Ontic: Language Specification and User's Manual

Robert Givan
David McAllester
Carl Witty
Kevin Zalondek

Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge Mass. 02139

Draft 4, March, 1992

Abstract: Ontic is an integrated system for the development and manipulation of technical information. Ontic can be used to develop and examine abstract mathematical concepts and theorems, formal system specifications, system implementations, and system verifications. At the foundation of the Ontic system is a formal language, also called Ontic. The Ontic language is a simple generalization of strongly typed functional programming languages such as ML or the typed λ-calculus. However, unlike functional programming languages, Ontic can be used to define objects that are best understood declaratively, such as the concept of a differentiable function on real numbers. The Ontic language is expressively equivalent to classical Zermelo-Fraenkel set theory with the axiom of choice. However, Ontic is also a functional programming language in the sense that a simple subset of the Ontic language is executable.

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-85-K-0124 and N00014-89-j-3202.

Contents

1 Introduction

Ontic is an integrated system for programming and mathematics. Programming and mathematics are similar activities.[1] Both involve definitions. In mathematics one defines concepts — in programming one defines procedures and subroutines. In both cases one often finds that, after writing certain definitions, the definitions do not have the desired properties — they do not allow one to prove a desirable theorem, or they do not produce the desirable behavior when executed. In both cases definitions must be "debugged". Mathematics and programming are also similar in other ways. For example, they both involve sophisticated notions of syntactically well formed expressions.

In spite of the great similarity of programming and mathematics, they are different in one important respect. Unlike theorems, programs run. This has a variety of implications. First, program execution often uncovers errors — execution serves as a kind of mechanical error detection. More importantly, however, execution often allows one to better understand programs. For example, if one does not understand the description of the append function given in these Lisp manual, one can simply apply the append procedure to `'(a b c)` and `'(d e f)` and see that the result `'(a b c d e f)`. Given this execution, one can form a conjecture about what the append procedure does in general. One can then reread the description in the manual and perhaps run some further test cases. In general, executing test cases can clarify one's understanding.

One of the primary motivations for the Ontic system is to provide a mechanism in the domain of mathematics that is analogous to running test cases for computer programs. We view running test cases for computer programs as analogous to answering mathematical questions. For example, given a definition of differentiable function, and a definition of polynomial, one can ask whether a polynomial is differentiable (it is). Or one could ask whether the

[1] The analogy between programming and mathematics underlying Ontic should not be confused with the "propositions as types" and "proofs as programs" framework espoused by intuitionistic logicians [Martin-Löf, 1982], [Constable *et al.*, 1986]. Ontic is a purely classical system — in Ontic a program denotes a classical function and a type denotes a classical set.

absolute value function is differentiable (it is not). Getting answers to specific questions can clarify one's understanding. This kind of question answering can also uncover errors in definitions.

The Ontic system responds to any question with "yes", "no", or "I don't know". This is done using a combination of knowledge and reasoning. Ontic's knowledge consists of a large set of definitions and theorems. Ontic's reasoning consists of an automated inference procedure which automatically accesses the stored knowledge. The inference procedure has been carefully designed so that for any question, no matter how difficult, the system always responds quickly. Furthermore, adding a fact to the knowledge base can never confuse the system — if a question could be be answered before the fact was added, that same question can still be answered after the fact is added. Great care has been given to making the system as intelligent as possible, but it can not be omniscient — there will always be questions for which the answer is "I don't know".

In addition to writing definitions of mathematical concepts, one can use the Ontic system to write and verify mathematical proofs. If the Ontic system answers "yes" when asked if a formula Φ is true, then we say that Φ is "obvious" to the Ontic system. A proof in the Ontic system consists of a series of statements where each statement is either self evident (relative to Ontic's knowledge base) or is derivable from previous steps using one of Ontic's built in proof rules. The Ontic system allows the user to define any mathematical concept and to prove any mathematical theorem provable from the axioms of classical Zermelo-Fraenkel set theory with the axiom of choice. Of course the intelligence of the Ontic system allows proofs to written much more concisely than in classical set theory.

Although Ontic is equivalent to set theory, the Ontic system does not use classical set theoretic syntax. In fact, Ontic avoids the syntax of first order logic altogether. The Ontic language is a generalization of functional programming languages such as Lisp. Some Ontic expressions can be treated as computer programs and executed. Other expressions are not executable but can be used in formulating questions. Even the nonexecutable expressions are written in a Lisp-like notation. Anyone familiar with Lisp should have little difficulty in learning to read and write Ontic expressions.

Like pure Lisp, Ontic provides a fairly small number of primitives that can be used to define a tremendous variety of procedures and concepts. The Ontic user is invited to develop her or his favorite branch of mathematics, or to invent and experiment with new mathematical concepts. Ontic should be thought of as an intelligent assistant for answering questions about user defined procedures and concepts. By answering questions, Ontic can uncover unintended properties of definitions (what programmers call "bugs"). By verifying proofs, Ontic can also provide increased confidence in the correctness of programs and mathematical theorems. Most of all, by being able to "understand" any mathematical concept, Ontic provides a new kind of environment in which mathematical hackers can explore and enjoy mathematics.

This manual is designed to support effective use of the Ontic system. Ontic is designed to be used with a minimum of training. It is hoped that an understanding of the inference mechanisms underlying the Ontic implementation is not required in using the system. This manual does not provide any such description of the inference mechanisms. However, it seems appropriate to make a few comments about the Ontic implementation and its relation to other systems.

Ontic's proof verification system is similar in motivation to a variety of other proof verification systems [Gordon *et al.*, 1979], [Boyer and Moore, 1979], [Constable *et al.*, 1986], [William Farmmer, 1990], [Harper *et al.*, 1987]. However, both the formal language of Ontic and the automated reasoning mechanisms underlying Ontic are significantly different from other systems. The Ontic language is simple. Although Ontic is a kind of type theory, there is no distinction between terms and types and the Ontic language Ontic is little more than pure untyped Lisp with nondeterminism. Both the semantics and the proof theory of Ontic correspond to classical set theory. The automated inference mechanisms are based on general purpose forward chaining techniques rather than term rewriting or backward chaining tactics [McAllester, 1989]. The forward chaining inference mechanisms are based on a general theory of local inference relations [McAllester *et al.*, 1989], [McAllester and Givan, 1989], [Givan *et al.*, 1991]. Readers interested in the deeper theoretical issues underlying Ontic and other verification systems might start by examining these references.

2 The Basic Language

The Ontic language is largely based on the syntax and semantics of Lisp. This section describes the Ontic language in much the same way that a Lisp manual would describe Lisp. However, those readers familiar with denotational semantics should not have great difficulty in assigning a fairly straightforward denotational meaning to Ontic expressions. Ontic includes the fundamental constructs of Lisp — numbers, the functions `+` and `*`, quoted symbols, the primitives `cons`, `car`, and `cdr`, conditional expressions (the primitive `if`), recursive definitions, and first class λ-expressions.[2] These primitives provide an adequate programming language. In addition, Ontic provides some nonstandard primitives. For example, Ontic provides the primitive `either` such that the expression `(either A B)` nondeterministically selects the value of `A` or the value of `B`. This makes Ontic a nondeterministic language — each expression has a set of possible values. Ontic includes a small number of additional primitives that provide all of the expressive power of Zermelo-Fraenkel set theory.

2.1 Symbols, Numbers, and Pairs

symbols and **quotation.** A quoted symbol denotes that symbol. For example, `'foo` denotes the symbol foo. Distinct quoted symbols always denote distinct values.

numbers, + and *. The Ontic language includes decimal numerals. A decimal numeral, such as 375 or -257, denotes the corresponding integer in the standard way. Rational numbers and real numbers are not supported as primitives. The Ontic language also includes the functions `+` and `*` which denote addition and multiplication on integers.

cons, car, and **cdr.** The function `cons` is the pairing function. This function takes two objects x and y and produces the pair of those two objects. For example, `(cons 'foo 'bar)` is the mathematical pair containing the symbols foo and bar. The expression `(cons 'foo 325)` denotes the pair containing

[2]Ontic is a purely functional language — side effects are not allowed.

the symbol foo and the integer 325. The functions `car` and `cdr` extract the first and second components of pairs respectively. For example, `(car (cons 'foo 325))` denotes the symbol foo while `(cdr (cons 'foo 325))` denotes the integer 325.

list. Expressions of the form `(list` e_1 e_2 $\cdots$ e_n`)` are treated as abbreviations for `(cons` e_1 `(cons` e_2 $\cdots$ `(cons` e_n `'nil)))`.

2.2 Nondeterminism

either. Ontic can be viewed as a *nondeterministic* programming language — a given expression can have many different possible values. Nondeterminism is introduced with the primitive `either`.[3] For example, the expression `(either 'foo 325)` has two possible values — the symbol foo and the integer 325. The expression `(list (either 'foo 325) (either 'foo 325) (either 'foo 325) (either 'foo 325))` has sixteen possible values, one of which is `(list 'foo 325 325 'foo)`. Ontic allows `either` to be used with any number of arguments. For example, `(either 'a 'b 'c 'd)` has four possible values.

Nondeterminism allows the Ontic language to be viewed as a notation for set theory. In general, each Ontic expression has a set of possible values. Formally, this is equivalent to saying that each Ontic expression denotes a set — its set of possible values. Although it is possible to think of Ontic as a purely declarative language in which expressions denote sets, many readers will find it easier to think of Ontic as a nondeterministic programming language in which a given expression has a set of possible values. The programming language view will be used throughout the remainder of this section.

[3]The primitive `either` is essentially the same as McCarthy's primitive `amb` [McCarthy, 1967].

2.3 Let

let. Ontic includes the special form `let` which can be used in expressions of the form `(let ((`x e`))` b`)` where x is a variable and e and b are Ontic expressions. The value of the expression `(let ((`x e`))` b`)` is the value of b in an environment in which x has been bound to the value of e. More precisely, each possible value of `(let ((`x e`))` b`)` can be derived by binding x to a possible value of e and then computing a possible value of b in the resulting environment.

The expression

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

has two possible values, `(cons 'foo 'foo)` and `(cons 'bar 'bar)`.

Note that `(cons (either 'foo 'bar) (either 'foo 'bar))` has four different possible values while the above expression has only two. This example shows that β-reduction, i.e., the substitution of an expression for a variable bound to that expression, does not preserve meaning.

The special form `let` can bind more than one variable. In Ontic the expression

```
(let ((x1 e1)
      ⋮
      (xn en))
  b)
```

is equivalent to

```
(let ((x1 e1))
  (let ((x2 e2))
    ⋮
```

```
(let ((x_n  e_n))
  b)))
```

Unlike most Lisp implementation of `let`, the Ontic implementation allows early variables to be used in latter bindings, e.g., x_1 can appear in e_2.

2.4 Thunks

A thunk is an expression of the form `(lambda ()` e`)` where e is some Ontic expression. The word *thunk* originated as a description of the data structure used to represent a call-by-name argument in Algol 60.[4] In Lisp, thunks are used to represent delayed computation. The value of a thunk `(lambda ()` e`)` is a "procedure" which, when called on no arguments, computes a value of e.

Thunks play an important rule in nondeterministic programming languages. For example, the expression

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

has two possible values while the expression

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

has four possible values.

an-integer and **a-symbol**. Ontic includes several primitive thunks. The thunk `an-integer`, when applied to no arguments, nondeterministically generates some integer. The thunk `a-symbol`, when applied to no arguments,

[4]Sussman and Abelson attribute the origin of the term thunk to the sound made by data structures when pushed onto the Algol stack [Sussman and Abelson, 1985].

nondeterministically returns a symbol. The expression

```
(let ((x (an-integer)))
  (+ x x))
```

has all the even integers as its set of possible values. Note that this is different from the expression `(+ (an-integer) (an-integer))` which has all integers as possible values. The expression `(cons (a-symbol) (an-integer))` has an infinite number of different possible values, each of which is a pair of a symbol and an integer.

