Recursion and Induction in Ontic

David McAllester

MIT Artificial Intelligence Laboratory
545 Technology Square
Cambridge Mass. 02139

phone: (617)253-6599
email: dam@ai.mit.edu
paper category: research

Abstract: This paper describes Ontic, a higher order formal specification language designed for expressing formal concepts as clearly and concisely as possible. The consiceness and clarity is achieved through the use of nondeterminism. In a nondeterministic language each term has a set of possible values. This improves conciseness in a variety of ways. First, since terms denotes sets, nondeterminism allows terms to be used as types. Second, since each term has a set of possible values it is natural for a term not to have any values. This provides a natural representation of partial functions. Third, nondeterminism allows a single recursive definition and induction mechanism to be used for both data type definition and recursive function definition. Fourth, nondeterminism provides a clean integration of programming languages and set theory. Ontic has the expressive power of ZFC set theory. This paper presents the syntax and semantics of Ontic together with a large collection of examples of Ontic definitions. The examples are essential in demonstrating the conciseness and clarity of Ontic definitions. This paper contains only the basic inference rules for induction for reasoning about recursive definitions. A later version of this paper will include a more complete description of induction mechanisms.

This research was supported by the Advanced Research Projects Agency of the Department of Defense contracts N00014-89-j-3202 and F33615-91-C-1788.

1 Introduction

This paper describes a descendent of the Ontic language described in [7]. This earlier system was used to verify the Stone representation theorem from the foundations of set theory. The main innovation of the language described here is the introduction of nondeterminism and the resulting elimination of the syntactic distinction between terms and types. Although a verification system has been constructed for Ontic, this paper is primarily about the syntax and semantics of the language and contains a large set of examples intended to demonstrate the conciseness and clarity resulting from the declarative use of nondeterminism. This paper presents both a formal denotational semantics and a formal inference system, i.e., a set of inference rules, for the Ontic language.

The inference rules presented here can be used as the foundation for a mechanical verification system. However, any practical verification system must be highly automated and the inference rules alone do not provide a means of achieving this automation. Although this paper addresses the clarity and conciseness achieved through nondeterminism, nondeterminism can also be motivated as a means of achieving greater automation. Nondeterminism can be viewed as an alternative syntax for first order terms and formulas — the so called taxonomic syntax for first order logic [6]. Taxonomic syntax has significant advantages for automated reasoning — it allows a larger fragment of the reasoning process to be handled by efficient type inference mechanisms. The use of taxonomic syntax is part of a larger program of basing inference procedures on nonstandard syntactic constructions. A syntax based on natural language under Montague semantics seems to be particularly effective [5], [4], [2]. The Montagovian syntax for first order logic is not discussed here.

Ontic is best viewed as an extension of pure Lisp. Nondeterminism is introduced through the use of McCarthy's `amb` operator [8], which is renamed in Ontic to `either`. From an operational viewpoint, when evaluating an expression of the form `(either` u v`)` the evaluator nondeterministically selects either the first or second argument and returns a value of the selected argument. In this language each term has a set of possible values. It is easy to construct a recursive definition of an expression `(an-integer)` such that any integer is a possible value of this expression. Because each term is naturally associated with a set, terms can be used as types. Ontic is strongly typed in the sense that each bound variable is associated with a term, such as `(an-integer)`, specifying a range of allowed values.

In addition to providing the foundation of certain inference algorithms, nondeterminism provides greater linguistic elegance. Collapsing terms and types eliminates redundancy that is otherwise present between term and type expressions. For example, consider the lisp function `cons`. The expression `(cons 'a 'b)` is a traditional term while the expression `(cons (a-symbol) (a-symbol))` is a natural type. Using nondeterminism to collapse terms and types eliminates the need for separate function and type constructors for `cons`.

In addition to nondeterminism, the language defined here extends Lisp with a few additional primitives giving Ontic the power of higher order quantification. In Ontic the type `(a-subset-of (the-set-of-all (an-integer)))` expresses the type of arbitrary subsets of the integers. One can also express the type of all functions from integers to integers. The mechanisms for higher quantification combined with a primitive universal type give Ontic the expressive power of Zermelo Fraenkel set theory with the axiom of choice (ZFC).

From the basic primitives of Ontic one can quickly define sophisticated mathematical concepts. Section 6 contains a complete foundational construction of the real numbers. Fractions are defined as pairs of integers, rationals as equivalence classes of fractions, and the reals as Dedekind cuts in the rationals.

The denotational semantics and inference system presented in sections 10 and 11 should allow other researchers to build verification systems for the Ontic language. It would be particularly interesting to construct a verification system for this language using a metasystem such as Isabelle [10]. Ideally any verification system for Ontic would take advantage of inference algorithms for taxonomic syntax such as those described in described in [6].

2 Basic Ontic

The Ontic language is presented here in stages. First, this section describes a strongly typed nondeterministic Lisp. Later sections extend the language with additional primitives providing all the expressive power of Zermelo Fraenkel set theory with the axiom of choice (ZFC). The language is introduced here through examples. A formal presentation of syntax and denotational semantics is given in section 10.

The terms considered in this section are constructed from variables, quoted symbols, `cons`, `car`, `cdr`, `either`, `if`, `lambda`, application, and recursively defined operators. `cons` is the fundamental pairing operation operation of Lisp and `car` and `cdr` are the functions for taking the first and second component of a pair. A quoted symbol always has exactly one possible value — the symbol itself. The term `(either 'foo 'bar)` has two possible values — the symbol `'foo` and the symbol `'bar`. The term

```
(cons (either 'foo 'bar) (either 'foo 'bar))
```

has four different possible values.

Ontic allows recursive definitions of operators. Operators of no arguments are called "thunks". In Ontic thunks are used to represent types. As an example we use recursion to define the concept of an expression and the concept of a list of expressions.

```
(define (an-expression)
  (either (a-symbol)
          (cons (an-expression) (an-expression))))

(define (an-exp-list)
  (either 'nil
          (cons (an-expression) (an-exp-list))))

(define (a-symbol-alist)
  (either 'nil
          (cons (cons (a-symbol) (an-expression)) (a-symbol-alist))))
```

The above definitions can be made more elegant once we have introduced higher order types in sections 5 and 9. We can also define the natural numbers.

```
(define (zero)
  (list 'the-natnum 'zero))

(define (a-natnum)
  (either (zero) (list 'the-natnum 'successor (a-natnum))))

(define (succ (x (a-natnum)))
  (list 'the-natnum 'successor x))
```

As usual, a term of the form `(list` u_1 u_2 ... u_n`)` can be treated as an abbreviation for `(cons` u_1 `(cons` u_2 ... `(cons` u_n `'nil)` ...`))`. Note that in the definition of `succ` the argument `x` is assigned the "type" `(a-natnum)`. In Ontic all bound variables are assigned types. The expression `(let ((x (a-foo)))` u`)` is taken to be an abbreviation for `((lambda ((x (a-foo)))` u`) (a-foo))`.

Variables always have exactly one possible value — the value of that variable in the current environment. Since a variable can only have one possible value, the expression

```
(let ((x (either 'foo 'bar)))
  (cons x x))
```

has only two possible values. Note that if we replace `x` by `(either 'foo 'bar)` in the body `(cons x x)` we get `(cons (either 'foo 'bar) (either 'foo 'bar))` which has more possible values. The term

```
(let ((x (lambda () (either 'foo 'bar))))
  (cons (x) (x)))
```

has four possible values.

We now define an expression that has no possible values. In Ontic there is no distinction between divergent computation and failure to have a value.

```
(define (fail)
```

```
(fail))
```

Ontic has a separate syntactic category of formulas. Formulas are used as tests in `if` expressions. All Ontic formulas can be constructed from the primitive formula constructor `is` and Boolean combinations. The formula (`is` u w) is true if every possible value of u is also a possible value of w. If u has only one possible value then the formula (`is` u v) expresses the statement that u has type v. If u has more than one possible value then (`is` u v) expresses the statement that u is a subtype of v. If u has no possible values then (`is` u v) is true. If both u and v have exactly one possible value then (`is` u v) expresses the statment that u and v have the same value, i.e., that they are equal. Note that each formula is either true or false, and never both, even though formulas can contain terms with many possible values.

