Stump, and S. Weirich, "Irrelevance, heterogeneous equality, and call-by-value dependent type systems," in *MSFP*, 2012, pp. 112–162.