2.5 Formulas and Conditional Expressions

In Ontic formulas are syntactically different from expressions. Expressions can have many different values or fail to have any values at all. A formula, on the other hand, always has exactly one truth value — a formula is always either true or false.

is. If e_1 and e_2 are Ontic expressions, then `(is` e_1 e_2`)` is an Ontic formula which is true if and only if every possible value of e_1 is also a possible value of e_2. For example, the formula `(is 2 (a-number))` is true while the formula `(is 'foo (a-number))` is false. The formula `(is 'foo 'bar)` is false while the formula `(is 'foo 'foo)` is true. Note that in the case where expressions e_1 and e_2 both have exactly one possible value, the formula `(is` e_1 e_2`)` is true just in case the value of e_1 is the same as the value of e_2. In other words, for expressions with exactly one value, the primitive `is` acts as an equality test. In expressions where e_1 has exactly one value and e_2 has many values, we can think of e_2 as expressing a type and the formula `(is` e_1 e_2`)` as expressing the statement that e_1 denotes an object of type e_2. In cases where e_1 and e_2 both have many possible values, the formula `(is` e_1 e_2`)` can be viewed as a statement that e_1 is a subtype of e_2. For example, if `an-even-integer` is a thunk which, which called, generates an even number, then the formula `(is (an-even-integer) (an-integer))` expresses the statement that every even integer is an integer. If e_1 has no possible values then the formula `(is` e_1 e_2`)` is true.

there-exists. If e is an Ontic expression then `(there-exists` e`)` is a formula which is true if e has some possible value. For example, the formula `(there-exists (a-symbol))` is true, while the formula

```
(there-exists (a-square-circle))
```

is false assuming that `a-square-circle` is a thunk that, when applied to no arguments, fails to have any value.

at-most-one. If e is an Ontic expression then `(at-most-one` e`)` is a formula which is true if there is at most one possible value of e. For example `(at-most-one (a-square-circle))` and `(at-most-one 'foo)` are both true while `(at-most-one (a-symbol))` is false.

and, or, not, implies and **iff.** Formulas can be constructed from other formulas using the Boolean operations. For example, if Φ and Ψ are formulas then so are `(not` Φ`)` and `(or` Φ Ψ`)`.

if. Formulas can be included in conditional expressions. A conditional expression is an expression of the form `(if` Φ e_1 e_2`)` where Φ is a formula and e_1 and e_2 are expressions. If Φ is true, then v is a possible value of `(if` Φ e_1 e_2`)` if and only if v is a possible value of e_1. If Φ is false then v is a possible value of of `(if` Φ e_1 e_2`)` if and only if v is a possible value of e_2.

when. Ontic includes the primitive `when` where an expression of the form `(when` Φ e`)` is equivalent to `(if` Φ e `(a-square-circle))`. In other words, if Φ is true then `(when` Φ e`)` is equivalent to e. However, if Φ is false then `(when` Φ e`)` has no possible values.

cond. Ontic also includes the Lisp special form `cond`. An expression of the form

`(cond` `(`Φ_1 e_1`)`
`(`Φ_2 e_2`)`
$\vdots$
`(`Φ_n e_n`))`

is semantically equivalent to

```
(if Φ1
    e1
    (if Φ2
        e2
          ⋮
            (when Φn en))).
```

Note that if none of the formulas in a `cond` expression are true then the `cond` expression has no possible values. This is different from Lisp where the value is nil if none of the conditions are true — having no value is quite different from having the value nil.

Equality. Ontic also includes equality. A formula of the form `(=` e_1 e_2`)` is true just in case the set of possible values of e_1 is the same as the set of possible values of e_2. This definition of the meaning of equality is essentially forced by the constraint that formulas have well defined truth values — a formula always has exactly one truth value no matter how many possible values the terms in that formula have. Semantically, the formula `(=` e_1 e_2`)` is equivalent to the the conjunction of `(is` e_1 e_2`)` and `(is` e_2 e_1`)`.

2.6 Some-such-that, Exists, and Forall

some-such-that. Ontic includes a primitive `some-such-that` for expressing generate and test expressions. The expression `(some-such-that` x e $\Phi(x)$`)`, where x is a variable e is an expression, and $\Phi(x)$ is a formula, is equivalent to the following.

```
(let ((x e))
  (when Φ(x) x))
```

For example, the expression

`(some-such-that` x `(an-integer) (= (* 3` x`) (+` x `8)))`

has a single possible value — the number 4.

exists and **forall**. Ontic also includes the primitives `exists` and `forall` for concisely expressing quantified statements. The formula `(exists ((`x e`))` $\Phi(x)$`)` is semantically equivalent to `(there-exists (some-such-that` x e $\Phi(x)$`)))`. For example, the following formula is true.

```
(exists ((x (an-integer)))
  (= (* 3 x) (+ x 8)))
```

The formula `(forall ((`x e`))` $\Phi(x)$`)` is semantically equivalent to `(is` e `(some-such-that` x e $\Phi(x)$`)))`. For example, the following formula is true.

```
(forall ((x (an-integer)))
  (exists ((y (an-integer)))
    (= (+ x y) 0)))
```

Ontic also allows the primitives `exists` and `forall` to bind several variables. As with `let`, the bindings are done sequentially. The formula

`(exists ((`x_1 e_1`) (`x_2 e_2`) ... (`x_n e_n`))`
$\Phi(x_1, \ldots, x_n)$`)`

abbreviates

`(exists ((`x_1 e_1`))`
`  (exists ((`x_2 e_2`))`
$\vdots$
`    (exists ((`x_n e_n`))`
$\Phi(x_1, x_2, \ldots, x_n)$`))))`.

A similar comment holds for the primitive `forall`.

2.7 Sets

the-set-of-all. Ontic allows for the construction of sets using the special primitive `the-set-of-all`. An expression of the form `(the-set-of-all` e`)` has exactly one possible value which is the set of all possible values of the expression e. For example, given an operator `a-number-greater-than` we can represent the set of all integers greater than a given integer x by the expression

`(the-set-of-all (a-number-greater-than` x`))`.

This expression has a single value which is the set of all integers no smaller than x. An expression of the form `(the-set-of-all` e`)` is similar to the thunk `(lambda ()` e`)` — both expressions have a single possible value that either is, or is a representation of, the set of all possible values of e. In many applications it seems to be natural to make a distinction between sets and thunks. In Ontic thunks are different objects from sets — a thunk is never equal to a set.

a-member-of and **a-subset-of**. Ontic provides two primitive nondeterministic operators for dealing with sets. If s is an expression that denotes a set, i.e., has a single possible value which is a set, then each element of the set denoted by s is a possible value of the expression `(a-member-of` s`)`. Similarly, each subset of the set denoted by s is a possible value of the expression `(a-subset-of` s`)`.

2.8 Lambda Expressions and Definitions

lambda. Ontic has first class lambda expressions. The expression `(lambda ((`x e`))` $b(x)$`)` has one possible value, namely the operator that takes a possible value x of the expression e and nondeterministically maps x to a possible value of $b(x)$.

definitions. Ontic also allows definitions. A definition has the form

`(define` *name* *e*`)`

where *name* is a symbol and *e* is an expression. For example, one can define a function that multiplies a number by 2 as follows.

```
(define double
  (lambda ((x (an-integer)))
    (+ x x)))
```

This definition introduces the symbol `double` as an abbreviation for the above λ-expression. Any λ-expression has a single possible value so the above definition is well formed. Recursive definitions are allowed with certain restrictions discussed below.

To avoid confusion, in any definition `(define` *f* *e*`)` the expression *e* must have a single possible value — the expression abbreviated by a defined symbol must be singleton. Since λ-expressions always have exactly one possible value, this constraint will always be satisfied when defining operators or thunks.

Definitions of the above form can be abbreviated as follows.

```
(define (double (x (an-integer)))
  (+ x x))
```

In λ-expressions with more than one argument the type of an argument can depend on the value of an earlier argument. For example consider the following.

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

```
(define an-integer-upper-bound
   (lambda ((s (a-set-of-integers)) (u (a-subset-of s)))
     (some-such-that y (a-member-of s)
        (is (member-of u) (an-integer-less-than-or-equal-to y)))))
```

In the above definition the type of `u` involves the first argument `s`. An expression of the form (an-integer-upper-bound `s u`) will have a value only when `s` is a set of integers, `u` is a member of `s`, and there exists an element of `s` which is at least as large as every element of `u`.[5] A type that depends on the value of an earlier argument is called a *dependent type*. The above definition has been given in the long form (with an explicit λ-expression) to emphasize the fact that dependent types can occur in arbitrary λ-expressions, not just in the parameters of a defined operator.

2.9 Currying

Currying. In Ontic, all multi-argument operators are actually abbreviations for Curried expressions built up from single-argument λ-expressions. Ontic λ-expressions of more than one argument are actually just abbreviations for nested λ-expressions each of which takes only one argument. For example, a λ-expression of the form `(lambda ((`x e`) (`y h`))` $b(x, y)$`)` is actually an abbreviation for `(lambda ((`x e`)) (lambda ((`y h`))` $b(x, y)$`))`. This representation of multi-argument operators by single argument operators is called *Currying*. An application of the form `(`f a b`)` is treated as an abbreviation for `((`f a`)` b`)`.

Currying simplifies both the formal semantics and the implementation of the Ontic system. For the most part the user can ignore Currying and just assume that operators can take more than one argument. However, Ontic does allow the user to write expressions of the form `(`f a`)` where f is a λ-expression of *two* arguments. In this case `(`f a`)` denotes an operator of one argument. The fact that all λ-expressions are actually Curried is also

[5] If `s` is the set of all integers and `u` is the set of all even integers then there is no possible value for `(an-integer-upper-bound s u)` even though `s` and `u` match the types of the operator `an-integer-upper-bound`.

important in understanding the use of the primitive `a-domain-member-of` as described in section 4.2.

2.10 Basic Semantics

This section discusses the meaning of Ontic expressions at a more detailed level than that given in previous sections. Each Ontic expression has a set of possible values. The set of possible values of an Ontic expression can be precisely defined by structural induction on expressions. This precise definition is just a more formal treatment of the informal discussion presented in the previous section. Because the formal semantics is largely determined by the informal discussion of the previous sections, a complete presentation of a formal denotational semantics is not given here. Instead, this section points out some of the highlights and subtleties of of the formal semantics.

Each Ontic expression has a set of possible values. There are six basic kinds of values — symbols, integers, cons cells, thunks, operators, and sets. Every Ontic value belongs to exactly one of these groups. Each kind of value either is, or is a representation of, a standard Mathematical object.

- We assume the reader is familiar with symbols and integers.
- A cons cell is any value that can be returned by the operator `cons` applied to two other values. A cons cell is a representation of a mathematical pair.
- A thunk is value which, when applied to no arguments, has a set of possible values. A thunk is a representation of a set of values — the set of possible values of applying the thunk to no arguments.
- An operator consists of two things — a domain set and a set of input/output pairs. The domain set is the set of values to which the operator can be applied. Each input/output pair specifies a possible output value for a given input value. A given input value can be associated with more than one possible output value. Also, there may be elements of the domain set which are not associated with any output

value. However, every input value in an input/output pair must be a member of the domain set.

Ontic operators of more than one argument are internally Curried (see section 2.9). This implies that internally every operator takes only a single argument. An expression of the form `(lambda ((x A) (y B)) E)` has a single possible value which is an operator. When this operator is applied to a possible value of `A` it returns an operator which can then be applied to a possible value of `B`.

- We assume the reader is familiar with sets — sets are one kind of Ontic value. A set carries exactly the same information as a thunk — thunks are just representations of sets. However, the distinction between thunks and sets seems to greatly improve the intuitive readability of Ontic expressions.

Let E be an Ontic expression and let ρ be a semantic variable interpretation, i.e., a map from variables to semantic values.[6] We let $\mathcal{V}(E,\ \rho)$ denote the set of possible values of the expression E under variable interpretation ρ. The formal definition of $\mathcal{V}(E,\ \rho)$ is by structural induction on the expression E. More precisely, we first define $\mathcal{V}(E,\ \rho)$ for the case where E is a numeral or a quoted symbol. In both of these cases $\mathcal{V}(E, \rho)$ is a singleton set. (Recall that $\mathcal{V}(E,\ \rho)$ is the set of possible values of E and that numerals and quoted symbols have exactly one possible value.) A variable always has a single possible value — the value assigned to that variable by the given semantic variable interpretation. In other words, if x is a variable then $\mathcal{V}(x,\ \rho)$ is the singleton set $\{\rho(x)\}$. We now define the possible values of other expressions by structural induction. We consider an expression E and assume that for any expression W smaller than E we have defined $\mathcal{V}(W,\ \rho)$ for all possible variable interpretations ρ. Given this assumption we can proceed to define $\mathcal{V}(E,\ \rho)$ for an arbitrary variable interpretation ρ. For example, $\mathcal{V}(\texttt{(either}\ S\ W\texttt{)},\ \rho)$ equals the union of the sets $\mathcal{V}(S,\ \rho)$ and $\mathcal{V}(W,\ \rho)$. For the most part the rest of the definition of $\mathcal{V}(E,\ \rho)$ is a straightforward formal treatment of the informal semantics given in the previous section.

[6] Unlike most treatments of semantics, we give no specification of a semantic domain. The variable interpretation ρ can be *any function* whose domain is the set of Ontic variables.

Most cases will be left as exercises for the reader. (We encourage readers to reread the previous and assign a formal denotational semantics to each kind of Ontic expression.)

One particularly interesting case is λ expressions. Because Ontic operators are internally Curried, we need only consider λ-expressions of one argument. We define $\mathcal{V}(\texttt{(lambda ((x T)) W)},\ \rho)$ to be the operator whose domain set is $\mathcal{V}(T,\ \rho)$ and whose input/output pairs are those pairs $<a,\ b>$ where a is a member of $\mathcal{V}(T,\ \rho)$ and b is a member of $\mathcal{V}(W,\ \rho[x := a])$ where $\rho[x := a]$ is the variable interpretation that is identical to ρ except that it maps x to a. The semantics of λ-expressions is related to the semantics of application. If $(S\ W)$ is an application expression then $\mathcal{V}(\texttt{(}S\ \ W\texttt{)},\ \rho)$ is the set of values b such that there exists an operator f in the set $\mathcal{V}(S,\ \rho)$ and a value a in $\mathcal{V}(W,\ \rho)$ such that the pair $<a,\ b>$ is an input/output pair of f.