Ontic includes considerable "syntactic sugar" for concisely writing formulas which are semantically equivalent to expressions constructed from `is` and Boolean combinations.[1] The formula (`there-exists` u) can be taken as an abbreviation for (`not (is` u `(fail)))`. The "generate and test" term (`some` x `(a-foo) such-that` $\Phi[x]$) can be taken as an abbreviation for (`let` ((x `(a-foo)))` (`when` $\Phi[x]$ x)) where (`when` Φ u) is an abbreviation for (`if` Φ u `(fail))`. Formulas of the form `(forall ((x (a-foo)))` $\Phi[x]$) can be taken as an abbreviations for `(is (a-foo) (some x (a-foo) such-that` $\Phi[x]$`))`. Formulas of the form `(exists ((x (a-foo)))` $\Phi[x]$) can either be constructed from universal quantification and negation in the standard way or treated as abbreviations for `(there-exists (some x (a-foo) such-that` $\Phi[x]$`))`. The formula `(at-most-one (a-foo))` can be taken as an abbreviation for `(forall ((x (a-foo)) (y (a-foo))) (is x y))`. The formula `(singleton (a-foo))` can be taken as an abbreviation for `(and (there-exists (a-foo)) (at-most-one (a-foo)))`.

We can define the predecessor function and the addition function on natural numbers as follows.

```
(define (pred (x (a-natnum)))
```

[1] In practice we have found it important to implement syntactic sugar directly using additional primitives rather than translate it into the small number of primitives listed in the previous section. Additional semantically redundant primitives allow inference rules to operate directly on syntactic sugar resulting in greatly simplified formal proofs.

```
  (some y (a-natnum) such-that (is x (succ y))))

(define (sum (x (a-natnum)) (y (a-natnum)))
  (if (is x (zero)) y (succ (sum (pred x) y))))

(define (prod (x (a-natnum)) (y (a-natnum)))
  (if (is x (zero)) (zero) (sum y (prod (pred x) y))))
```

In Ontic predicates are represented by nondeterministic operators. The following operator takes a number and nondeterministically returns a larger number. The statement that x is greater then y can be expressed in Ontic as `(is` x `(greater-than` y`))`.

```
(define (greater-than (x (a-natnum)))
  (either (succ x)
          (greater-than (succ x))))

(define (between (x (a-natnum)) (y (greater-than x)))
  (both (greater-than x) (less-than y)))
```

Notice the use of dependent types in the definition of `between` — the type of `y` depends on the value of `x`. In general multiple argument operators are treated by Currying. This allows a natural use of dependent types. The definition of `between` also involves a term of the form `(both (a-foo) (a-bar))`. Terms of this form can be treated as syntactic sugar for `(some x (a-foo) such-that (is x (a-bar)))`.

3 Recursion

Ontic allows set theoretically monotone and continuous recursive definitions (STMC). Each recursive definition has a least fixed point which is taken to be the meaning of the recursively defined symbol. The phrase "set theoretic" means that the ordering used in defining this least fixed point is simple set theoretic inclusion on relations. Set theoretic recursion is used in the μ-calculus [9], [1]. This should be contrasted with least fixed points in complete partial orders (CPOs) as used in the semantics of many higher order programming languages. The fixed point induction rule for STMC definitions (given in section 11) is simpler than the corresponding fixed point induction rule for CPO least fixed points [3]. All recursive Ontic definitions have the following form.

```
(define (foo (X_1 τ_1) ... (X_n τ_n))
  B[foo X_1 ... X_n])
```

Such definitions must be syntactically STMC where we define this to mean that it satisfies the following five conditions.

1. `foo` does not occur in any type restrictions τ_i.
2. All occurrences of `foo` in the body must be of the form `(foo` u_1 ... u_n`)` where n is the given number of parameters of `foo`.
3. `foo` does not occur inside a formula.
4. `foo` does not occur inside the primitives `the-set-of-all`, `choice` or Y.
5. `foo` does not occur inside `lambda` expressions other than those on the left hand sides of applications.

It seems that one would rarely, if ever, construct recursive definitions of natural operators that violate restriction 1. Restriction 1 is not really required — definitions that violate 1 can still be STMC — but restriction 1 simplifies the processing of recursive definitions. Restriction 2 is more significant — it prevents the defined symbol from being passed as a higher order parameter to mapping or iteration functions. A more liberal, but more complex, definition of syntactically STMC can be constructed which allows passing the defined symbol to mapping functions. We will stick to the simpler restrictions given above. Restriction 3 is rarely a problem in practice. Many definitions ruled out by condition 3 can be converted to equivalent acceptable definitions by using `let` expressions to move recursions out of formulas. Unfortunately, this can not always be done. An example of problematic recursion inside a formula is given below. Restriction 4 can be ignored in basic Ontic because the primitives mentioned there do not arise. The primitive `the-set-of-all` is described in section 5. The constructors `choice` and Y are described in section 10. Restriction 5 can be ignored for first order definitions, such as those of the Boyer-Moore logic, because those definitions do not involve `lambda` expressions. `lambda` expressions are implicit in `let` expressions but these implicit `lambda` expressions are always the left hand side of applications — restriction 5 can be ignored for `let` expressions also.

To understand the semantic significance of the above restriction we need to define the notion of a semantically STMC context. Let C[`foo`] be an expression containing one or more occurrences of `foo`. An Ontic operator f of n arguments can be associated with an $n+1$-ary relation where the tuple $<x_1, \ldots, x_n, y>$ is a member of the relation associated with f if y is a possible value of f applied to $<x_1, \ldots, x_n>$. Let $V[C[f], \rho]$ be the set of possible values of the term C[`foo`] when `foo` is interpreted as f and free variables other than `foo` are interpreted according to variable interpretation ρ.[2] Let $f \subseteq g$ signify that the relation associated with the operator f is a subset of the relation associated with the operator g. A context C[`foo`] is *set theoretically monotone* if for any operators f and g such that $f \subseteq g$ and any variable interpretation ρ we have that $V[C[f], \rho] \subseteq V[C[g], \rho]$. A context C[`foo`] is *set theoretically continuous* if for any infinite sequence of operators $f_1 \subseteq f_2 \subseteq f_3 \subseteq \ldots$ and any variable interpretation ρ we have $V[C[\cup_i f_i], \rho] = \cup_i V[C[f_i], \rho]$. The context C[`foo`] is STMC if it is both set theoretically monotone and set theoretically continuous. The above syntactic restrictions on recursive definitions ensure that the body of each definition is an STMC context for the defined symbol. This ensures the existence of a unique least set theoretic fixed point for the recursive definition.

The definition of syntactically STMC definitions can be liberalized considerably — a larger class of definitions can be syntactically recognized as STMC. The main problem with more liberal definitions is the increased complexity of the statement of the syntactic condition. For example, recursion can be allowed inside formulas provided that the recursion occurs positively and that no recursion occurs in the negative branch of the enclosing conditional. Or the recursion can occur negatively provided that no recursion occurs in the positive branch. One of the main objections to formal verification languages is that they can only be used by expert users. It seems that

[2] A formal definition of the denotational semantics of Ontic is presented in section 10.

the above restrictions, while more restrictive than necessary, strike a reasonable balance between allowing a large class of definitions and providing a simply stated syntactic restriction.

It is interesting to note some examples of illegal recursions. For example, consider the following.

```
(define (a-paradoxical-thing)
  (if (there-exists (a-paradoxical-thing))
      (fail)
      'the-thing))
```

If this definition were accepted one could prove that `(a-paradoxical-thing)` has a value if and only if it does not have a value. This definition is ruled out because the recursion occurs inside a formula. This definition is not monotone. An example of a monotone definition which is ruled out due to a failure of continuity is given at the end of section 8.

4 Some Examples

As another example we define some simple data types for symbolic computation.

```
(define (a-variable)
  (list 'the-variable (a-symbol)))

(define (a-constant)
  (list 'the-constant (a-natnum)))

(define (a-num-expression)
  (either (a-constant)
          (a-variable)
          (list 'sum (a-num-expression) (a-num-expression))
          (list 'prod (a-num-expression) (a-num-expression))))
```

We can also define the "semantics" of the data objects defined above. The following definitions involve terms of the form

$$\texttt{(cond ((}\Phi_1\ u_1\texttt{) (}\Phi_2\ u_2\texttt{)}\ \ldots\ \texttt{(}\Phi_n\ u_n\texttt{))).}$$

This is an abbreviation for

$$\texttt{(if }\Phi_1\ u_1\ \texttt{(if }\Phi_2\ u_2\ \texttt{(if}\ \ldots\ \texttt{(if }\Phi_n\ u_n\ \texttt{(fail))}\ \ldots\texttt{))).}$$

The formula `(true)` can be taken to be an abbreviation for `(is 'a 'a)`.

```
(define (an-environment)
  (either 'nil
          (cons (cons (a-variable) (a-natnum))
                (an-environment))))

(define (env-value (v (a-variable)) (env (an-environment)))
  (cond ((is env 'nil) (zero))
        ((is (car (car env)) v)
         (cdr (car env)))
        ((true) (env-value v (cdr env)))))

(define (num-value (exp (a-num-expression)) (env (an-environment)))
  (cond ((is exp (a-variable)) (env-value exp env))
        ((is exp (a-constant)) (cdr exp))
        ((is (car exp) 'sum)
         (sum (num-value (car (cdr exp)) env)
              (num-value (car (cdr (cdr exp))) env)))
        ((is (car exp) 'prod)
         (prod (num-value (car (cdr exp)) env)
               (num-value (car (cdr (cdr exp))) env)))))
```

Finally, we can express the concept of a valid Diophantine equation.

```
(define (a-di-equation)
  (list '= (a-num-expression) (a-num-expression)))

(define (a-valid-di-equation)
  (some e (a-di-equation) such-that
    (forall ((env (an-environment)))
      (is (num-value (car (cdr e)) env)
          (num-value (car (cdr (cdr e))) env)))))
```

5 Predicative Set Theory

In this section we consider three additional primitives that provide a natural way of talking about sets. In particular we have the three new primitives `the-set-of-all`, `a-member-of` and `a-subset-of`. A term of the form `(the-set-of-all` u`)` has exactly one possible value which is the set containing all of the possible values of u. For example `(the-set-of-all (a-natnum))` is the set of all natural numbers. It is possible to show that any term constructed from the primitives given in section 2 has a countable set of possible values. However, the primitive `a-subset-of` allows one to express uncountable sets, such as the family of all subsets of the natural numbers.

```
(define (an-expression-set)
  (a-subset-of (the-set-of-all (an-expression))))

(define (the-empty-set)
  (the-set-of-all (fail)))

(define (a-finite-expression-set)
  (either (the-empty-set)
          (let ((s (a-finite-expression-set))
                (exp (an-expression)))
            (the-set-of-all (either exp (a-member-of s))))))
```

6 The Real Numbers

We now give a definition of the real numbers as an example of foundational constructions in Ontic. This is done by defining fractions as pairs of natural numbers, rationals as equivalence classes of fractions, and real numbers as downward closed sets of rationals (Dedekind cuts). Although the presentation given here is foundational, it is also possible to take an axiomatic approach in Ontic. This would be done by defining a real closed field in a manner similar to the definition of a group given in section 9. Some foundational construction, such as the one given in this section, is still required if one wants to prove that real closed fields exist.

We start the foundational construction with the definitions of fractions. We define a fraction to be a pair of natural numbers.

```
(define (a-nonzero-natnum)
  (succ (a-natnum)))

(define (a-fraction)
  (list 'the-fraction
        (a-natnum)
        (a-nonzero-natnum)))

(define (numerator (f (a-fraction)))
  (car (cdr f)))

(define (denominator (f (a-fraction)))
  (car (cdr (cdr f))))

(define (make-fraction (n (a-natnum)) (d (a-nonzero-natnum)))
  (list 'the-fraction n d))

(define (frac-prod (f1 (a-fraction)) (f2 (a-fraction)))
  (make-fraction (prod (numerator f1) (numerator f2))
                 (prod (denominator f1) (denominator f2))))

(define (frac-sum (f1 (a-fraction)) (f2 (a-fraction)))
  (make-fraction (sum (prod (numerator f1) (denominator f2))
                      (prod (numerator f2) (denominator f1)))
                 (prod (denominator f1) (denominator f2))))

(define (frac-less-than (f (a-fraction)))
  (some g (a-fraction) such-that
    (is (prod (numerator f) (denominator g))
        (greater-than (prod (numerator g) (denominator f))))))
```

Next we define equivalence for fractions and define the rationals to be equivalence classes of fractions.

```
(define (an-equivalent-fraction (f1 (a-fraction)))
  (some-such-that f2 (a-fraction)
    (= (prod (numerator f1) (denominator f2))
       (prod (numerator f2) (denominator f1)))))

(define (the-rat-rep-by (f (a-fraction)))
  (the-set-of-all (an-equivalent-fraction f)))

(define (a-rational)
  (the-rat-rep-by (a-fraction)))
```

```
(define (frac-representing (r (a-rational)))
  (a-member-of r))

(define (rat-less-than (r (a-rational)))
  (the-rat-rep-by (frac-less-than (frac-representing r))))
```

Now we define cuts. These are downward closed subsets of rationals that do not contain all rationals and that do not contain greatest elements.

```
(define (a-cut)
  (some c (a-subset-of (the-set-of-all (a-rational))) such-that
    (and (is (rat-less-than (a-member-of c)) (a-member-of c))
         (exists ((r (a-rational))) (not (is r (a-member-of c))))
         (forall ((r1 (a-member-of c)))
           (exists ((r2 (a-member-of c)))
             (is r1 (rat-less-than r2)))))))

(define (cut-prod (c1 (a-cut)) (c2 (a-cut)))
  (the-set-of-all
     (rat-rep-by
      (frac-prod (frac-representing (a-member-of c1))
                 (frac-representing (a-member-of c2))))))

(define (cut-sum (c1 (a-cut)) (c2 (a-cut)))
  (the-set-of-all
    (rat-rep-by
      (frac-sum (frac-representing (a-member-of c1))
                (frac-representing (a-member-of c2))))))

(define (cut-difference (c1 (a-cut)) (c2 (a-cut)))
  (some c3 (a-cut) such-that (is c1 (cut-sum c2 c3))))

(define (cut-less-or-equal (c (a-cut)))
  (both (a-cut) (a-subset-of c)))

(define (cut-greater-or-equal (c (a-cut)))
  (some c2 (a-cut) such-that (is c (cut-less-or-equal c2))))
```

The above definitions only deal with nonnegative fractions, rationals, and cuts. We now construct the reals as pairs of a sign and a cut.

```
(define (a-real-rep)
  (list (either 'plus 'minus) (a-cut)))

(define (sign-part (r (a-real-rep)))
```

```
  (car r))

(define (cut-part (r (a-real-rep)))
  (car (cdr r)))

(define (fix-zero (r (a-real-rep)))
  (if (is (cut-part r) (the-empty-set))
      (list 'plus (the-empty-set))
      r))

(define (a-real)
  (fix-zero (a-real-rep)))

(define (sign-prod (s1 (either 'plus 'minus)) (s2 (either 'plus 'minus)))
  (cond ((is s1 'plus) s2)
        ((is s2 'plus) s1)
        ((true) 'plus)))

(define (real-prod (x (a-real)) (y (a-real)))
  (fix-zero
   (list (sign-prod (sign-part x) (sign-part y))
         (cut-prod (cut-part x) (cut-part y)))))

(define (real-sum (x (a-real)) (y (a-real)))
  (cond ((is (sign-part x) (sign-part y))
         (list (sign-part x) (cut-sum (cut-part x) (cut-part y))))
        ((is (sign-part x) 'plus)
         (if (is (cut-part x) (cut-less-or-equal (cut-part y)))
             (fix-zero
              (list 'minus (cut-difference (cut-part y) (cut-part x))))
             (list 'plus (cut-difference (cut-part x) (cut-part y)))))
        ((true) (real-sum y x))))

(define (real-less-or-equal (x (a-real)))
  (if (is (sign-part x) 'plus)
      (either (fix-zero (list 'minus (a-cut)))
              (list 'plus (a-cut-less-or-equal (cut-part x))))
      (list 'minus (a-cut-greater-or-equal (cut-part x)))))
```

7 Operator Types

The primitives `a-subset-of` introduced in the previous section can be viewed as introducing higher order quantification — it allows quantification over all subsets of the natural numbers. In this section we describe two more primitives for higher order quantification — `an-operator-from-to` and `a-thunk-to`. These primitives allow one to quantify over functions from numbers to numbers and to quantify over all thunks representing types of numbers. If s and w each deterministically denote sets, i.e., they each have one possible value which is a set, then the possible values of a term of the form (`an-operator-from-to` s w) consist of all the operators f whose domain is s and such that all possible values of f applied to a member of s is a member of w. Similarly, the possible values of the term (`a-thunk-to` w) consist of all thunks whose output sets are subsets of w. If s and w have more than one possible value then the possible values of (`an-operator-from-to` s w) are all possible values of (`an-operator-from-to` x y) where x is a possible value of s and y is a possible value of w. A similar statement holds for `a-thunk-to`. Consider the following definitions. Keep in mind that the natural numbers as defined above are also expressions.

```
(define (an-expression-operator)
  (an-operator-from-to (an-expression-set) (the-set-of-all (an-expression))))

(define (an-exp-thunk)
  (a-thunk-to (the-set-of-all (an-expression))))

(define (an-exp-list-of (t (an-exp-thunk)))
  (either 'nil
          (cons (t) (an-exp-list-of t))))

(define (a-natnum-list)
  (an-exp-list-of a-natnum))

(define (a-natnum-operator)
  (an-operator-from-to (the-set-of-all (a-natnum)) (the-set-of-all (a-natnum))))

(define (exp-map (t (an-exp-thunk))
 (f (an-operator-from-to (the-set-of-all (t))
 (the-set-of-all (t))))
 (l (an-exp-list-of t)))
  (if (is l 'nil)
      'nil
      (cons (f (car l)) (exp-map f (cdr l)))))
```