2.11 Operator Spaces

This section extends the basic Ontic language with two additional kinds of expressions. These expressions do not extend the kinds of semantic values in the ontic language — every value of these new expressions is either a thunk or an operator. However, these new expressions make it more convenient to use Ontic expressions as types. The new primitive `a-function` is also closely related to the axiom of choice in Zermelo-Fraenkel set theory.

an-operator-from. Ontic includes expressions of the form

$$\texttt{(an-operator-from}\ d\ \texttt{to}\ r\texttt{)}.$$

Each possible value of the expression `(an-operator-from` d `to` r`)` is an operator f whose domain is the set of possible values of the expression d and such that for any domain value x, i.e., any possible value of d, every possible value of `(`$f\ \ x$`)` is also a possible value of r. For example, consider the operator `double` defined as follows.

```
(define (double (x (an-integer)))
  (+ x x))
```

The operator `double` is a possible value of `(an-operator-from (an-integer) to (an-integer))`. Consider the operator `an-integer-greater-than` which takes an integer and returns an integer x and returns an integer greater than x. The operator `an-integer-greater-than` is also a possible value of `(an-operator-from (an-integer) to (an-integer))`.

The primitive `an-operator-from` can take more than one domain type. For example, the operator `+` which takes two integers and returns an integer is a possible value of `(an-operator-from (an-integer) (an-integer) to (an-integer))`. In general, the possible values of an expression of the form `(an-operator-from` τ_1 ... τ_n `to` σ`)` are all the operators f whose domain sets are the sets of possible values of τ_1, ..., τ_n and such that for any possible values x_1, ..., x_n of τ_1, ..., τ_n respectively, we have that every possible value of `(`f x_1 ... x_n`)` is a possible value of σ. Because all Ontic operators are Curried, the expressions `(an-operator-from` τ_1 τ_2 `to` σ`)` is equivalent to the expression `(an-operator-from` τ_1 `to (an-operator-from` τ_2 `to` σ`))`.

The axiom of choice is incorporated into Ontic by including the primitive construct `a-choice-function-from`. The primitive `a-choice-function-from` is similar to the primitive `an-operator-from` except that it introduce bound variables to represent elements of the domain types and it always returns a function rather than an arbitrary operator. For example, the possible values of the expression

```
(a-choice-function-from (x (an-integer))
                        to (an-integer-greater-than x))
```

are all operators f such that for any integer x there is exactly one possible value of `(`f x`)` and that value is an integer greater than x. Note that the value f must be a function, i.e., for any element x of the domain type `(`f x`)` must have exactly one possible value. The function `successor` which takes an integer x and return $x + 1$ is a possible value of the above `a-choice-function-from` expression. A function such as `successor` is called a "choice function" because it selects a possible value of the expression `(an-integer-greater-than` x`)`. The operator `an-integer-greater-than` is *not* a possible value of the above `a-choice-function-from` expression be-

cause it is not a function — it has more than one possible output value for a given input value. For any expression of the form

(a-choice-function-from (x τ) to B[x])

if the ontic system can prove the formula

(forall ((x τ)) (there-exists B[x]))

then the ontic system will infer

(there-exists (a-choice-function-from (x τ) to B[x])).

This is the set-theoretic axiom of choice.

As with `an-operator-from`, the primitive `a-choice-function-from` can take more than one domain type. For example one can write the following.

```
(a-choice-function-from (x (an-integer)) (y (an-integer))
  to (an-integer-between x y))
```

All operators, including functions, are Curried. So the above expression is equivalent to the following.

```
(a-choice-function-from (x (an-integer))
   to (a-choice-function-from (y (an-integer))
        to (an-integer-between x y)))
```

2.12 Type Restrictions in λ-expressions

Where ever a bound variable is introduced, Ontic allows a "such-that" notation which allows a formula to be used to further restrict the possible values of the bound variable. For example, consider the following definitions of subtraction on the natural numbers.

```
(define (a-number-not-less-than (n (a-natural-number)))
  (either n (a-number-not-less-than (+ 1 n))))


(define (difference (x (a-natural-number))
                    (y (a-natural-number)
                       such-that
                       (is x (a-number-not-less-than y))))
   (some-such-that z (a-natural-number)
     (= x (+ y z))))
```

The above definition of the difference function is an abbreviation for the following.

```
(define difference
  (lambda ((x (a-natural-number))
           (y (a-natural-number)
              such-that
              (is x (a-number-not-less-than y))))
    (some-such-that z (a-natural-number)
      (= x (+ y z)))))
```

Such-that restrictions can also be used in quantified formulas. For example, the following formula is true (under any standard definition of the terms involved).

```
(forall ((s1 (a-set-of-integers))
         (s2 (a-set-of-integers)
             such-that
             (there-exists (a-member-of (intersection s1 s2)))))
  (exists ((y (+ (a-member-of s1) (a-member-of s2))))
    (is y (an-even-number))))
```

Such-that restrictions can also be used with the operator `the-set-of-all`.

For example, one might write the following.

```
(the-set-of-all x (an-integer)
  such-that (exists ((y (an-integer)))
              (= x (+ y y))))
```

Most of the axioms of set theory are incorporated into principles for reasoning about the primitives that have been described in this section. In principle much of mathematics could be done with just these primitives. However, the primitives described above do not allow for recursive definitions. Recursive definitions greatly reduce the complexity of a variety definitions and proofs. The next section discusses the semantics of recursion in Ontic.

3 Recursion

Ontic allows recursive definitions — definitions in which the symbol being defined appears in the body of the definition. Recursive definitions are restricted to the case where a symbol is either defined to be a thunk or operator. For recursively defined operators the symbol being defined can not appear in the type restriction on the arguments of the operator.[7] The recursive definition must also satisfy a semantic fixed point condition. Any "ordinary" recursive definition will satisfy the fixed point criterion and for the most part the fixed point restriction can be ignored. The fixed point restriction on recursive definitions is discussed below.

Consider the following recursive definition of the natural numbers.

```
(define a-natural-number
  (lambda ()
    (either 0 (+ 1 (a-natural-number)))))
```

The above definition defines a thunk, `a-natural-number` which, when ap-

[7]This restriction simplifies the mechanisms for reasoning about recursive definitions.

plied, can return any natural number. This thunk can also be defined using the following syntax.

```
(define (a-natural-number)
  (either 0 (+ 1 (a-natural-number))))
```

An exponentiation function can be defined as follows.

```
(define (expt (x (an-integer)) (y (a-natural-number)))
  (if (= y 0)
      1
      (* x (expt x (- y 1)))))
```

As stated above, the symbol being defined, `expt` in this case, can not appear in the type restrictions on the arguments. In this example the type restrictions are `(an-integer)` and `(a-natural-number)`.

3.1 The Fixed Point Restriction on Recursive Definitions

The vast majority of recursive definitions encountered in practice satisfy the fixed point criterion. For this reason casual users should simply write recursive definitions and assume they will satisfy the fixed point criterion. If a definition does not satisfy the fixed point criterion the ontic system will print out an error message and refuse to accept the definition. If this happens there are simple techniques for modifying the definition so that it becomes acceptable to the Ontic system. These modification techniques are described in section **??**. Such modifications are usually quite simple, such as declaring the output type of a defined operator, and do not require an understanding of the fixed point restriction. For the sake of completeness, however, the fixed point restriction is discussed in some detail here.

For any recursive definition, whether or not it satisfies the fixed point restriction, Ontic assigns a well defined meaning to the defined symbol. The

meaning is a transfinite limit over all ordinals. In this section we are not concerned with the details of this method of assigning meaning — we simply assume that Ontic has *some* way of assigning such meanings.[8] The fixed point restriction on recursive definitions states that *Ontic must be able to automatically prove* that the transfinite limit meaning is a fixed point of the recursive definition. The concept of a fixed point is best understood in terms of an example. Consider the following definition.

```
(define (an-integer-list)
  (either 'nil (cons (an-integer) (an-integer-list))))
```

This definition satisfies the fixed point restriction — Ontic can verify that the transfinite limit meaning derived from this definition is a fixed point of the definition. The transfinite limit meaning is such that `an-integer-list` is a thunk which, when applied, can return any finite list of integers. The meaning of the thunk is a fixed point of the definition in the sense that the call `(an-integer-list)` has exactly the same set of possible values as the body of the above definition, i.e., the same of possible values as

```
(either 'nil (cons (an-integer) (an-integer-list))).
```

Consider an arbitrary recursive thunk definition.

```
(define (a-foo)
    C[a-foo])
```

In the above definition the body `C[a-foo]` is an expression that involves `a-foo`. The meaning of `a-foo` should be a fixed point of this definition. This means that the expression `(a-foo)` should have the same set of possible values as `C[a-foo]`.

There are recursive definitions which do not have fixed points. For example, consider the following.

```
(define (a-paradoxical-object)
```

[8]The transfinite limit meaning is described in more detail in section 3.2.

```
(if (there-exists (a-paradoxical-object))
    (fail)
    1))
```

The transfinite limit meaning assigned by Ontic is such that `a-paradoxical-operator` is a thunk which, when applied, returns the single value 1. But this meaning is not a fixed point of the above definition. The expression `(a-paradoxical-object)` has the single value 1 while the expression

```
(if (there-exists (a-paradoxical-object)) (fail) 1)
```

has no possible values. This definition has no fixed point, no matter how we interpret the meaning of `a-paradoxical-object` the expression `(a-paradoxical-object)` must have a different set of possible values than the body of the definition.

There are two ways a definition can fail to satisfy the fixed point restriction. First, the transfinite limit meaning of the defined term may not, in fact, be a fixed point of the definition. In this case there is no alternative but to either modify the definition or give up. A second way a definition can fail to satisfy the fixed point condition is that the transfinite limit meaning is a fixed point but the Ontic system is not powerful enough to prove this fact automatically. In this case the definition can usually be made to satisfy the fixed point criterion by adding declarations to the definition that Ontic can use in its attempt to prove that the meaning is a fixed point. Such declarations are described in section **??**.

It is worth noting that a given recursive definition can have more than one fixed point. As an example consider the following definition of the natural numbers.

```
(define (a-foo-number)
  (either 0 (+ 1 (a-foo-number))))
```

Given this definition Ontic interprets `a-foo-number` as the thunk which, when applied, returns any nonnegative integer. Suppose, however, that for some reason we wished to interpret `a-natural-number` as the the thunk which, when applied, return any integer. This latter interpretation is also a

fixed point of the above definition — the expression `(a-foo-number)` would have the same set of possible values as

```
(either 0 (+ 1 (a-foo-number))).
```

For any definition satisfying the fixed point restriction, the fixed point assigned by the transfinite limit meaning is always the *least* fixed point of the definition. If a recursive definition of a thunk `a-foo` is accepted by the Ontic system then the meaning of that thunk is the least fixed point of the definition, i.e., the set of possible values of `(a-foo)` is the smallest set possible given that `a-foo` must denote a fixed point of the definition. Another way of saying this is that only those possible values which are *forced* by the definition are actually included as possible values of `(a-foo)`.

All of the above aspects of recursive thunk definitions apply to recursive definitions of other operators. Consider the following definition of `append` on integer lists.

```
(define (append (l1 (an-integer-list)) (l2 (an-integer-list)))
  (if (= l1 'nil)
      l1
      (cons (car l1) (append (cdr l1) l2))))
```

Given this definition, Ontic interprets `append` as the normal append operations on lists. The operator defined in a recursive definition can not be used in the type constraints on the parameters to that operator. Thus the domain of the operator is directly given and independent of the transfinite limit process for constructing the meaning of the operator. The standard meaning of the append operation on lists is a fixed point of the above definition in the sense that for any integer lists x and y we have that `(append` x y`)` is equivalent to

```
(if (= x 'nil)
    x
    (cons (car x) (append (cdr x) y)))
```

Recursive operator definitions can fail to have any fixed point, or have

more than one fixed point. As an example of an operator with more than one fixed point consider the following.

```
(define (an-integer-greater-than (x (an-integer)))
  (either (+ 1 x)
          (+ 1 (an-integer-greater-than x))))
```

Given this definition, Ontic interprets `an-integer-greater-than` to be the operator such that for any integer x, the possible values of `(an-integer-greater-than` x`)` are precisely the integers greater than x. However, if we interpret `an-integer-greater-than` as the operator which takes an integer and then returns *any* integer whatsoever, then we also get a fixed point of the above definition. As with thunks, the transfinite limit semantics constructs the least possible fixed point. The semantic value of a recursively defined operator f is such that for any values x_1, x_2, ..., x_n we have the set of possible values of `(`$f\ x_1\ x_2\ \ldots\ x_n$`)` is the least set possible given that f must be a fixed point of the definition.