8 Predicative Universes

STMC recursion can be used to construct quite large sets. However, STMC recursion can not be used to construct the set of all sets. Continuous recursion has the property that the least fixed point can be expressed as a union of the (countably many) finite approximations. Using monotone and continuous recursion we can construct "universes".

```
(define (a-first-universe)
  (either (the-set-of-all (an-expression))
          (let ((u (a-first-universe)))
            (the-set-of-all
              (either (a-subset-of u)
                      (cons (a-member-of u) (a-member-of u))
                      (an-operator-from-to (a-subset-of u) u)
                      (a-thunk-to u))))))

(define (a-first-thing)
  (a-member-of (a-first-universe)))
```

The recursive definition of `(a-first-universe)` is syntactically STMC. The expression `(a-first-universe)` has countably many different values each of which is "a universe". The type `(a-first-thing)` is large enough to contain natural representations of all the objects of ordinary mathematics. However, it is not large enough to contain as a member the very large universe `(the-set-of-all (a-first-thing))`. To construct a universe containing this larger object we can construct yet another sequence of universes.

```
(define (a-second-universe)
  (either (the-set-of-all (a-first-thing))
          (let ((u (a-second-universe)))
            (the-set-of-all
              (either (a-subset-of u)
                      (cons (a-member-of u) (a-member-of u))
                      (an-operator-from-to (a-subset-of u) u)
                      (a-thunk-to u))))))

(define (a-second-thing)
  (a-member-of (a-second-universe)))
```

Of course the larger universe `(the-set-of-all (a-second-thing))` is not itself a possible value of `(a-second-thing)` and even larger universes can be defined.[3] Intuitively, we would like to be able to define one truly universal type that contains *everything*. Unfortunately no definable universe can contain itself as a member. More generally, the principles of ZFC set theory prohibit any set, definable or not, from containing itself. So there is no universe of everything.

One can construct recursive definitions which appear to define the entire universe. For example consider the following definition of `(a-thing)`.

```
(define (a-thing)
  (either (a-symbol)
          (a-subset-of (the-set-of-all (a-thing)))
          (cons (a-thing) (a-thing))
          (an-operator-from-to (a-subset-of (the-set-of-all (a-thing)))
                               (the-set-of-all (a-thing)))
          (a-thunk-to (the-set-of-all (a-thing)))))
```

This definition is monotone. However, it is not continuous. It fails to be syntactically STMC because recursions occurs inside the primitive `the-set-of-all`. Although the above definition is not acceptable in Ontic, it does give an intuitive account of the universe of Ontic values. A formal treatment of the universe of Ontic values is given in section 10.

The main problem with STMC definitions of large universes is the practical difficulty of proving that particular objects are in them. Since no definable universe can contain all definable values, for any given universe one must use rules of inference to determine which denotable values are members of that universe. The inference rules given in section 11 for the impredicative thunks discussed in the next section are more practical than the rules for recursive definitions applied to recursively defined large universes.

[3]It seems that using STMC recursion one can only define objects of rank less than ω^2.

9 The Impredicative Universe

It is possible to approximate the universe of everything by introducing a primitive thunk `a-thing` which can generate all values definable without the use of this primitive itself. Given the primitive `a-thing` we can define a variety of other useful thunks.

```
(define (the-universe)
  (the-set-of-all (a-thing)))

(define (a-class)
  (a-subset-of (the-universe)))

(define (a-set)
  (both (a-class) (a-thing)))

(define (a-cons-cell)
  (cons (a-thing) (a-thing)))

(define (an-operator)
  (an-operator-from-to (a-class) (the-universe)))

(define (a-thunk)
  (a-thunk-to (the-universe)))
```

Expressions that do not involve the thunk `a-thing` are called *predicative*. Intuitively, predicative expressions are well defined — their meaning is deterministically derived from the fundamental notions of symbols, pairs, and monotone and continuous recursion. Impredicative expressions, those involving the primitive `a-thing`, rely on the somewhat undefined meaning of this primitive. The impredicative universe `(the-set-of-all (a-thing))` contains all values of predicative expressions. However, the impredicative universe does not contain itself. The semantics of the primitive `a-thing` is formally described in section 10.

The primitive `a-thing` and the other "large thunks" defined above are useful for defining "polymorphic" objects such as the following.

```
(define (a-list)
  (either 'nil
          (cons (a-thing) (a-list))))
```

```
(define (a-list-of (t (a-thunk)))
  (either 'nil
          (cons (t) (a-list-of t))))

(define (map (f (an-operator))
             (l (a-list-of (lambda () (a-member-of (domain-of f))))))
  (if (is l 'nil)
      'nil
      (cons (f (car l)) (map f (cdr l)))))

(define (a-set-of (t (a-thunk)))
  (a-subset-of (the-set-of-all (t))))

(define (insert (x (a-thing)) (s (a-set)))
  (the-set-of-all (either x (a-member-of s))))

(define (union (s1 (a-set)) (s2 (a-set)))
  (the-set-of-all (either (a-member-of s1) (a-member-of s2))))
```

The impredicative universe also allows for the construction of standard mathematical structures such as groups, rings, fields, topological spaces, and so on. For example the concept of a group can be defined as follows.

```
(define (a-function-from-to2 (d1 (a-set)) (d2 (a-set)) (d3 (a-set)))
  (some f (an-opertator-from-to d1 (the-set-of-all (an-operator-from-to d2 d3)))
    (forall ((x (a-member-of d1))
             (y (a-member-of d2)))
      (singleton (f x y)))))

(define (a-group)
  (let ((domain (a-set)))
    (let ((operator (a-function-from-to2 domain domain domain)))
      (when (and ... group axioms ...)
        (list 'a-group domain op)))))

(define (group-dom (g (a-group))) (car (cdr g)))

(define (group-op (g (a-group)) (car (cdr (cdr g)))))
```

Defining structures is common in mathematics and computer science. To simplify the definition of structures we can add syntactic sugar for structure definitions. The syntactic sugar presented below abbreviates the definitions given above and is syntactically modeled after the Common Lisp defstruct primitive.

```
(defstruct (a-group)
  (group-dom (a-set))
  (group-op (a-function-from-to2 group-dom group-dom group-dom))
  such-that ...  group axioms ...)
```

10 Formal Syntax and Semantics

This section presents the formal syntax and semantics of the Ontic language. A set of inference rules for Ontic is presented in section 11. A grammar for the syntax of Ontic terms and formulas is given below. In the grammar $\mathcal{V}$ represents an infinite collection of variables, $\mathcal{S}$ represents an infinite set of quoted symbols, $\mathcal{T}$ represents the set of terms and $\mathcal{F}$ represents the set of formulas. Some of the formulas can be expressed in terms of others but inclusion of these redundant formulas simplifies the presentation of the inference rules presented in section 11.

$$
\begin{array}{rcl}
\mathcal{T} & ::= & \mathcal{C} \mid \mathcal{V} \mid \texttt{a-symbol} \mid \texttt{(cons } \mathcal{T}\ \mathcal{T}\texttt{)} \mid \texttt{(car } \mathcal{T}\texttt{)} \mid \texttt{(cdr } \mathcal{T}\texttt{)} \mid \\
& & \texttt{(either } \mathcal{T}\ \mathcal{T}\texttt{)} \mid \texttt{(if } \mathcal{F}\ \mathcal{T}\ \mathcal{T}\texttt{)} \mid \texttt{(lambda ((}\mathcal{V}\ \mathcal{T}\texttt{)) } \mathcal{T}\texttt{)} \mid \texttt{(domain-of } \mathcal{T}\texttt{)} \mid \\
& & \texttt{(apply } \mathcal{T}\ \mathcal{T}\texttt{)} \mid \texttt{(lambda () } \mathcal{T}\texttt{)} \mid \texttt{(apply-thunk } \mathcal{T}\texttt{)} \mid \texttt{(}Y\ \mathcal{V}\ \mathcal{T}\texttt{)} \mid \\
& & \texttt{(the-set-of-all } \mathcal{T}\texttt{)} \mid \texttt{(a-member-of } \mathcal{T}\texttt{)} \mid \texttt{(a-subset-of } \mathcal{T}\texttt{)} \mid \\
& & \texttt{(an-operator-from-to } \mathcal{T}\ \mathcal{T}\texttt{)} \mid \texttt{(a-thunk-to } \mathcal{T}\texttt{)} \mid \texttt{a-thing} \mid \texttt{(choice } \mathcal{T}\texttt{)} \\
\mathcal{F} & ::= & \texttt{(is } \mathcal{T}\ \mathcal{T}\texttt{)} \mid \texttt{(or } \mathcal{F}\ \mathcal{F}\texttt{)} \mid \texttt{(not } \mathcal{F}\texttt{)} \mid \texttt{(there-exists } \mathcal{T}\texttt{)} \mid \\
& & \texttt{(at-most-one } \mathcal{T}\texttt{)} \mid \texttt{(singleton } \mathcal{T}\texttt{)} \mid \texttt{(is-symbol } \mathcal{T}\texttt{)} \mid \texttt{(is-set } \mathcal{T}\texttt{)} \mid \\
& & \texttt{(is-cons-cell } \mathcal{T}\texttt{)} \mid \texttt{(is-operator } \mathcal{T}\texttt{)} \mid \texttt{(is-thunk } \mathcal{T}\texttt{)} \mid \texttt{(small } \mathcal{T}\texttt{)}
\end{array}
$$

A variety of additional syntactic constructs can be introduced as abbreviations. The most common abbreviations, including quantified formulas, are described in section 2. Multiple argument λ-expressions and multiple argument applications are treated in the standard way using Currying. The phrase constructors `apply` and `apply-thunk` are generally suppressed — the application (`apply` f g) is written as (f g) and (`apply-thunk` f) is written as (f).

Definitions are used to introduce symbols as abbreviations for terms. A recursive definition introduces a symbol as an abbreviation for a fixed point expression. Consider the following recursive definition.

```
(define (an-expression)
  (either (a-symbol) (cons (an-expression) (an-expression))))
```

This definition introduces the symbol `an-expression` an abbreviation for (Y F `(lambda () (either (a-symbol) (cons (`F`) (`F`)))))`. In general, fixed point expressions (Y F $U[F]$) are restricted so that $U[F]$ must be of the form (`lambda` ((X_1 τ_1) ... (X_k τ_k)) $B[F\ X_1\ \ldots\ X_k]$) with $k \geq 0$ where F does not occur free in any τ_i and $B[F\ X_1\ \ldots\ X_k]$ is syntactically set theoretically monotone and continuous (STMC) in F as an operator on k arguments. This means that every free occurrence of F in $B[F\ X_1\ \ldots\ x_k]$ has the form (F u_1 ... u_k) and that no occurrence of F in $B[F\ X_1\ \ldots\ x_k]$ occurs inside a formula or inside the phrase constructors `the-set-of-all`, `choice` or Y or inside a `lambda` expressions other than the left hand sides of applications.

To define the semantics of Ontic we first need the notion of a universe. A universe is a "standard" model of Zermelo Fraenkel set theory.

> **Definition:** A *universe* is a family of sets U satisfying the following conditions.
>
> - If $s \in U$ then $P(s) \in U$ where $P(s)$ is the set of all subsets of s.
> - If $s \in U$ then $s \subseteq U$.
> - If $w \subseteq U$ and w is countable then $w \in U$.
> - If $s \in U$ and $w \subseteq U$, where $|w| \leq |s|$, then $w \in U$.
> - If $s \in U$ then the union of all sets in s is in U.

It is interesting to note that the above definition can be viewed as a recursive definition of a family of sets. One can start with the empty set and continue to add sets forced by the definition (starting with the power set of the empty set). This process will reach a fixed point at the first strongly inaccessible cardinal (assuming that such cardinals exist). The above definition allows for universes larger than this least fixed point.

The semantics of Ontic is defined relative to three parameters — U_0, U_1 and C. U_0 and U_1 must be universes and U_1 must contain U_0 as a member. The universe U_0 will be used as a model of the impredicative thunk `a-thing`. C must be a function from U_1 to U_1 such that for any nonempty element x of U_1 we have $C(x) \in x$. C is used to give the semantics of the choice primitive. Although we can not prove from the foundations of set theory that these object exist, all of modern large cardinal theory and much of modern category theory are based on assumptions implying the existence of such objects.

In Ontic we distinguish five kinds of semantic values — symbols, cons cells, sets, operators and thunks. Any encoding of these values as sets is sufficient provided that the representation of a value determines all the relevant properties of that value, including what kind of value it is. For the sake of definiteness we give one particular representation here. We assume that natural numbers and tuples are represented in the standard way as sets in U_1.[4] First we define a representation of symbols. A symbol is represented by a list of numbers the first of which gives the length of the list. For example, `'foo` is represented by the tuple $<3,\ 6,\ 15,\ 15>$. The first number 3 indicates that the symbol is spelled with three letters, and the numbers 6, 15, and 15 indicate that the letters consists of the 6th letter followed by two copies of the 15th letter. In the remainder of this section we use expressions such as `'foo` as synonyms for the corresponding representation of the symbol as a tuple. To distinguish the different kinds of Ontic values we use a tagged representation. Every Ontic value is a tuple whose first component is a one of the "tags" `'the-symbol`, `'the-cons-cell`, `'the-set`, `'the-operator` or `'the-thunk`. We now define the semantic notion of a value.

[4]The number 0 is represented by the empty set and $n+1$ is represented by $n \cup \{n\}$. The pair $<x,\ y>$ is represented by the set $\{x,\ \{x,y\}\}$. The n-tuple $<x_1,\ x_2,\ \ldots,\ x_n>$ is represented by $<x_1, <x_2,\ <\ldots\ x_n> \ldots>>$.

Definition: An *Ontic value* is any one of the following.

- $<$`'the-symbol`, $x>$ where x is a symbol (as defined above).
- $<$`'the-cons-cell`, $x,\ y>$ where x and y are values.
- $<$`'the-set`, $x>$ where x is a set of values.
- $<$`'the-operator`, $d,\ r>$ where d is a set of values and r is a set of pairs of values whose first component is an element of d. The set d is the domain of the operator and the set r is the set of input/output pairs of the operator.
- $<$`'the-thunk`, $x>$ where x is a set of values.

The above definition is recursive. The intended meaning is the least fixed point over the universe U_1.[5]

The meaning of Ontic terms and formulas is defined by a semantic value function. The value function takes an Ontic expression and an Ontic variable interpretation and returns a semantic value. The value function is defined by structural recursion on Ontic expressions. If ρ is a mapping from variables to values, and u is an Ontic term, then $V[u,\ \rho]$ is a set of values that is a member of U_1 — namely the set of possible values of the term u under variable interpretation ρ. If Φ is a formula then $V[\Phi,\ \rho]$ is a truth value.

Definition: An *Ontic variable interpretation* is a mapping from Ontic variables to Ontic values.

Definition: Let ρ be an Ontic variable interpretation. For any Ontic expression e we define $V[e,\ \rho]$ as follows.

- For any variable X we have $V[X,\ \rho] = \{\rho[X]\}$
- If s is a quoted symbol then $V[s,\ \rho] = \{<$`'the-symbol`, $s>\}$

[5]The recursive definition of an Ontic value will have larger fixed points over larger universes.

- $V[\texttt{a-symbol},\ \rho] = \{<\texttt{'the-thunk},\ s>\}$ where s is the set of all symbol values.