3.2 Transfinite Limit Semantics for Recursion

This section provides a more detailed discussion of the process by which semantic meaning is assigned to recursively defined thunks and operators. First we consider thunks. Consider an arbitrary recursive definition of the following form.

```
(define a-foo
  (lambda ()
    C[a-foo]))
```

The meaning of a recursively defined thunk is a transfinite limit of a series of approximations or "versions" of the thunk. Given the above definition, one can write the expression `(version a-foo` α`)` where α is an ordinal.[9]

[9]Expressions of this form are used internally in reasoning about recursively defined thunks and operators. Such version expressions are not intended to be used by Ontic users. Ordinals are used internally in reasoning about recursively defined terms but are

The ordinals are totally ordered — given any two ordinals one is less than than the other. Furthermore, any subset of the ordinals contains a least member. Ordinals are like natural numbers. There is a least ordinal, which is normally written as 0, a next least ordinal written as 1, a next least written as 2 and so on. There is also a least ordinal which can not be reached in this way, written as ω — ω is the least ordinal larger than all of 0, 1, 2, ... Of course there are ordinals larger than ω. The expression `(version a-foo 0)` denotes the thunk which, when applied, does not have any possible values. The expression `(version a-foo 1)` is the thunk whose body is given by the recursive definition of `a-foo` where each recursive call to the thunk `a-foo` is replaced by a call to the thunk `(version a-foo 0)`. The expression `(version a-foo 2)` is the thunk given by the definition where recursive calls invoke `(version a-foo 1)`. Again consider an arbitrary recursive definition.

```
(define a-foo
  (lambda ()
    C[a-foo]))
```

Given this definition, the expression `(version a-foo` α`)` is equivalent to

```
(lambda ()
  (let ((β (an-ordinal-less-than α)))
    C[(version a-foo β)])))
```

As an example, consider the following definition of a thunk that returns a list of integers.

```
(define (an-integer-list)
  (either 'nil (cons (an-integer) (an-integer-list))))
```

The expression `(version an-integer-list 0)` denotes the thunk which, when applied, has no value. The expression `(version an-integer-list`

not provided as primitives for the user. To use ordinals in proofs the user can define ordinals in the same way as any other mathematical concept.

`1)` denotes the thunk which, when applied, has the single value `nil`. The expression `(version an-integer-list 2)` denotes the thunk which, when applied, can return either `nil` or a list of one integer. If n is a finite ordinal then `(version an-integer-list` n`)` denotes the thunk which, when applied, can return any integer list of length less than n. If α is any infinite ordinal then `(version an-integer-list` α`)` denotes the thunk which, when applied, can return any finite list of integers. Note that if α and β are any two infinite ordinals then `(version an-integer-list` α`)` has the same value as `(version an-integer-list` β`)`. This is because `(version an-integer-list` ω`)` is a fixed point of the definition. Most recursive definitions reach a fixed point at ω.

As an example of a recursive definition which reaches a fixed point at an ordinal larger than ω consider the definition of a Borel set. A Borel set is a subset of the real numbers. Any open interval is a Borel set. However, the Borel sets are closed under countable intersections and countable unions. The set of Borel sets is the least family of sets that satisfy these two conditions. The set of Borel sets is defined recursively below. The versions of this recursive definition reach a fixed point at the first uncountable ordinal.

```
(define (a-borel-set)
  (either (open-interval (a-real-number) (a-real-number))
          (family-union (a-countable-set-of a-borel-set))
          (family-intersection (a-countable-set-of a-borel-set))))
```

Not all recursive definitions reach fixed points. Consider the following definition of a "well founded set".

```
(define (a-pure-set)
  (a-subset-of (the-set-of-all (a-pure-set))))
```

Given this definition we have that `(version a-pure-set 1)` is a thunk which, when applied, returns the empty set. The expression `(version a-pure-set 2)` is a thunk which, when applied, returns either the empty set or the set containing the empty set. `(version a-pure-set 3)` is a thunk

which, when applied, has four possible values. For any finite ordinal n we have that `(version a-pure-set` n`)` is a thunk which, when applied, has a finite set of possible values — the set of all hereditarily finite sets of rank n that can be built from the empty set. `(version a-pure-set` ω`)` is a thunk which, when applied, can return any hereditarily finite set built from the empty set. `(version a-pure-set` $\omega + 1$`)` is a thunk which, when applied, can return any subset of hereditarily finite sets — including infinite subsets. Because any set has more subsets than members, there can not be any fixed point of the above definition — for any ordinal α we have that `(version a-pure-set` $\alpha + 1$`)` is a thunk which, when applied, has more possible values than `(version a-pure-set` α`)`.

The meaning of a recursively defined thunk is taken to be the transfinite limit of the versions the thunk. More precisely, given a recursive definition of `a-foo`, the thunk `a-foo` can be defined in terms of the versions of `a-foo` as follows.

```
(define a-foo
  (lambda ()
    (let ((α (an-ordinal)))
      (let ((v (version a-foo α)))
        (v)))))
```

To compute a value for the call `(a-foo)` one first selects an ordinal `n`. One then constructs the thunk `(version a-foo n)` and calls this thunk to get a value. For example, given the above definition of `an-integer-list`, the symbol `an-integer-list` denotes a single well defined thunk which, when applied, can return an integer list of any finite length. This is a fixed point of the definition. Given the definition of a Borel set, the expression `(a-borel-set)` returns any one of a very large collection of possible subsets of the real numbers. The definition of a Borel set does reach a fixed point and the recursive definition is acceptable to Ontic. Any set built purely from the empty set is a possible value of `(a-pure-set)`. Unfortunately, this well defined meaning for the expression `(a-pure-set)` is not a fixed point of the definition. The class denoted by `(the-set-of-all (a-pure-set))` is not a possible value of `(a-pure-set)`. This fact is closely related to the

fact that the collection of all sets does not contain itself — the collection of all sets is a "class" rather than a set. The collection of all pure sets is not a set. So there is a possible value of `(a-subset-of (the-set-of-all (a-pure-set)))` which is not a possible value of `(a-pure-set)`.

Recursive operator definitions are treated in the same manner as recursive thunk definitions. For example consider the following definition of the append function on integer lists.

```
(define (append (l1 (an-integer-list)) (l2 (an-integer-list)))
  (if (= l1 'nil)
      l1
      (cons (car l1) (append (cdr l1) l2))))
```

Given this recursive definition, `(version append` α`)` is equivalent to the following.

```
(lambda ((l1 (an-integer-list)) (l2 (an-integer-list)))
  (let ((β (an-ordinal-less-than α)))
    (if (= l1 'nil)
        l1
        (cons (car l1) ((version append β) (cdr l1) l2)))))
```

For any ordinal α `(version append` α`)` denotes an "approximation" to the append function. `(version append` α`)` is the operator whose definition is given by the recursive definition of `append` except that recursive calls to `append` use the version of `append` at a smaller ordinal than α. The larger the ordinal α the better the approximation. Let n be a finite ordinal. If we apply the operator `(version append` n`)` to the lists `l1` and `l2` there are two possible results. If the length of `l1` is less than n then the application will return the expected append of `l1` and `l2`. If the length of `l1` is greater than or equal to n then the application will fail to have a value. If α is any infinite ordinal then `(version append` α`)` denotes the ordinary append function. The recursive definition of `append` reaches a fixed point at ω.

The recursive definition of the append function is treated as an abbrevi-

ation for the following.

```
(define append
   (lambda ((l1 (an-integer-list)) (l2 (an-integer-list)))
     (let ((α (an-ordinal)))
       (let ((v (version append α)))
         (v l1 l2)))))
```

Note that the symbol `append`, as defined above, denotes a single well defined operator. Furthermore, an application of this operator to two integer lists has exactly one possible output value, namely the expected append of those two lists. For any version of `append` used to compute the output, either no output is produced or the output is the expected value.

4 Large Thunks and Other Features

There are a variety of features of Ontic which simplify definitions and theorems. The first such feature described in this section are the large primitive thunks. The large primitive thunks allow one to talk about "any set" or "any operator". These large thunks are formulated carefully to avoid set-theoretic paradoxes. This makes some aspects of these thunks rather subtle. A second convenient feature is type checking. The Ontic system checks that in every application the operator is being applied to values of the correct type. This type checking process reduces the time required to write definitions and proofs by identifying ill-formed expressions. A fourth convenient feature is the ability to define structures similar to the structures used in programming languages such as Scheme or Common Lisp. The final convenient feature discussed in this section is type coercion. Type coercion provides a form of object oriented behavior analogous to that found in the Common Lisp CLOS system or in C++.

4.1 Large Primitive Thunks

Every Ontic value is either a symbol, an integer, a cons cell, a thunk, an operator, or a set. The thunks `a-symbol` and `an-integer` were discussed in section 2. Ontic also provides the primitive thunks `a-cons-cell`, `a-thunk`, `an-operator`, and `a-set`. These six primitive thunks correspond to the six kinds of Ontic values. The last four primitive thunks will be called *large primitive thunks.*

The semantics of large primitive thunks is a little tricky. The basic problem is that the set of output values of a large primitive thunk is extremely large. Intuitively we would like the set of possible values of `(a-set)` to include all sets. It should at least include all sets that arise in normal mathematics. A similar observation holds for the expressions `(a-cons-cell)`, `(a-thunk)` and `(an-operator)`. Unfortunately, one can prove that there is no set of all sets. In particular, the set denoted by `(the-set-of-all (a-set))` can not be a possible value of `(a-set)`. `(the-set-of-all (a-set))` is a set, but is not a member of itself. There must be sets which are not possible values of `(a-set)` and hence there must be sets which are not members of `(the-set-of-all (a-set))`.

Fortunately, the large primitive thunks can be made large enough to include all the objects that arise in normal mathematics. In particular, the large primitive thunks cover all objects that can be named without using the large primitive thunks. The `(the-set-of-all (a-set))` is not a possible value of `(a-set)` because the large primitive thunk `a-set` is used it its definition. An expression that does not contain a large primitive thunk will be called a *predicative* expression. Expressions that contain large primitive thunks will be called *impredicative. Any Ontic value that is a possible value of a predicative expression is also a possible output value of one of the six primitive thunks.*

All "normal" mathematics can be done with purely predicative expressions. For example, the expression `(the-set-of-all (an-integer))` which denotes the set of all integers, is predicative. Similarly, the set of all subsets of the integers can also be defined by a predicative expression as follows.

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

Since normal mathematics can be done with predicative expressions, one might think that the large primitive thunks are not needed at all. However, large primitive thunks are natural for expressing certain concepts. For example, consider the following definitions.

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


(define (a-superset-of (s (a-set)))
  (some-such-that s2 (a-set)
    (is s (a-subset-of s2))))


(define (a-thing)
    (either (a-symbol)
            (an-integer)
            (a-cons-cell)
            (a-thunk)
            (an-operator)
            (a-set)))


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


(define (a-list-member-of (l (a-list)))
```

```
  (when (not (= l 'nil))
    (either (car l)
            (a-list-member-of (cdr l)))))


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

The above definitions are extremely general.[10]

4.2 Type Checking

a-domain-member-of. An Ontic operator consists of two things — a domain set and a set of input/output pairs. All applications of the form (*f a*) must be well typed in the sense that for all possible values of *f* and *a* the value of *f* is an operator and the value of *a* is an element of the domain of that operator. There are certain cases where one might like to apply operators to objects that are not necessarily elements of the domain of the operator. For example, consider the `map` operator defined in the previous section. One might like to define the `map` operation as follows.

```
(define (map (f (an-operator)) (l (a-list)))
  (if (= l 'nil)
      'nil
      (cons (f (car l)) (map f (cdr l)))))
```

[10]The primitive `a-domain-member-of` is needed in the definition of `map` to ensure that the expression `(f (car l))` is well typed. The primitive `a-domain-member-of` is described in the next section.

However, in this definition the application `(f (car l))` is not well typed — the value of `(car l)` is not guaranteed to be in the domain of `f`. However, by rewriting this definition as follows, the application becomes well typed.

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

Alternatively, one can guarantee that the application is well typed by using the following definition.

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

Intuitively, the application `(f (car l))` will be well typed in any execution of this procedure which reaches the application. This example exploits the fact that in the Ontic system well-typedness is context sensitive, i.e., whether or not a given expression is well typed depends on the context in which that expression appears. In the above definition, the occurrence of `(f (car l))` is well typed because it appears in a place which can only be "reached" when the formula `(is (car l) (a-domain-member f))` is true. Similarly, the occurrence of the expression `(car l)` is well typed because it can only be reached when the formula `(= l 'nil)` is false.[11]

[11] In the Ontic system one can not take the `car` or `cdr` of the symbol `nil`.

4.3 Structures

defstruct. Ontic allows for structure definitions analogous to those in common Lisp. For example, suppose we wish to define the concept of a point in the $x-y$ plain. We can do this using the following structure definition (here we assume the concept of a real number).

```
(defstruct an-xy-point
  (x-coordinate (a-real-number))
  (y-coordinate (a-real-number)))
```

This structure definition is an abbreviation for the following series of definitions.

```
(define (make-an-xy-point (x (a-real-number))
                          (y (a-real-number)))
  (list 'an-xy-point x y))


(define (an-xy-point)
  (let ((x (a-real-number))
        (y (a-real-number)))
    (make-an-xy-point x y)))


(define (x-coordinate (p (an-xy-point)))
  (car (cdr p)))


(define (y-coordinate (p (an-xy-point)))
  (car (cdr (cdr p))))
```

Structure definitions can involve dependent types. For example, we can define a directed graph as follows.

```
(defstruct a-directed-graph
  (the-node-set (a-subset-of
                  (the-set-of-all (a-symbol))))
  (the-arc-set (a-subset-of
                 (the-set-of-all
                   (cons (a-member-of the-node-set)
                         (a-member-of the-node-set))))))
```

In the above definition the type of the second slot, `the-arc-set`, depends on the object in the first slot. Every arc in the arc set must be an arc between two nodes that are elements of the node set. In general the slot names in a structure definition are treated as variables and slot names can appear in the types of later slots.