- $V[\texttt{(cons}\ u\ w\texttt{)},\ \rho]$

$$= \{<\texttt{'the-cons-cell},\ x,\ y> :\ x \in V[u,\ \rho] \land y \in V[w,\ \rho]\}$$

- $V[\texttt{(car}\ u\texttt{)},\ \rho] = \{y :\ \exists w,\ \exists z \in V[u,\ \rho], z = <\texttt{'the-cons-cell},\ y,\ w>\}$

- $V[\texttt{(cdr}\ u\texttt{)},\ \rho] = \{y :\ \exists w,\ \exists z \in V[u,\ \rho], z = <\texttt{'the-cons-cell},\ w,\ y>\}$

- $V[\texttt{(either}\ u\ w\texttt{)},\ \rho] = V[u,\ \rho] \cup V[w,\ \rho]$

- $V[\texttt{(if}\ \Phi\ u\ w\texttt{)},\ \rho] = \begin{cases} V[u,\ \rho] & \text{if } V[\Phi,\ \rho] = T \\ V[w,\ \rho] & \text{if } V[\Phi,\ \rho] = F \end{cases}$

- $V[\texttt{(lambda ((}X\ \tau\texttt{))}\ B[X]\texttt{)},\ \rho]$

  $= \{<\texttt{'the-operator},\ V[\tau,\ \rho],\ \{<z,\ y> :\ z \in V[\tau, \rho],\ y \in V[B[X],\ \rho[X := z]]\}>\}$

  where $\rho[X := z]$ is the Ontic variable interpretation identical to ρ except that it maps X to z

- $V[\texttt{(domain-of}\ u\texttt{)},\ \rho]$

$$= \{<\texttt{'the-set},\ d> :\ \exists p\ \exists f \in V[u,\ \rho],\ f = <\texttt{'the-operator},\ d,\ p>\}$$

- $V[\texttt{(apply}\ u\ w\texttt{)},\ \rho]$

  $= \{y :\ \exists f \in V[u,\ \rho]\ \exists\ x \in V[w,\ \rho]\ \exists p,\ d,\ f = <\texttt{'the-operator},\ d,\ p> \land <x,\ y> \in p\}$

- $V[\texttt{(lambda ()}\ u\texttt{)},\ \rho] = \{<\texttt{'the-thunk},\ V[u,\ \rho]>\}$

- $V[\texttt{(apply-thunk}\ u\texttt{)},\ \rho]$

$$= \{y :\ \exists f \in V[u,\ \rho],\ \exists s,\ f = <\texttt{'the-thunk},\ s> \land y \in s\}$$

- $V[\texttt{(Y } F\ U[F]\texttt{)},\ \rho]$

  $$= V[U[\texttt{(apply-thunk } H\texttt{)}],\ \rho[H := \texttt{<'the-thunk},\ \bigcup_{n\geq 0} G^n(\{\bot\})\texttt{>}]]$$

  where $\bot$ is $\texttt{<'the-operator},\ \emptyset,\ \emptyset\texttt{>}$, G is a function mapping sets to sets defined by

  $$G[s] = V[U[\texttt{(apply-thunk } H\texttt{)}],\ \rho[H := \texttt{<'the-thunk},\ s\texttt{>}]]$$

  and $G^n(\{\bot\})$ denotes n applications of G to the set $\{\bot\}$

- $V[\texttt{(the-set-of-all } u\texttt{)},\ \rho] = \{\texttt{<'the-set},\ V[u,\ \rho]\texttt{>}\}$

- $V[\texttt{(a-member-of } u\texttt{)},\ \rho]$

  $$= \{y : \exists s \in V[u,\ \rho],\ \exists z,\ s = \texttt{<'the-set},\ z\texttt{>} \wedge y \in z\}$$

- $V[\texttt{(a-subset-of } u\texttt{)},\ \rho]$

  $$= \{\texttt{<'the-set},\ y\texttt{>} : \exists s \in V[u,\ \rho],\ \exists z,\ \ s = \texttt{<'the-set},\ z\texttt{>} \wedge y \subseteq z\}$$

- $V[\texttt{(an-operator-from-to } u\ w\texttt{)},\ \rho]$

  $$= \{\texttt{<'the-operator},\ d_1,\ r\texttt{>} :\ \exists t \in V[u,\ \rho],\ t = \texttt{<'the-set},\ d_1\texttt{>}$$

  $$\wedge \exists s \in V[w,\ \rho],\ \exists d_2,\ s = \texttt{<'the-set},\ d_2\texttt{>} \wedge r \subseteq d_1 \times d_2\}$$

- $V[\texttt{(a-thunk-to } u\texttt{)},\ \rho]$

  $$= \{\texttt{<'the-thunk},\ d\texttt{>} : \exists s \in V[u,\ \rho],\ \exists d_2,\ s = \texttt{<'the-set},\ d_2\texttt{>} \wedge d \subseteq d_2\}$$

- $V[\texttt{a-thing},\ \rho] = \{\texttt{<'the-thunk},\ s\texttt{>}\}$ where s is the set of all Ontic values that are members of U_0

- $V[\texttt{(choice } u\texttt{)},\ \rho] = \begin{cases} \{C[V[u,\ \rho]]\} & \text{if } V[u,\ \rho] \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$

- $V[\texttt{(is } u\ w\texttt{)},\ \rho] = \begin{cases} T & \text{if } V[u,\ \rho] \subseteq V[w,\ \rho] \\ F & \text{otherwise} \end{cases}$

- $V[\texttt{(or } \Phi\ \Psi\texttt{)},\ \rho] = \begin{cases} T & \text{if } V[\Phi,\ \rho] = T \text{ or } V[\Psi,\ \rho] = T \\ F & \text{otherwise} \end{cases}$

- $V[\texttt{(not } \Phi\texttt{)},\ \rho] = \begin{cases} T & \text{if } V[\Phi,\ \rho] = F \\ F & \text{otherwise} \end{cases}$

- $V[\texttt{(there-exists } u\texttt{)},\ \rho] \begin{cases} T & \text{if } V[u,\ \rho] \neq \emptyset \\ F & \text{otherwise} \end{cases}$

- $V[\texttt{(at-most-one } u\texttt{)},\ \rho] \begin{cases} T & \text{if } |V[u,\ \rho]| \leq 1 \\ F & \text{otherwise} \end{cases}$

- $V[\texttt{(singleton } u\texttt{)},\ \rho] \begin{cases} T & \text{if } |V[u,\ \rho]| = 1 \\ F & \text{otherwise} \end{cases}$

- $V[\texttt{(is-symbol } u\texttt{)},\ \rho] \begin{cases} T & \text{if every member of } V[u,\ \rho] \text{ is a symbol value} \\ F & \text{otherwise} \end{cases}$
  A similar condition holds for formulas of the form `(is-set` u`)`, `(is-cons-cell` u`)`, `(is-operator` u`)` and `(is-thunk` u`)`.

- $V[\texttt{(small } u\texttt{)},\ \rho] \begin{cases} T & \text{if the set } V[u,\ \rho] \text{ is a member of } U_0 \\ F & \text{otherwise} \end{cases}$

11 Inference Rules

This section presents inference rules for Ontic. The inference rules given here are a mixture of ordinary rules and natural deduction rules. It should be straightforward to verify the soundness of each rule relative to the semantics of the previous section. The rules can not be semantically complete because the true formulas of Ontic include the true formulas of arithmetic and hence are not recursively enumerable. However, the rules have been designed to incorporate all of the inference principles of ZFC set theory applied to the denotational semantics of the expressions. Any formula of set theory has a natural representation in Ontic in which each set theoretic quantifier is replaced by an Ontic quantifier ranging over the type `(a-set)` (as defined in section 9). For any theorem of ZFC set theory the corresponding Ontic formula is provable using the rules of this section. To prove this one can simply verify that any instance of the axioms of ZFC is provable in Ontic. Actually, because includes a term denoting the impredicative universe, Ontic can prove the consistency of ZFC and so is somewhat more powerful than ZFC. Of course the inference rules for Ontic should be adequate for proving properties of symbols, cons cells, operators and thunks as well as sets.

First we give some "pure formula" rules, i.e, rules not involving term constructors. We assume some complete set of natural deduction rules for Boolean connectives. The formula constructors `is-symbol`, `is-set` and so on will be called "classifiers". Formulas of the form (`small` u) are used in the inference rules for the thunk `a-set`.

- (is u u)

- (is u w)
  (is w s)
  ____
  (is u s)

- (there-exists u)
  (is u w)
  ____
  (there-exists w)

- (is u w)
  (at-most-one w)
  ____
  (at-most-one u)

- (is u w)
  (there-exists u)
  (at-most-one w)
  ____
  (is w u)

- (there-exists u)
  (at-most-one u)
  ____
  (singleton u)

- (singleton u)
  ____
  (there-exists u)

- $\dfrac{\texttt{(singleton } u\texttt{)}}{\texttt{(at-most-one } u\texttt{)}}$

- $\dfrac{\texttt{(singleton } u\texttt{)}}{\texttt{(or (is-symbol } u\texttt{) (is-set } u\texttt{) (is-cons-cell } u\texttt{) (is-operator } u\texttt{) (is-thunk } u\texttt{))}}$

- $\dfrac{\texttt{(is-foo } u\texttt{)}}{\texttt{(not (is-bar } u\texttt{))}}$ where `is-foo` and `is-bar` are distinct classifiers

In the following rules X, Y and Z range over variables.