Structure definitions can also include a "such-that" clause. For example, we can define a rectangle to be a structure consisting of four x-y points which are the corners of the rectangle.

```
(defstruct a-rectangle
  (lower-left (an-xy-point))
  (lower-right (an-xy-point))
  (upper-left (an-xy-point))
  (upper-right (an-xy-point))
  such-that
  (and (= (y-coordinate lower-left)
          (y-coordinate lower-right))
       (= (y-coordinate upper-left)
          (y-coordinate upper-right))
       (= (x-coordinate lower-left)
          (x-coordinate upper-left))
       (= (x-coordinate lower-right)
          (x-coordinate upper-right))))
```

As another example, we can define a group to be a domain, an identity element, an inverse operation and a group operation satisfying certain con-

ditions.

```
(define (a-monadic-function-on (d (a-set)))
  (lambda-fun ((x (an-element-of d)))
    (an-element-of d)))


(define (a-binary-function-on (d (a-set)))
  (lambda-fun ((x (an-element-of d)) (y (an-element-of d)))
    (an-element-of d)))


(defstruct (a-group)
  (domain (a-set))
  (ident (an-element-of domain))
  (inverse (a-monadic-function-on domain))
  (operator (a-binary-function-on domain))
  such-that
  (and (forall ((x (an-element-of domain))
                (y (an-element-of domain))
                (z (an-element-of domain)))
         (= (operator (operator x y) z)
            (operator x (operator y z))))
       (forall ((x (an-element-of domain)))
         (and (= (operator x ident) x)
              (= (operator x (inverse x)) ident)))))
```

Structure definitions can also be recursive. For example, one might define a data structure for representing information about employees as follows.

```
(defstruct an-employee-record
  (age (a-natural-number))
  (salary (a-natural-number))
  (supervisor (either 'none (an-employee-record))))
```

In general, a structure definition of the form

```
(defstruct  name
  (slot-1  type-1)
  ⋮
  (slot-n  type-n)
  such-that
  Φ)
```

is an abbreviation for the following:

```
(define (name)
  (let ((slot-1  type-1)
        ⋮
        (slot-n  type-n))
    (when Φ
      (list 'name  slot-1 ··· slot-n))))


(define (make-name (slot-1  type-1)  ··· (slot-n  type-n))
  (when Φ
    (list 'name  slot-1 ··· slot-n)))


(define (slot-1 (x  (name)))
  (car (cdr x)))


   ⋮

(define (slot-n (x  (name)))
  (car (cdr ... (cdr x))))
```

4.4 Type Coercion

defcoercion and **def-o-piece**. A coercion function maps objects of a variety of different types into a standard type. For example, consider directed graphs, groups, and lists as defined above. An object of any of these types can be viewed as a set. We can define a `coerce-to-set` operation which maps objects of these types to sets. The function `coerce-to-set` is declared to be a coercion function using the primitive `defcoercion`. The value of the coercion function is defined for various types using the Ontic primitive `def-o-piece`.

```
(defcoercion coerce-to-set
  (a-set))


(def-o-piece (coerce-to-set (g (a-directed-graph)))
  (the-node-set g))


(def-o-piece (coerce-to-set (g (a-group)))
  (domain g))


(def-o-piece (coerce-to-set (l (a-list)))
  (if (= l 'nil)
      the-empty-set
      (insert (car l) (coerce-to-set (cdr l)))))
```

The primitive `defcoercion` is used to introduce the coercion function and declare its output type. The `defcoercion` form also defines the coercion function to be the identity operation on the given output type. For example, the following definition is implicitly present in the above declaration of `coerce-to-set`.

```
(def-o-piece (coerce-to-set (s (a-set)))
  s)
```

The primitive `def-o-piece` is used to define the value of the coercion function on various types. For each `def-o-piece` form the Ontic system checks to make sure that the new domain type, e.g., the type `(a-group)` above, does not overlap with any of the previous domain types. This ensures that the various domain types are pairwise disjoint and that only one definition can be used in any single application of the coercion function. Ontic also checks each definition introduced with `def-o-piece` to ensure that the value generated is an instance of the output type declared for the coercion function.

If f is a coercion function, i.e., a function introduced using `defcoercion`, then the domain of f is open-ended — the objects to which f can be applied can always be expanded. If f is a coercion then the possible values of the expression `(a-domain-member-of` f`)` are not well defined. At any point in time there are certain objects, i.e., those that can be coerced using f, that are definitely possible values of `(a-domain-member-of` f`)`. However, one can never be sure that a particular object is not in the domain of f. The domain of a coercion function is usually a very useful type. For example, consider the following definitions.

```
(define (a-virtual-set)
  (a-domain-member coerce-to-set))


(define a-vmember-of (s (a-virtual-set))
  (a-member-of (coerce-to-set s)))


(define (a-vsubset-of (s (a-virtual-set)))
  (a-subset-of (coerce-to-set s)))


(define (set-difference (s1 (a-virtual-set))
                        (s2 (a-virtual-set)))
  (the-set-of-all
    (some-such-that x (a-vmember-of s1)
```

```
(not (is x (a-vmember-of s2))))))
```

5 Proofs

Most traditional proof systems are defined by rules of inference — each statement must be derived from previous lines using one of the inference rules. The Ontic system does not work this way. In Ontic each line must be an "obvious consequence" of the preceding lines. The notion of obviousness is best viewed as a semantic rather than a syntactic notion. Ontic uses a complex automated inference procedure to determine whether a given statement is obvious. The inference mechanisms used in determining if a given statement is obvious at a given point in a proof are not described here. Readers interested in the inference mechanisms should consult technical papers on the inference mechanisms such as [McAllester *et al.*, 1989], [McAllester, 1990], [McAllester, 1991b], [McAllester, 1991a].

Ontic is a knowledge intensive system. Ontic maintains a "lemma library" containing a set of definitions and theorems. Ontic automatically accesses definitions and theorems from this library when it determines if a given statement is obvious. In this way the notion of obviousness can change over time as more theorems are added to the lemma library. Proofs never make explicit reference to definitions or theorems. If a theorem is in the lemma library then any particular instance of that theorem is obvious to the Ontic system. Usually, however, the obviousness of a statement requires a combination of many definitions and theorems and a "subconscious" inference process that puts the definitions and theorems together in verifying a statement.

The next section describes the basic structure of ontic proofs and the three basic proof constructs *show*, *suppose*, and *let-be*. Three later sections describe other general purpose proof constructs, specific methods for proving specific kinds of formulas, and induction proofs.

5.1 Show, Suppose, and Let-be

A proof is a kind of expression that is read by the system. The result of reading a proof is one or more lemmas which are added to the lemma library. There are three basic kinds of proof expressions — show expressions, suppose expressions, and let-be expressions. The simplest proof expressions are show expressions of the form `(show` Φ`)` where Φ is a formula. A series of show expressions can often be given which culminate in some desired result. For example, consider a group `g` with group operation `op`. We can define an identity element of `g` to be an element `id` such that for any other element `x` we have that `(op id x)` equals `(op x id)` equals `x`. Suppose that we wish to show that a group has at most one identity element. In particular, suppose that we are considering two identity elements `id1` and `id2`. We wish to show `(= id1 id2)`. This can be done with the following series of show expressions.

```
(show (= (op id1 id2) id1))
(show (= (op id1 id2) id2))
(show (= id1 id2))
```

This is a simple "Socratic" proof — it is a series of show expressions leading to a desired result. Reading the three show assertions given above results in the addition of three lemmas to the current context (the notion of context is described in the next section). It might be more desirable to prevent the first two lemmas from being added since they are really just steps in the proof of the third expression. To express the fact that the first two expressions are just steps in the proof of the third we can place the first two equations inside the third show expression resulting in the following proof.

```
(show (= id1 id2)
   (show (= (op id1 id2) id1))
   (show (= (op id1 id2) id2)))
```

This proof only adds a single lemma to the current context. In general, a show expression either fails or generates exactly one lemma — the for-

mula that is shown. Any number of proofs can be placed inside the show expressions. The proofs inside a show expression are called the body of that expression.

Let-be proofs are used to prove universally quantified formulas. In the above example we are really interested in proving that any group has at most one identity element. This result is actually more general than group theory. It applies to any structure, like a group, that contains a set of elements and a binary operation on those elements. We might call such structures binary operator structures and define them in ontic as follows.

```
(defstruct a-binary-operator-structure
  (domain (a-set))
  (operator (a-function-from (a-member-of domain)
                             (a-member-of domain)
                             to (a-member-of domain))))
```

For any binary operator structure we can define the concept of an identity element.

```
(define (identity-of (m (a-binary-operator-structure)))
  (some-such-that id (a-member-of (domain m))
    (forall ((x (a-member-of (domain m))))
      (and (= ((operator m) x id) x)
           (= ((operator m) id x) x)))))
```

We now wish to prove that for any binary operator structure there is at most one identity element. This can be done with the following proof.

```
(let-be ((m (a-binary-operator-structure)))
  (show (at-most-one (identity-of m))
    (suppose (there-exists (identity-of m))
      (let-be ((x (identity-of m))
               (y (identity-of m)))
        (show (= x y)
```

```
        (show (= ((operator m) x y) x))
        (show (= ((operator m) x y) y)))))))
```

This proof generates the following single lemma.

```
(forall ((m (a-binary-operator-structure)))
  (at-most-one (identity-of m)))
```

In general, a let-be expression has the following form.

```
(let-be ((x_1 τ_1) ... (x_n τ_n))
  <sub-proof>
  ⋮
  <sub-proof>)
```

The sub-proofs are called the body of the expression. Each sub-proof in the body generates one or more lemmas. The lemmas generated subproofs usually contain one or more of the bound variables of the let-be expression, e.g., one or more of the x_i's in the above expression. A let-be expression generates a lemma for each lemma generated by a subproof. If a subproof generates $\Phi[y_1, \ldots, y_k]$, where $y_1, \ldots, y_k$ are bound variables of the let-be expression, then the let-be expression generates a lemma of the form `(forall ((`y_1 τ_1`) ... (`y_k τ_k`))` $\Phi[y_1, \ldots, y_k]$). The variables $y_1, \ldots, y_k$ can be any subset of the bound variables of the let-be expression.

A let-be expression can only be used to introduce objects that are known to exist. For example suppose we define an "impossible number" to be a number that is both even and odd.

```
(define (an-impossible-number)
  (both (an-even-number) (an-odd-number)))
```

If we then try to evaluate `(let-be ((x (an-impossible-number))) ...)` the system will generate an error stating that it is unable to prove

`(there-exists (an-impossible-number))`. It does not really matter if the type involved actually exists — to use a let-be expression the system must be able to *prove* that objects of the given type exist. In the above proof about binary operator structures the system is able to prove that binary operator structures exist. However, a binary operator structure need not contain an identity element so before we can evaluate a let-be expression that introduces an identity element we need to first suppose that some identity element exists. In general, a suppose proof expression has the following form.

```
(suppose Φ
  <subproof>
  ⋮
  <subproof>)
```

If Ψ is a lemma generated by one of the subproofs in the body of the suppose expression, then the suppose expression generates the lemma `(implies Φ Ψ)`.

Ontic proofs are Socratic in the sense that for any show expression of the form `(show Φ <subproof> ... <subproof>)` the goal formula Φ must be an obvious consequent of the set of lemmas generated by the subproofs, i.e., the goal formula must be obvious in a context where the lemmas generated by the subproofs are included in the lemma library. It is useful to study the structure of the above proof that any operator structure has at most one identity element. Consider the following proof fragment.

```
(show (at-most-one (identity-of m))
  (suppose (there-exists (identity-of m))
    (let-be ((x (identity-of m))
             (y (identity-of m)))
      (show (= x y)
        ...))))
```

The body of the outer show expression consists of a single subproof. This

subproof generates the following lemma.

```
(implies (there-exists (identity-of m))
         (forall ((x (identity-of m))
                  (y (identity-of m)))
           (= x y)))
```

When this lemma is included in the lemma library Ontic is able to prove `(at-most-one (identity-of m))`. The relation between a goal and the subproofs used to show that goal is "semantic" in the sense that the facts proven in the subproofs must "obviously imply" the goal. The notion of obviousness is determined by Ontic's internal theorem proving mechanisms.

5.2 Suppose-For-Refutation

Ontic allows for proof by contradiction. Proofs by contradiction are also known as refutations. In a refutation proof one supposes the negation of the formula one is trying to prove. Ontic provide a primitive proof constructor called `suppose-for-refutation`. `Suppose-for-refutation` has the same syntax as `suppose` but behaves somewhat differently. Consider a proof expression of the following form.

```
(suppose-for-refutation Φ
  <subproof>
  ⋮
  <subproof>)
```

If Ontic can derive a contradiction from the supposition Φ and the lemmas generated by the subproofs then the above expression generates the theorem `(not Φ)`. Otherwise the above proof expression fails. A proof that there is no bijection between a set and its power set might start as follows.

```
(let-be ((s (a-set)))
```

```
(suppose-for-refutation
        (there-exists (a-bijection-between s (power-set s)))
  ...))
```

5.3 Case Analysis

Each proof is read in a context. A context consists of a current set of definitions, known lemmas, suppositions, and a current goal formula. The goal component of a context is important in proofs that involve case analysis. For example, consider the following definition of the mod2 function on integers.

```
(define (mod2 (n (an-integer)))
  (if (is n (an-even-integer))
      0
      1))
```

Now suppose that we wish to prove that `(is (* n (mod2 n)) (either 0 n))`. This can be done with the following proof.

```
(let-be ((n (an-integer)))
  (show (is (* n (mod2 n)) (either 0 n))
    (suppose (is n (an-even-integer)))
    (suppose (not (is n (an-even-integer))))))
```

Under the supposition that `n` is an even integer Ontic finds the goal formula obvious — under this supposition `(mod2 n)` is 0 so `(* n (mod2 n))` is also 0. The first suppose expression generates the implication stating that the supposition implies the given goal formula. Similarly the second suppose expression generates an implication that stating that the second supposition implies the given goal formula. Given these two implications the truth of the goal is obvious.[12]

[12]The current Ontic implementation does a certain amount of case analysis automat-

Consider an arbitrary suppose expression `(suppose` Φ `<subproof> ... <subproof>)`. If this show expression is read in a context with a goal formula Ψ, and if Ontic is able to determine that Ψ is an obvious consequence of the supposition Φ and the lemmas generated by the subproofs, then in addition to the implications generated by the subproofs, the suppose expression generates the implication `(implies` Φ Ψ`)`. Each subproof is read in a context with the same goal formula Ψ.

5.4 Suppose-there-is and Consider

Consider and Suppose-there-is are both similar to let-be. They both have exactly the same syntax as let-be but have slightly different usage. Suppose-there-is is used in cases where the types of the bound variables are not known to exist. For example, the proof that there is at most one identity element of a binary operator structure can be stated as follows.

```
(let-be ((m (a-binary-operator-structure)))
  (show (at-most-one (identity-of m))
    (suppose-there-is ((x (identity-of m))
                       (y (identity-of m)))
        (show (= x y)
          (show (= ((operator m) x y) x))
          (show (= ((operator m) x y) y))))))
```

The suppose-there-is expression in the above proof generates the same universally quantified that would be generated by a let-be expression but the suppose-there-is expression does not require Ontic to verify that objects of the specified types exist. By using the appropriate proof form, either let-be or suppose-there-is, the user provides a declaration to the Ontic system regarding what the user expects the Ontic system to know. This declaration is useful in uncovering faulty assumptions about the state of Ontic's knowledge.

The proof construct consider is also syntactically identical to let-be. How-

ically and the proof of the formula `(is (* n (mod2 n)) (either 0 n))` can be read successfully even if the case analysis in the body of the show is omitted.

ever, consider is intended to be used to introduce witnesses to existential statements. For example consider the following proof that in a partial order in which every set has a least upper bound, every set also has a greatest lower bound.

```
(suppose-there-is ((l (a-lattice)
                      such-that
                      (forall ((s (a-subset-of (domain l))))
                        (there-exists (a-least-upper-bound l s)))))
  (let-be ((s (a-subset-of (domain l))))
    (show (there-exists (a-greatest-lower-bound l s))
      (consider ((x (a-least-upper-bound l
                      (the-set-of-all (lower-bound l s)))))
        (show (is x (a-greatest-lower-bound l s)))))))
```

In the above proof consider is used to introduce a "witness" to the statement `(there-exists (a-greatest-lower-bound l s)`. A consider expression can only be used in a context where a goal is present. In a context with goal Ψ the expression `(consider ((`x_1 τ_1`) ... (`x_n τ_n`)) <subproof> ... <subproof>)` is equivalent to the following let-be expression.

```
(let-be ((x_1 τ_1) ... (x_n τ_n))
  (show Ψ
    <subproof>
    ⋮
    <subproof>))
```

A consider expression will either fail or will generate the goal expression of the context in which it is read.

The proof constructs `let-be`, `suppose-there-is`, and `consider` all allow `such-that` restrictions on the bound variables. A such-that restriction is used in introducing the variable `l` in the above proof about lattices.

5.5 Write-As Constructions

If an integer n is not prime then it can be written as a product pq where p and q are both integers greater than one. The statement "n can be written as pq" is an instance of a general kind of proof step which we call a "write-as" step. Write-as proof steps are closely related to the fundamental semantics of Ontic expressions. Consider the following definition.

```
(define (a-composite-number)
  (* (an-integer-greater-than 1)
     (an-integer-greater-than 1)))
```

The above definition states that a composite number is any number that can be written as the product of two integers greater than one. A proof about composite numbers might start as follows.

```
(let-be ((n (a-composite-number)))
  (let-be ((g (integers-mod n)))
    (show (not (is g (a-field)))
      (consider ((p (an-integer-greater-than 1))
                 (q (an-integer-greater-than 1)))
          such-that (is n (* p q))
        (show (not (there-exists ((multiplicative-inverse g) p)))
          ...)))))
```

The write-as step in the above proof is the following subproof.

```
(consider ((p (an-integer-greater-than 1))
           (q (an-integer-greater-than 1)))
     such-that (is n (* p q))
 ...)
```

In general, a write-as step has the following form.

```
(consider ((x_1 τ_1)
           ...
           (x_n τ_n))
   (such-that (is s t))
  ...)
```

Expressions such as the above are subject to both syntactic and semantic conditions. The syntactic restrictions are rather severe (they may be relaxed in later versions of the system). The such-that formula must be an `is` formula. Equality formulas can not be used even when the both terms involved are singleton and the is formula is semantically equivalent to the equality formula. No variable x_i may appear in any type τ_j. The expression t must be of the form $(t_i \ldots t_k)$ where each bound variable x_i appears as one of the t_j's and the order of the x_i's in t must agree with the order in which the x_i's are introduced in the `consider` expression. Finally, no bound variable x_i can appear as a proper subexpression of any t_j. For example, the following is a syntactically well formed write-as expression.

```
(consider ((f (a-binary-operator-on s))
           (x (an-element-of s)))
   such-that (is z (f (an-element-of s) x))
 ...)
```

The following expression is syntactically ill-formed both because the variables are introduced in an order that is inconsistent with the order in which they are used in the such-that expression, and because an equality is used instead of an is formula.

```
(consider ((x (an-element-of s))
           (f (a-binary-operator-on s)))
   such-that (= z (f (an-element-of s) x))
 ...)
```

To understand the semantic restriction again consider the general form

of a write-as proof.

```
(consider ((x_1 τ_1)
           ...
           (x_n τ_n))
   (such-that (is s t))
  ...)
```

For this expression to be syntactically well formed t must be of the form `(`t_1 ... t_n`)` where each x_i appears as one of the t_i's. Let t' be the result of replacing each x_i in t by the corresponding type τ_i. Ontic must be able to verify the truth of the formula `(is` s t'`)`. For example, consider following the write-as proof fragment.

```
(let-be ((n (a-composite-number)))
  (consider ((p (an-integer-greater-than 1))
             (q (an-integer-greater-than 1)))
      such-that (is n (* p q))
    ...))
```

This proof fragment is semantically well formed because Ontic can verify the formula

```
(is n
    (* (an-integer-greater-than 1)
       (an-integer-greater-than 1))).
```

as another example consider the following proof fragment.

```
(consider ((f (a-binary-operator-on s))
           (x (an-element-of s)))
   such-that (is z (f (an-element-of s) x))
 ...)
```

This fragment is semantically well formed provided that Ontic can verify the following formula.

```
(is z
    ((a-binary-operator-on s)
     (an-element-of s)
     (an-element-of s)))
```

5.6 Write-as-like Proofs

The write-as proof steps described in the previous section are used to introduce objects whose existence can be inferred from statements of a given form. For example, if one knows that the formula `(is n (f (an-integer)))` is true then one can use a write-as proof fragment to introduce the integer whose existence is implied by this formula. This would be done with the following write-as proof.

```
(consider ((x (an-integer)))
   such-that (is n (f x))
 ...)
```

There are other cases in which the existence of objects can be inferred from formulas of a certain kind. In particular, if one knows the formula `(not (is` s t`))` one can infer that there exists some possible value of s that is not a possible value of t. This object whose existence can be inferred from this statement can be introduced with the following proof fragment.

```
(consider ((x s))
    such-that (not (is x t))
 ...)
```

Another case in which the existence of certain objects can be inferred is when one knows a formula of the form `(not (at-most-one` s`))` In this case

there must exist two distinct possible values of s. These two objects can be introduced with a proof fragment of the following form.

```
(consider ((x s)
           (y s))
    such-that (not (= x y))
  ...)
```

Any proof fragment in which a such-that condition immediately follows all of the variable bindings in a consider proof must be of one of the above three forms, i.e., it is either a write-as proof fragment justified by a formula of the form `(is` s t`)`, or the introduction of a individual whose existence is justified by a formula of the form `(not (is` s t`))`, or the introduction of two individuals whose existence is justified by a formula of the form `(not (at-most-one` s`))`.

5.7 Proof Idioms

Although Ontic proofs are not based on explicit proof rules, there are certain patterns, or idioms of inference, which are guaranteed to be successful. Each idiom can be viewed as an inference rule. For the most these idioms are rather obvious. For example, to prove a formula of the form `(is` s t`)` where s has more than one possible value one uses the following proof fragment.

```
(show (is s t)
  (suppose-there-is ((x s))
    (show (is x t)
      ...)))
```

If the inner show succeeds the outer show is guaranteed to succeed. The following is a list of proof idioms. In each case if the inner show succeeds the outer show is guaranteed to succeed. The inner shows may be given in any order and inner shows which are obvious to Ontic can be omitted. Any proof idiom that involves `consider` allows the `consider` subproof to contain

a such-that after the bindings, i.e., the `consider` subproof can be a write-as or write-as like proof step as described in sections 5.5 and 5.6.

with `consider` proofs that involve write-as steps

```
(show (is x (f (a-foo)))
  (consider ((y w))
    (show (is y (a-foo))
      ...)
    (show (is x (f y))
      ...)))
```

```
(show (is x (either s t))
  (suppose (not (is x s))
    (show (is x t)
      ...)))
```

```
(show (is x (some-such-that y (a-foo) Φ[y]))
  (show (is x (a-foo))
    ...)
  (show Φ[x]
    ...))
```

```
(show (a-most-one (a-foo))
  (suppose-there-is ((x (a-foo))
                     (y (a-foo)))
    (show (= x y)
      ...)))
```

```
(show (there-exists (a-foo))
  (consider ((x_1 τ_1)
             ⋮
             (x_n τ_n))
```

```
    (show (there-exists t[x_1, ..., x_n])
      ...)
    (show (is t[x_1, ..., x_n] (a-foo))
      ...)))


(show (forall ((x τ)) Φ[x])
  (suppose-there-is ((x τ))
    (show Φ[x])))


(show (exists ((x (a-foo))) Φ[x])
  (consider ((x_1 τ_1)
             ⋮
             (x_n τ_n))
    (show (is x_n (a-foo))
      ...)
    (show Φ[x_n])))


(show (is f (an-operator-from τ to σ))
  (show (is (a-domain-member-of f) τ)
    ...)
  (show (is τ (a-domain-member-of f))
    ...)
  (suppose-there-is ((x τ))
    (show (is (f x) σ))
      ...))


(show (is f (an-operator-from τ_1 τ_2 to σ))
  (show (is (a-domain-member-of f) τ_1)
    ...)
  (show (is τ_1 (a-domain-member-of f))
    ...)
  (suppose-there-is ((x τ_1))
```

```
    (show (is (f x) (an-operator-from τ2 to σ))
      ...)))


(show (is f (a-function-from τ to σ))
  (show (is (a-domain-member-of f) τ)
    ...)
  (show (is τ (a-domain-member-of f))
    ...)
  (suppose-there-is ((x τ))
    (show (is (f x) σ))
      ...)
    (show (singleton (f x))
      ...))


(show (is f (a-function-from τ1 τ2 to σ))
  (show (is (a-domain-member-of f) τ1)
    ...)
  (show (is τ1 (a-domain-member-of f))
    ...)
  (suppose-there-is ((x τ1))
    (show (is (f x) (a-function-from τ2 to σ))
      ...)))
```

5.8 Induction Proofs

Proving facts about recursively defined thunks and operators requires mathematical induction. For example, consider the following definition of a `a-node-connected-to`. The operator `a-node-connected-to` takes two arguments, a graph and a node in the graph, and nondeterministically returns any node that can be reached from the given node by traversing arcs of the graph.

```
(define (a-node-connected-to (g (a-graph)) (n (a-node-of g)))
```

```
  (either n
          (a-node-connected-to g (a-neighbor-of g n))))
```

It should be clear that for any graph g and node n of g the possible values of `(a-node-connected-to` g n`)` are all nodes of g. We would like to be able to write a proof of the following form.

```
(let-be ((g (a-graph)))
  (suppose-there-is ((n (a-node-of g)))
    (show (is (a-node-connected-to g n)
              (a-node-of g))
      ...)))
```

Unfortunately, the proof constructs discussed in the previous sections can not be used to fill in the above proof. Given the recursive definition Ontic knows that the expression `(a-node-connected-to g n)` is equivalent to `(either n (a-node-connected-to g (a-neighbor-of g n)))`. Unfortunately, this equivalence, in itself, does not imply the desired theorem — the equivalence is true of a nonstandard interpretations of `a-node-connected-to` which includes the number 1 in the possible output values independent of the inputs. This equivalence is called the fixed point equation of the definition (see section 3). In general, properties of recursively defined thunks and operators can not be proved from the fixed point equation alone.