- $\texttt{(singleton } X\texttt{)}$

- $\dfrac{\Sigma, \texttt{(is } X\ u\texttt{)}, \texttt{(is } Y\ u\texttt{)} \vdash \texttt{(is } X\ Y\texttt{)}}{\Sigma, \vdash \texttt{(at-most-one } u\texttt{)}}$ X, Y not free in Σ

- $\dfrac{\Sigma, \texttt{(is } X\ u\texttt{)} \vdash \texttt{(is } X\ w\texttt{)}}{\Sigma \vdash \texttt{(is } u\ w\texttt{)}}$ X not free in Σ, u or w

- $\dfrac{\Sigma, \texttt{(is } X\ u\texttt{)} \vdash \Phi}{\Sigma, \texttt{(there-exists } u\texttt{)} \vdash \Phi}$ X not free in Σ or Φ

- $\dfrac{\Sigma, \texttt{(is } X\ u\texttt{)} \vdash \texttt{(is-foo } X\texttt{)}}{\Sigma \vdash \texttt{(is-foo } u\texttt{)}}$ `is-foo` a classifier and X not free in Σ or u

The term constructors `cons`, `car`, `cdr`, `either`, `domain-of`, `apply`, `apply-thunk`, `a-member-of`, `a-subset-of`, `an-operator-from-to` and `a-thunk-to` will be called *regular* constructors. The remaining constructors `lambda`, `the-set-of-all`, `choice`, Y and `if` will be called *irregular* constructors. We have the following rules for regular constructors c. Each rule has a monadic and a binary form.

- $\dfrac{\texttt{(is } u\ v\texttt{)}}{\texttt{(is (}c\ u\texttt{) (}c\ v\texttt{))}}$

- $\dfrac{\texttt{(is } u_1\ v_1\texttt{)} \quad \texttt{(is } u_2\ v_2\texttt{)}}{\texttt{(is (}c\ u_1\ u_2\texttt{) (}c\ v_1\ v_2\texttt{))}}$

- $$\frac{\Sigma, \texttt{(is } X \texttt{ (} c\ Y\texttt{))}, \texttt{(is } Y\ u\texttt{)} \vdash \Phi}{\Sigma, \texttt{(is } X \texttt{ (} c\ u\texttt{))} \vdash \Phi} \quad Y \text{ not free in } \Sigma,\ u, \text{ or } \Phi$$

- $$\frac{\Sigma, \texttt{(is } X \texttt{ (} c\ Y\ Z\texttt{))}, \texttt{(is } Y\ u\texttt{)}, \texttt{(is } Z\ w\texttt{)} \vdash \Phi}{\Sigma, \texttt{(is } X \texttt{ (} c\ u\ w\texttt{))} \vdash \Phi} \quad Y \text{ and } Z \text{ not free in } \Sigma,\ u,\ w \text{ or } \Phi$$

We now give inference rules specific to the various features of Ontic. First we give the rules for symbols.

- `(singleton` s`)` where s is a quoted symbol
- `(is-symbol` s`)` where s is a quoted symbol
- `(not (is` s_1 s_2`))` where s_1 and s_2 are distinct quoted symbols
- `(is-thunk a-symbol)`
- $$\frac{\texttt{(is } u \texttt{ (apply-thunk a-symbol))}}{\texttt{(is-symbol } u\texttt{)}}$$
- $$\frac{\texttt{(is-symbol } u\texttt{)}}{\texttt{(is } u \texttt{ (apply-thunk a-symbol))}}$$

Next we give the rules for cons cells. In these rules, and in the remainder of this section, the formula `(=` u v`)` is used as an abbreviation for `(and (is` u v`)` `(is` v u`))`.

- `(is-cons-cell (cons` u w`))`
- $$\frac{\texttt{(is-cons-cell } u\texttt{)} \quad \texttt{(singleton } u\texttt{)}}{\texttt{(= } u \texttt{ (cons (car } u\texttt{) (cdr } u\texttt{)))}}$$
- $$\frac{\texttt{(singleton } u\texttt{)} \quad \texttt{(singleton } v\texttt{)}}{\texttt{(singleton (cons } u\ v\texttt{))}}$$

- $$\frac{\begin{array}{l}\texttt{(there-exists (car } u\texttt{))}\\ \texttt{(singleton } u\texttt{)}\end{array}}{\texttt{(is-cons-cell } u\texttt{)}}$$

- $$\frac{\texttt{(there-exists } w\texttt{)}}{\texttt{(= (car (cons } u\ w\texttt{)) } u\texttt{)}}$$

- $$\frac{\begin{array}{l}\texttt{(there-exists (cdr } u\texttt{))}\\ \texttt{(singleton } u\texttt{)}\end{array}}{\texttt{(is-cons-cell } u\texttt{)}}$$

- $$\frac{\texttt{(there-exists } u\texttt{)}}{\texttt{(= (cdr (cons } u\ w\texttt{)) } w\texttt{)}}$$

Now we give the rules for `either`.

- $\texttt{(is } u \texttt{ (either } u\ w\texttt{))}$

- $\texttt{(is } w \texttt{ (either } u\ w\texttt{))}$

- $$\frac{\texttt{(is } X \texttt{ (either } u\ w\texttt{))}}{\texttt{(or (is } X\ u\texttt{) (is } X\ w\texttt{))}}$$

And the rules for `if`.

- $$\frac{\Phi}{\texttt{(= (if } \Phi\ u\ w\texttt{) } u\texttt{)}}$$

- $$\frac{\texttt{(not } \Phi\texttt{)}}{\texttt{(= (if } \Phi\ u\ w\texttt{) } w\texttt{)}}$$

Now the rules for operators.

- $\texttt{(is-operator (lambda ((}X\ \tau\texttt{)) } B[X]\texttt{))}$

- $$\frac{\begin{array}{l}\texttt{(is-operator } f\texttt{)}\\ \texttt{(singleton } f\texttt{)}\end{array}}{\texttt{(= } f \texttt{ (lambda ((}X \texttt{ (a-member-of (domain-of } f\texttt{)))) (apply } f\ X\texttt{)))}} \qquad X \text{ not free in } u$$

- $\texttt{(singleton (lambda ((}X\ \tau\texttt{)) } B[X]\texttt{))}$

- $$\frac{\texttt{(there-exists (apply (lambda ((}X\ \tau\texttt{))}\ B[X]\texttt{)}\ u\texttt{))} \qquad \texttt{(singleton }u\texttt{)}}{\texttt{(is }u\ \tau\texttt{)}}$$

- $$\frac{\texttt{(singleton }u\texttt{)} \qquad \texttt{(is }u\ \tau\texttt{)}}{\texttt{(= (apply (lambda ((}X\ \tau\texttt{))}\ B[X]\texttt{)}\ u\texttt{)}\ B[u]\texttt{)}}$$

- $$\frac{\begin{array}{l}\Sigma \vdash \texttt{(is-operator }f\texttt{)}\\ \Sigma \vdash \texttt{(is-operator }g\texttt{)}\\ \Sigma \vdash \texttt{(= (domain-of }f\texttt{) (domain-of }g\texttt{))}\\ \Sigma, \texttt{(is }X\ \texttt{(a-member-of (domain-of }f\texttt{)))} \vdash \texttt{(= (apply }f\ X\texttt{) (apply }g\ X\texttt{))}\end{array}}{\Sigma \vdash \texttt{(= }f\ g\texttt{)}} \quad X \text{ not free in } \Sigma,\ f, \text{ or } g$$

- $$\frac{\texttt{(there-exists (apply }f\ u\texttt{))} \qquad \texttt{(singleton }f\texttt{)}}{\texttt{(is-operator }f\texttt{)}}$$

- $$\frac{\texttt{(there-exists (domain-of }f\texttt{))} \qquad \texttt{(singleton }f\texttt{)}}{\texttt{(is-operator }f\texttt{)}}$$

- $\texttt{(= (domain-of (lambda ((}X\ \tau\texttt{))}\ B[X]\texttt{)) (the-set-of-all }\tau\texttt{))}$

Now the rules for thunks.

- $\texttt{(is-thunk (lambda ())}\ u\texttt{))}$

- $$\frac{\texttt{(is-thunk }f\texttt{)} \qquad \texttt{(singleton }f\texttt{)}}{\texttt{(= }f\ \texttt{(lambda () (apply-thunk }f\texttt{)))}}$$

- $\texttt{(singleton (lambda ()}\ u\texttt{))}$

- $\texttt{(= (apply-thunk (lambda ()}\ u\texttt{))}\ u\texttt{)}$

- $$\frac{\begin{array}{l}\texttt{(is-thunk }f\texttt{), (singleton }f\texttt{)}\\ \texttt{(is-thunk }g\texttt{), (singleton }g\texttt{)}\\ \texttt{(= (apply-thunk }f\texttt{) (apply-thunk }g\texttt{))}\end{array}}{\texttt{(= }f\ g\texttt{)}}$$

- `(there-exists (apply-thunk` f`))`
  `(singleton` f`)`
  ―――――――――――――
  `(is-thunk` f`)`

Now the rules for fixed points. The first rule is simple.