In Ontic, properties of recursive concepts are proved by *computational induction* — when verifying a property of an operator we assume that the property holds for recursive calls to the operator and, under this assumption, show that the property holds for the operator. This is analogous to the classical notion of partial correctness for computer programs. An Ontic induction proof that every node connected to a node of g is a node of g is given below.

```
(let-be ((g (a-graph)))
  (show-by-induction-on ((n (a-node-of g))
                         (m (a-node-connected-to g n)))
```

```
    (is m (a-node-of g))))
```

The above proof works as written — no additional detail is needed. Consider the `show-by-induction-on` subproof.

```
(show-by-induction-on ((n (a-node-of g))
                       (m (a-node-connected-to g n)))
    (is m (a-node-of g)))
```

This proof fragment generates the following lemma.

```
(forall ((n (a-node-of g))
         (m (a-node-connected-to g n)))
  (is m (a-node-of g)))
```

In general, the proof construct `show-by-induction-on` has the same syntax as `suppose-there-is` except that the type given to the final variable must be an application of a recursively defined thunk or operator. The induction is always a computational induction on the definition of the recursive operator in the final variable binding. In computational induction we assume that the desired result holds for recursive calls to the procedure and prove that this implies the desired result for the procedure. To capture the idea that the recursive calls satisfy the induction hypothesis the system creates a version of the recursive procedure which is guaranteed to satisfy the hypothesis. This version is called "wishful" because the induction hypothesis is assumed to hold by a kind of wishful thinking. Ontic automatically converts the above `show-by-induction-on` to the following.

```
(suppose (forall ((n (a-node-of g))
                  (m ((wishful-version-of a-node-connected-to)
                      g
                      n)))
           (is m (a-node-of g)))
  (suppose (forall ((g (a-graph))
```

```
                (n (a-node-of g)))
          (is ((wishful-version-of a-node-connected-to) g n)
              (either n
                      ((wishful-version-of a-node-connected-to)
                       g
                       (a-neighbor-of g n)))))
(suppose-there-is ((n (a-node-of g))
                   (m (either n
                              ((wishful-version-of a-node-connected-to)
                               g
                               (a-neighbor-of g n)))))
  (show (is m (a-node-of g))))))
```

The first `suppose` expression given the induction hypothesis for the wishful version. The induction hypothesis is identical to the lemma to be proved except that the call to the operator `a-node-connected-to` has been replaced by a call to the wishful version. If there operator on which the induction is being performed appears more than once the binding list then only the outermost application in the final binding is replaced by the wishful version. Examples of this will be given below. The second `suppose` gives the fact that the wishful version is like the original in that any value generated by the wishful version is generated by recursion on the definition of `a-node-connected-to` but where the wishful version is still used on recursive calls. The type of `m` in the `suppose-there-is` proof expresses the fact that `m` is generated by a call to `a-node-connected-to` in which recursive calls have been replaced by the wishful version. Finally, given these suppositions, the system proves that `m` is a node of `g`. If the innermost show succeeds then the original `show-by-induction-on` succeeds.

Now consider an arbitrary recursive definition.

`(define (`f `(`y_1 τ_1`) ... (`y_k τ_k`))`
$B[y_1, \ldots, y_k, f]$`)`

The body of the definition is an expression that involves the parameters

of the procedure as well as the operator f that is being defined. Given the way that Ontic expands `show-by-induction-on` proofs it is possible to state an induction proof idiom which has the standard idiom property that if the innermost show succeeds the whole proof is guaranteed to succeed. The proof idiom is stated for induction on the recursive definition of f as stated above.

```
(show-by-induction-on ((x_1 τ_1)
                       ⋮
                       (x_n τ_n)
                       (y (f t_1 ... t_k)))
    Φ[x_1, ..., x_n, y]
  (suppose-there-is ((z B[t_1, ..., t_n, (wishful-version-of f)]))
    (show Φ[x_1, ..., x_n, z]
          ...)))
```

If the innermost show succeeds the overall induction proof is guaranteed to succeed. In the body of the induction proof the induction hypothesis and recursive unrolling properties are assumed for the wishful version. The following is the instance of the above general idiom for the proof about `a-node-connected-to`.

```
(show-by-induction-on ((n (a-node-of g))
                       (m (a-node-connected-to g n)))
    (is m (a-node-of g))
  (suppose-there-is ((m2 (either n
                                 ((wishful-version-of a-node-connected-to)
                                  g
                                  (a-neighbor-of g n)))))
    (show (is m (a-node-of g))
      ...)))
```

If the inner show succeeds then the induction proof is guaranteed to succeed. As pointed out above, however, this particular proof succeeds without a body. In general the body only need generate enough lemmas to make the automated induction proof succeed in the presence of those lemmas.

As another example of an induction proof consider the following definition of append.

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


(define (append (l1 (a-list)) (l2 (a-list)))
  (if (= l1 'nil)
      l2
      (cons (car l1) (append (cdr l1) l2))))


(let-be ((l2 (a-list)))
  (show-by-induction-on ((l1 (a-list)))
      (singleton (append l1 l2))))


(let-be ((l2 (a-list))
         (l3 (a-list)))
  (show-by-induction-one ((l1 (a-list)))
    (= (append (append l1 l2) l3)
       (append l1 (append l2 l3)))))
```

Both of the above proofs are by computational induction on the definition of the thunk a-list. The first of the two proofs establishes that append is a function (has exactly one output for any particular pair of input lists). To better understand the variety of possible induction proofs it is worth considering the Ontic expansion of the proof that append is a function. This expansion is given below.

```
(let-be ((l2 (a-list)))
  (suppose (forall ((l1 ((a-wishful-version-of a-list))))
             (singleton (append l1 l2)))
    (suppose (forall ((l1 ((a-wishful-version-of a-list))))
               (is l1 (either 'nil
```

```
                              (cons (a-thing)
                                    ((a-wishful-version-of a-list))))))
      (suppose-there-is
           ((l1 (either 'nil
                        (cons (a-thing)
                              ((a-wishful-version-of a-list))))))
        (show (singleton (append l1 l2)))))))
```

The expansion of the second induction proof above, the proof that `append` is associative, is similar — it is also a computational induction on the definition of `a-list`. It is interesting to note that the associativity of `append` can also be proved by induction on the definition of `append` as in the following proof.

```
(show-by-induction-on ((l1 (a-list))
                       (l2 (a-list))
                       (l3 (a-list))
                       (l4 (append (append l1 l2) l3)))
   (= l4 (append l1 (append l2 l3))))
```

The fact that the append of two lists is a list can also be proven by list induction on the first argument to append or by computational induction on append itself. The induction on `append` itself is given below.

```
(show-by-induction-on ((l1 (a-list))
                       (l2 (a-list))
                       (l3 (append l1 l2)))
  (is l3 (a-list)))
```

It is interesting to compare the above induction proof with the following proof attempt.

```
(let-be ((l1 (a-list))
         (l2 (a-list)))
```

```
(show-by-induction-on ((l3 (append l1 l2)))
   (is l3 (a-list))))
```

This proof attempt fails because the induction hypothesis is not strong enough. This failed proof attempt constructs the following induction hypothesis.

```
(suppose (forall ((l3 ((wishful-version-of append) l1 l2)))
           (is l3 (a-list)))
  ...)
```

The above successful proof that the append of two lists is a list generates the much stronger induction hypothesis.

```
(suppose (forall ((l1 (a-list))
                  (l2 (a-list))
                  (l3 ((wishful-version-of append) l1 l2)))
           (is l3 (a-list)))
  ...)
```

This stronger induction hypothesis can be used for arbitrary invocations of the wishful version of `append` while the weaker induction hypothesis can only be used when the wishful version is applied to the particular lists `l1` and `l2`. In general, including more bindings within a `show-by-induction-on` will generate a stronger induction hypothesis and will generally make the proof more likely to succeed.

5.9 Modules

When Ontic first reads a definition it usually does not fully understand the newly defined concept. A concept is "fully understand" when occurrences of the concept no longer have to be replaced by the definition of the concept. In practice this means that the lemma library must contain enough facts

about the concept so that the definition of the concept is no longer needed, or only needed in rare occasions. For example, a real number is defined as a set of rational numbers — the set of rationals less then or equal to the reals. However, once enough facts have been proven about the real numbers this definition is no longer useful and we can think of a real number as a point on a line. Until enough facts are proven, however, the definition of a real as a set of rationals must be invoked. The idea that definitions are only needed until enough facts have been proven motivates the concept of a *module*. An Ontic module consists of a set of definitions and lemmas. The following is a simple example of the use of an Ontic module.

```
(defmodule
  (define (make-an-employee-record (n (a-name)) (s (a-whole-number)))
    (list 'a-structure-object 'the-employee-record n s))


  (define (an-employee-record)
    (make-employee-record (a-name) (a-whole-number)))


  (define (employee-name (r (an-employee-record)))
    (car (cdr r)))


  (define (employee-salary (r (an-employee-record)))
    (car (cdr (cdr r))))


  (let-be ((n (a-name))
           (s (a-whole-number)))
    (show (is (make-employee-record n s)
              (an-employee-record))
    (show (= (employee-name (make-an-employee-record n s))
             n))
    (show (= (employee-salary (make-an-employee-record n s))
```

```
            s))))


  (let-be ((r (an-employee-record)))
    (show (is (employee-name r) (a-name)))
    (show (is (employee-salary r) (a-whole-number)))
    (show (= r (make-an-employee-record (employee-name r)
                                        (employee-salary r))))))
```

The above module has essentially the same effect as the following defstruct.

```
(defstruct an-employee-record
  (employee-name (a-name))
  (employee-salary (a-whole-number)))
```

Once the lemmas in the above module are proven the definitions which given the implementation of the employee records in terms of lists are no longer needed. However, Ontic does *not* forget the definitions — Ontic heuristically reduces the amount of effort it invests in using the definitions. Ontic always attempts to find the smallest way of writing any given expression. The expression `(make-an-employee-record n s)` is smaller (has fewer tokens) than the expression `(list 'a-structure-object 'the-employee-record n s)`. Since the list expression is not the smallest way of writing the record term, lemmas about lists will not be automatically applied to the record term. Inside the module, however, the definitions are not allowed to reduce the size of expressions and the system focuses on the term

```
(list 'a-structure-object 'the-employee-record n s)
```

rather than

```
(make-an-employee-record n s).
```

This makes the lemmas within the module easier to prove. In general, it a good idea to put definitions inside modules that include lemmas stating the most basic facts about the definitions. However, modules should not contain more than the basic facts about a definition because inference in-

volving defined symbols is more efficient outside of the module containing the definition.

5.10 The Emacs User Interface

The Ontic system is implemented in Common Lisp and runs as an inferior process to the emacs text editor under unix. Ontic definitions and proofs are written in an emacs buffer in ontic mode. In an appropriately configured emacs system, emacs enters ontic mode whenever it is used to edit a file which ends in extension `.ont`. The following characters are defined in Ontic mode. Most character commands start with the <control>-z prefix.

C-z e Evaluate Form. This is used to enter definitions and to run proofs. Place the cursor at any line that is part of a definition or proof and type this character command. When performed on a proof the cursor moves through the proof and stops at the first place it is unable to verify a step. The user can then add proof detail at that point. Any proof that is read successfully expands the lemma library. When this command is performed inside a module the entire module is read and the lemma library is not expanded unless the the entire module succeeds. To prevent accident expansion of the lemma library one can insert the proof `(show false)` at the end of the module. The Ontic system is loaded the first time this command is executed so you should expect a considerable delay. The progress of the load is reported in the mode line of the emacs buffer.

C-z f Faith load a file. File is read into a buffer named *faith* and evaluated in faith-mode. Faith mode is a way of extracting definitions and lemmas from a file without wasting time verifying proofs. It should only be used with files that have been previously verified. If an error happens during a faith load the *faith* buffer is brought to the currently selected window so the user can see where the error occurred. If no errors occur the temporary *faith* buffer is killed.

C-z i Ontic-init. This will restore the lemma library the initial state. All definitions and lemmas entered since starting the system will be lost. If ontic hasn't yet been started, this command will start it.

C-z r Evaluate a region. This is like the evaluate expression command except that it evaluates all expressions in the current region.

C-z b Evaluate the buffer. This evaluates the entire buffer.

C-z g Abort last command. This is useful for terminating verifications where the user can see a problem with a proof while the system is still evaluating it.

C-z s Restart the ontic lucid process. This has the same effect as C-zi but takes much longer since it starts a new lisp process. This command is useful if for some reason the Lisp process becomes unresponsive.

C-u C-z e Evaluate a form using faith-mode. This is like the evaluate region command except that proofs are not checked.

C-u C-z r Evaluate a region using faith-mode. This evaluates a region without checking proofs.

C-u C-z b Evaluate the buffer using faith-mode. This evaluates the buffer without checking proofs.

6 Examples

Ontic can be viewed as a strongly typed functional programming language. As a programming language, the main difference between Ontic and other typed functional languages involves the nature of Ontic's type system. Ontic uses "semantic" types. Intuitively this means that any set which can be formally defined as a mathematical object can be used as a type in Ontic. This implies that, given application of an operator to an argument, it is not possible in general to determine if the argument is a member of the type accepted by the operator. We will not discuss this observation further here

except to say that we do not feel that this poses any difficulty in practice — in our experience this kind of type checking is practical.

Ontic, however, is more than a functional programming language. Ontic is a language for formally defining arbitrary concepts. Since most readers are already familiar with the variety of programs that can be written in a functional programming language, this section emphasizes examples of things that can be defined in Ontic which can not be defined in other programming languages. It is important to remember that not all Ontic expressions are executable. Rather than execute expressions, the Ontic system reasons about them. We start with the Dedekind cut construction of the real numbers.

6.1 Constructing the Real Numbers

The classical Dedekind cut construction of the real numbers is given below. It is interesting to note that this construction is purely predicative, i.e., the primitives `a-set` and `a-thing` are never used.

We start with some simple subsets of the integers and simple relation on integers.

```
(define (a-natural-number)
  (either 0 (+ 1 (a-natural-number))))


(define (a-whole-number)
  (either 1 (+ 1 (a-whole-number))))


(define (an-integer-greater-than (n (an-integer)))
  (either (+ n 1)
          (an-integer-greater-than (+ n 1)))))
```

Now we construct the nonnegative fractions.

```
(defstruct fraction
  (numerator (a-natural-number))
  (denominator (a-whole-number)))


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


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


(define (a-fraction-less-than (f1 (a-fraction)))
  (some-such-that f2 (a-fraction)
    (is (* (numerator f1) (denominator f2))
        (an-integer-greater-than (* (numerator f2)
                                    (denominator f1))))))


(define (frac-difference (f1 (a-fraction))
                         (f2 (a-fraction-less-than f1)))
  (make-fraction (- (* (numerator f1) (denominator f2))
                    (* (numerator f2) (denominator f1)))
                 (* (denominator f1) (denominator f2))))


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

Two fractions can represent the same rational number. We now we construct

the nonnegative rational numbers by stating that each rational number is an equivalence class of rationals.

```
(define (the-rational-class-of (f (a-fraction)))
  (the-set-of-all (an-equivalent-fraction f)))


(define (a-rational)
  (the-rational-class-of (a-fraction)))


(define (rat-+ (r1 (a-rational)) (r2 (a-rational)))
  (the-rational-class-of
    (frac-+ (a-member-of r1) (a-member-of r2))))


(define (rat-* (r1 (a-rational)) (r2 (a-rational)))
  (the-rational-class-of
    (frac-* (a-member-of r1) (a-member-of r2))))


(define (a-rational-less-than (r (a-rational)))
  (the-rational-class-of
    (a-fraction-less-than (a-member-of r))))


(define (rat-difference (r1 (a-rational))
                        (r2 (a-rational-less-than r1)))
  (the-set-of-all
    (the-rational-of
      (frac-difference (a-member-of r1)
                       (a-member-of r2)))))
```

Now we construct the nonnegative real numbers as Dedekind cuts in the rationals. Intuitively, each real number x gets identified with the set of rationals less than or equal to x. The second clause in the following definition

is needed to ensure that if x happens to be a rational then x is included in the set.

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


(define (cut-+ (x (a-cut)) (y (a-cut)))
  (the-set-of-all (rat-+ (a-member-of x) (a-member-of y))))


(define (cut-* (x (a-cut)) (y (a-cut)))
  (the-set-of-all (rat-* (a-member-of x) (a-member-of y))))


(define (a-cut-less-than (c (a-cut)))
  (some-such-that c2 (a-cut)
        (and (not (= c2 c))
             (is c2 (a-subset-of c)))))


(define (a-cut-greater-than (c (a-cut)))
  (some-such-that c2 (a-cut)
    (and (not (= c2 c))
         (is c (a-subset-of c2)))))


(define (a-nonzero-cut)
  (some-such-that c (a-cut)
```

```
    (there-exists (a-member-of c))))


(define (cut-difference (c1 (a-cut))
                        (c2 (a-cut-less-than c1)))
  (the-set-of-all
    (let ((r1 (a-member-of c1))
          (r2 (a-member-of c2)))
      (when (is r2 (a-rational-less-than r1))
        (rat-difference r1 r2)))))
```

The cuts represent the positive reals. We can now define a real number to be a pair of a cut and a sign. Care must be taken to ensure that there is only one representation of zero.

```
(define (a-sign)
  (either 1 -1))


(defstruct a-real-number
  (sign (a-sign))
  (magnitude (if (= sign 1)
                 (a-cut)
                 (a-nonzero-cut))))


(define (real-+ (x (a-real-number)) (y (a-real-number)))
  (cond ((= (sign x) (sign y))
         (make-a-real-number (sign x)
                             (cut-+ (magnitude x)
                                    (magnitude y))))
        ((= -1 (sign x))
         (real-cut-difference (magnitude y) (magnitude x)))
        ((= -1 (sign y))
```

```
         (real-cut-difference (magnitude x) (magnitude y)))))


(define (real-cut-difference (c1 (a-cut)) (c2 (a-cut)))
  (cond ((= c1 c2) (make-a-real-number 1 0))
        ((is c1 (a-cut-greater-than c2))
         (make-a-real-number 1 (cut-difference c1 c2)))
        ((is c2 (a-cut-greater-than c1))
         (make-a-real-number -1 (cut-difference c2 c1)))))


(define (real-* (x (a-real-number)) (y (a-real-number)))
  (make-a-real-number (* (sign x) (sign y))
                      (cut-* (magnitude x) (magnitude y))))


(define (a-real-less-than (x (a-real-number)))
  (if (= 1 (sign x))
      (either (make-a-real-number -1 (a-nonzero-cut))
              (make-a-real-number 1
                                  (a-cut-less-than
                                    (magnitude x))))
      (make-a-real-number -1
                          (a-cut-greater-than
                            (magnitude x)))))
```

6.2 Axiomatizing the Real Numbers

Most modern textbooks on real analysis introduce the real numbers by giving a set of axioms that the real numbers satisfy. This can also be done using the impredicative features of the Ontic language, i.e., the primitives `(a-set)` and `a-thing`. The following structure definition assumes that certain concepts have already been defined, such as the concept of a total order, the concept of a commutative function, and the concept of the least upper bound of a

set. We leave it to the reader to define these concepts. This definition of the reals is *categorical*, i.e., any two structure objects satisfying the such-that formulas in the structure definition must be isomorphic. We say that the definition determines the structure of the real numbers up to isomorphism. Of course one would like to prove many statements. One would like to prove that the Dedekind cut construction of the reals given in the previous section provides a model of the axiomatic specification given here. One would also like to prove that any two structure instances satisfying the given conditions are isomorphic. These statements can be proven using the Ontic verification system described in section 5.

```
(defstruct the-reals
  (domain (a-set))
  (less-or-equal (a-total-order-on domain))
  (plus (a-binary-function-on domain))
  (times (a-binary-function-on domain))
  (zero (an-element-of domain))
  (one (an-element-of domain))
  (minus (a-unary-function-on domain))
  (one-over (a-unary-function-on domain))


  such-that
  (not (= zero one))
  (is plus (an-associative-function-on domain))
  (is plus (a-commutative-function-on domain))
  (forall ((x (an-element-of domain)))
    (and (= (plus x zero) x)
         (= (plus x (minus x)) zero)))


  (is times an-associative-function)
  (is times a-commutative-function)
  (forall ((x (an-element-of domain)))
    (and (= (times x one) x)
         (implies (not (= x zero))
           (= (times x (one-over x)) one))))
```

```
(forall ((x (an-element-of domain))
         (y (an-element-of domain))
         (z (an-element-of domain)))
  (= (times x (plus y z))
     (plus (times x y) (times x z))))


(is zero (less-or-equal one))
(forall ((x (an-element-of domain))
         (y (an-element-of domain))
         (z (less-than y)))
  (is (plus x z) (less-than (plus x y))))
(forall ((x (an-element-of domain))
         (z (an-element-of domain)))
  (implies (and (is zero (less-or-equal x))
                (is zero (less-or-equal y)))
    (is (zero (less-or-equal (times x y))))))
(forall ((s (a-subset-of domain)))
  (implies (and (there-exists (an-element-of s))
                (there-exists
                  (an-upper-bound-of domain
                                     less-or-equal
                                     s)))
      (there-exists (a-least-upper-bound-of domain
                                            less-or-equal
                                            s)))))
```

6.3 Unification

An expressions is a kind of tree. Tree is a general concept.

Unification is a good example of a concept that can be defined declaratively. The thunk `an-expression` is defined in section 2.11.

```
(defstruct a-variable
  (variable-print-name (a-symbol)))


(define a-substitution
  (lambda-fun ((var (a-variable))) (an-expression)))


(define (apply-substitution (sub (a-substitution))
                            (exp (an-expression)))
   (if (is exp (a-variable))
       (sub exp)
       (if (is exp (a-symbol))
           exp
           (map (lambda ((x2 (an-expression)))
                  (apply-substitution sub e2))
                exp))))


(define (a-unifier-of (e1 (an-expression))
                      (e2 (an-expression)))
   (some-such-that sub (a-substitution)
     (= (apply-substitution sub e1)
        (apply-substitution sub e2))))


(define (compose-substitutions (s1 (a-substitution))
                               (s2 (a-substitution)))
  (lambda ((var (a-variable)))
    (apply-substitution s1 (s2 var))))


(define (as-general-a-substitution-as (s (a-substitution)))
  (some-such-that s2 (a-substitution)
    (exists ((s3 (a-substitution)))
```

```
      (= s (compose-substitutions s3 s2)))))


(define (a-most-general-unifier (e1 (an-expression))
                                (e2 (an-expression)))
  (some-such-that u (a-unifier-of e1 e2)
    (forall ((u2 (a-unifier-of e1 e2)))
      (is u (as-general-a-substitution-as u2)))))
```

7 Epilogue

Most interactive verification systems require the user of the system to have a fairly deep understanding of the inference mechanisms involved. The Ontic system has been designed to be easy to use. Our hope is that people will be able to use the system with very little understanding of the underlying inference mechanisms The inference mechanisms have not been discussed in this manual. We believe that in order for a system to be easy to use it must be intelligent. People have great difficulty writing proofs for facts that seem "obvious". A system which demands proofs of obvious facts is very difficult to use. Conversely, any sufficiently intelligent verification system should be fairly easy to use — a system which rarely required any proof of any form would be quite easy to use. Ease of use can not be separated from intelligence.

The Ontic system is under continuous development and is steadily gaining intelligence. The current system requires far less proof detail than did the earlier Ontic system described in [McAllester, 1989]. The fact that the system is still evolving rapidly implies that many of the examples in this manual will be obsolete in a matter of months — many of the proofs given above are already no longer needed.

Ontic has already undergone a long period of evolutionary development. We feel that any truly useful verification system must undergo such development. It seems that theory alone can not predict whether a given technique

will work. Building a system provides a host of information about what methods work well and what kinds of proofs are difficult to verify. We expect to continue to benefit by examining verifications done by users other than ourselves. Users can help the Ontic project by reporting cases where Ontic appears particularly stupid. Of course we are equally interested in cases where Ontic appears spectacularly intelligent.

References

[Boyer and Moore, 1979] Robert S. Boyer and J Struther Moore. *A Computational Logic*. ACM Monograph Series. Academic Press, 1979.

[Constable *et al.*, 1986] R. L. Constable, S. F. Allen, H. M. Bromely, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Rowe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Development system*. Prentice-Hall, 1986.

[Givan *et al.*, 1991] Robert Givan, David McAllester, and Sameer Shalaby. Natural language based inference procedures applied to schubert's steamroller. In *AAAI-91*, pages 915–920. Morgan Kaufmann Publishers, July 1991.

[Gordon *et al.*, 1979] 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.

[Harper *et al.*, 1987] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *LICS-87*, pages 194–204. IEEE Computer Society Press, 1987.

[Martin-Löf, 1982] Per Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175. North Holland, 1982.

[McAllester and Givan, 1989] D. McAllester and R. Givan. Natural language syntax and first order inference. Memo 1176, MIT Artificial Intelligence Laboratory, October 1989. To Appear in AIJ.

[McAllester *et al.*, 1989] D. McAllester, R. Givan, and T. Fatima. Taxonomic syntax for first order inference. In *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, pages 289–300, 1989. To Appear in JACM.

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

[McAllester, 1990] D. McAllester. Automatic recognition of tractability in inference relations. Memo 1215, MIT Artificial Intelligence Laboratory, February 1990. To appear in JACM.

[McAllester, 1991a] David McAllester. Socratic sequent systems. *SIGART Bulletin*, 2(3):98–101, July 1991.

[McAllester, 1991b] David McAllester. Some observations on cognitive judgements. In *AAAI-91*, pages 910,915. Morgan Kaufmann Publishers, July 1991.

[McCarthy, 1967] 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.

[Sussman and Abelson, 1985] Gerald Sussman and Herald Abelson. *Structure and Interpretation of Computer Programs*. MIT Press, 1985. page 317.

[William Farmmer, 1990] Javier Thayer William Farmmer, Joshua Guttman. Imps: An interactive mathematical proof system. In *CADE-10*, pages 653–654. Springer-Verlag, 1990.