- `(= (`Y F $U[F]$`)` $U[(Y\ F\ U[F])]$`)`

To state the induction rule let $\Psi[G]$ be a formula of the following form involving the variable G.

```
(forall ((Y_1 γ_1)
           ⋮
         (Y_k γ_k)
         (Z (G u_1 ... u_n)))
  Φ[Y_1, ..., Y_k, Z])
```

We are using universal quantification as an abbreviation as described in section 2. We require that the only occurrence of the variable G in $\Psi[G]$ be the one explicitly shown. We now have the standard fixed point induction rule for formulas of this form.

- $$\frac{\Sigma, \Psi[G] \vdash \Psi[U[G]]}{\Sigma \vdash \Psi[(Y\ F\ U[F])]}$$

Unlike CPO fixed point induction, the above restricted form of $\Psi[G]$ is adequate for STMC fixed point induction. Using this induction principle it is possible to prove a set theoretic formula stating that $(Y\ F\ U[F])$ is in fact the least set theoretic fixed point of U.

Now the rules for sets.

- `(is-set (the-set-of-all` u`))`

- `(is-set` s`)`
  `(singleton` s`)`
  ―――――――――――――
  `(=` s `(the-set-of-all (a-member-of` s`)))`

- `(singleton (the-set-of-all` u`))`

- (= (a-member-of (the-set-of-all *u*)) *u*)

- (is-set (a-subset-of *s*))

- (is-set *s*)
(singleton *w*)
(is (a-member-of *s*) (a-member-of *w*))
――――――――――――
(is *s* (a-subset-of *w*))

- (is *s* (a-subset-of *w*))
――――――――――――
(is (a-member-of *s*) (a-member-of *w*))

- (is *s* (a-subset-of *w*))
(is *w* (a-subset-of *s*))
――――――――――――
(= *s* *w*)

- (there-exists (a-member-of *s*))
(singleton *s*)
――――――――――――
(is-set *s*)

- (there-exists (a-subset-of *s*))
(singleton *s*)
――――――――――――
(is-set *s*)

For compatibility with ZFC, we include the axiom of foundation.

```
(forall ((s (a-set)))
  (exists ((x (a-member-of s)))
    (not (there-exists (both (a-member-of s) (a-member-of x))))))
```

Now the rules for function spaces.

- (is-operator (an-operator-from-to *u* *w*))

- (there-exists (an-operator-from-to *u* *w*))
(singleton *u*)
――――――――――――
(is-set *u*)

- (there-exists (an-operator-from-to *u* *w*))
(singleton *w*)
――――――――――――
(is-set *w*)

- $$\frac{\texttt{(is } f \texttt{ (an-operator-from-to } u\ w\texttt{))}}{\texttt{(is (domain-of } f\texttt{) } u\texttt{)}}$$
- $$\frac{\texttt{(is } f \texttt{ (an-operator-from-to } u\ w\texttt{))}}{\texttt{(is (apply } f \texttt{ (a-member-of } u\texttt{)) (a-member-of } w\texttt{))}}$$
- $$\frac{\begin{array}{l}\texttt{(is-operator } f\texttt{)}\\ \texttt{(is-set } w\texttt{)}\\ \texttt{(singleton } w\texttt{)}\\ \texttt{(is (apply } f \texttt{ (a-member-of (domain-of } f\texttt{))) (a-member-of } w\texttt{))}\end{array}}{\texttt{(is } f \texttt{ (an-operator-from-to (domain-of } f\texttt{) } w\texttt{))}}$$
- (is-thunk (a-thunk-to u))
- $$\frac{\begin{array}{l}\texttt{(there-exists (an-thunk-to } w\texttt{))}\\ \texttt{(singleton } w\texttt{)}\end{array}}{\texttt{(is-set } w\texttt{)}}$$
- $$\frac{\texttt{(is } f \texttt{ (a-thunk-to } u\texttt{))}}{\texttt{(is (apply-thunk } f\texttt{) (a-member-of } u\texttt{))}}$$
- $$\frac{\begin{array}{l}\texttt{(is-thunk } f\texttt{)}\\ \texttt{(is-set } w\texttt{)}\\ \texttt{(singleton } w\texttt{)}\\ \texttt{(is (apply-thunk } f\texttt{) (a-member-of } w\texttt{))}\end{array}}{\texttt{(is } f \texttt{ (a-thunk-to } w\texttt{))}}$$

We now consider the primitive `a-thing`.

- (is-thunk a-thing)
- $$\frac{\texttt{(small } u\texttt{)}}{\texttt{(is } u \texttt{ (apply-thunk a-thing))}}$$
- $$\frac{\begin{array}{l}\texttt{(is } u \texttt{ (apply-thunk a-thing))}\\ \texttt{(singleton } u\texttt{)}\end{array}}{\texttt{(small } u\texttt{)}}$$
- $$\frac{\texttt{(small } u\texttt{)}}{\texttt{(small (}c\ u\texttt{))}}\quad c \text{ any monadic term constructor}$$
- $$\frac{\begin{array}{l}\texttt{(small } u\texttt{)}\\ \texttt{(small } w\texttt{)}\end{array}}{\texttt{(small (}c\ u\ w\texttt{))}}\quad c \text{ a binary term constructor other than } \texttt{lambda} \text{ or } Y$$
- (small a-symbol)
- $$\frac{\begin{array}{l}\Sigma \vdash \texttt{(small } \tau\texttt{)}\\ \Sigma, \texttt{(small } X\texttt{)} \vdash \texttt{(small } B[X]\texttt{)}\end{array}}{\Sigma \vdash \texttt{(small (lambda ((}X\ \tau\texttt{)) } B[X]\texttt{))}}\quad X \text{ not free in } \Sigma$$

- $\dfrac{\texttt{(small } u\texttt{)}}{\texttt{(small (lambda () } u\texttt{))}}$

- $\dfrac{\Sigma, \texttt{(small } G\texttt{)} \vdash \texttt{(small } U[G]\texttt{)}}{\Sigma \vdash \texttt{(small (} Y \;\; F \;\; U[F]\texttt{))}}$ $\quad G$ not free in Σ or U

Now the rules for `choice`.

- `(is (choice u) u)`

- $\dfrac{\texttt{(there-exists } u\texttt{)}}{\texttt{(singleton (choice } u\texttt{))}}$

- $\dfrac{\texttt{(not (there-exists } u\texttt{))}}{\texttt{(not (there-exists (choice } u\texttt{)))}}$

12 Conclusion

The primary novel feature of Ontic is the use of nondeterminism to simply formal representation. This use of nondeterminism to eliminate a separate syntactic category of types is motivated here as a way of simplifying definitions. Although type inference algorithms have not been discussed here, the use of nondeterminism as a representation for types can also be motivated as a technique for making type inference more generally applicable (see [6]). This paper is primarily about the syntax and semantics of a highly expressive and concise formal representation language. It is hoped that a variety of different approaches can be used to build verification systems for the language defined in this paper.

References

[1] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic model checking:10^{20} states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, June 1990.

[2] Robert Givan and David McAllester. New results on local inference relations. In *Principles of Knolwedge Representation and Reasoning: Proceedings of the Third International Conference*, pages 403–412. Morgan Kaufman Press, October 1992.

[3] Michael Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*. Springer-Verlag, 1979. Volume 78 of Lecture Notes in Computer Science.

[4] D. McAllester. Automatic recognition of tractability in inference relations. *JACM*, 40(2):284–303, April 1993.

[5] D. McAllester and R. Givan. Natural language syntax and first order inference. *Artificial Intelligence*, 56:1–20, 1992.

[6] D. McAllester and R. Givan. Taxonomic syntax for first order inference. *JACM*, 40(2):246–283, April 1993.

[7] David A. McAllester. *Ontic: A Knowledge Representation System for Mathematics*. MIT Press, 1989.

[8] John McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg, editors, *Computer Programing and Formal Systems*. North-Holland, 1967.

[9] D. Park. Finiteness is μ-ineffable. *Theoretical Computer Science*, 3(2):173–181, 1976.

[10] L. C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–385. Academic Press, 1990.