Technical Report 979

ONTIC: A Knowledge Representation System for Mathematics

David A. McAllester

MIT Artificial Intelligence Laboratory

ONTIC:
A Knowledge Representation System for Mathematics

by

David Allen McAllester

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 1987 in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy in
Computer Science

Abstract: Ontic is an interactive system for developing and verifying mathematics. Ontic's verification mechanism is capable of automatically finding and applying information from a library containing hundreds of mathematical facts. Starting with only the axioms of Zermelo-Fraenkel set theory, the Ontic system has been used to build a data base of definitions and lemmas leading to a proof of the Stone representation theorem for Boolean lattices. The Ontic system has been used to explore issues in knowledge representation, automated deduction, and the automatic use of large data bases.

Acknowledgments: This thesis would never have been completed without the unending encouragement of my thesis supervisor Gerry Sussman. Albert Meyer forced me to clarify my thinking and to demonstrate the utility of my ideas by constructing a working system. Jonathan Rees provided stimulating discussions and useful feedback as the first user of the Ontic formal language. I would also like to thank all the people who read and commented on early versions of the thesis.

This thesis describes research done at the Artificial Intelligence Laboratory at the Massachusetts Institute of Technology, supported by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-86-K-0180.

Contents

Chapter 1

Ontic in Brief

Ontic is a computer system for verifying mathematical arguments. Starting with the axioms of Zermelo-Fraenkel set theory, including Zorn's lemma as a version of the axiom of choice, the Ontic system has been used to to define concepts involving partial orders and lattices and to verify a proof of the Stone representation theorem for Boolean lattices. This theorem involves an ultrafilter construction and is similar in complexity to the Tychonoff theorem in topology which states that an arbitrary product of compact spaces is compact. The individual steps in the proof were verified with an automated theorem prover. The Ontic theorem prover automatically accesses a lemma library containing hundreds of mathematical facts; as more facts are added to the system's lemma library the system becomes capable of verifying larger inference steps.

The Ontic theorem prover is based on what I call object-oriented inference. Object-oriented inference is a forward chaining inference process applied to a large lemma library and guided by a set of *focus objects.* The focus objects are terms in the sense of first order predicate calculus; they are expressions which denote objects. It is well known that unrestricted forward chaining starting with a large lemma library leads to an immediate combinatorial explosion. However, the Ontic theorem prover is guided by the focus objects; the inference process is restricted to statements that are, in a technical sense, about the focus objects. Thus the inference process is "object-oriented". In verifying an argument the user specifies the set of focus objects. For example the user may tell the system to consider an arbitrary lattice L, an arbitrary subset S of L, and an arbitrary member x of S. Ontic's inference mechanisms are restricted to a finite set of formulas that are about the given focus objects. Certain forward chaining constraint propagation techniques can be effectively applied to this finite set of formulas. Natural language mathematical arguments, like those found in textbooks and journals, appear to be object-oriented in the sense that they instruct the reader to focus on certain objects. Thus Ontic's object-oriented inference mechanisms seem well suited for verifying natural arguments.

There are two motivations for building a system for verifying natural arguments. First there is an engineering motive: a sufficiently powerful mechanical verifier could have a variety of important practical applications, such as ensuring the correctness of mathematical arguments, the correctness of software systems, and the correctness of engineered devices in general. Second, the construction of a verification system for natural arguments can be motivated in terms of cognitive psychology. A verification system for natural arguments provides a computational model of the human cognitive processes involved in verifying arguments. The plausibility of such a cognitive model can be judged by comparing the length and structure of the arguments acceptable to people with the length and structure of arguments acceptable to the cognitive model.

The engineering motive and the cognitive model motive for building verification systems are not independent; a verification system that is a good cognitive model is likely to be pragmatically useful. More specifically, a verification system is a good cognitive model to the extent that arguments acceptable to the model are similar to the arguments acceptable to people. Thus if a verification system is a good cognitive model then it should be easy to convert arguments that are acceptable to people to arguments that can be verified by the system; a system that is a good cognitive model provides a good "impedance match" between the human user and the verification system.

On the other hand the two motivations for verifications system, the engineering motive and the cognitive model motive, are different motivations with different criteria for success. A verification system that exhibits clearly superhuman performance in its ability to verify statements is a bad cognitive model but a good verifier from an engineering point of view. It turns out that Ontic's mechanism for reasoning about equality, congruence closure, leads to some clear examples of superhuman performance on the part of the Ontic system. Thus congruence closure is not a good cognitive model for the way people reason about equality—there are equality reasoning mechanisms which are weaker than congruence closure which provide better cognitive models. However, from an engineering point of view congruence closure is better than the weaker mechanism (at least on serial machines). The analysis of congruence closure as a bad cognitive model is presented in detail in chapter 3.

The Ontic system was designed with both motivations in mind—an attempt was made to make the system a pragmatically effective verification system and the same time to make the system a rough model of human mathematical cognition. The Ontic system should be judged on two independent grounds relative to these two goals. First, one can evaluate the system as an engineered device for verifying proofs by attempting to use the system for that purpose. Second, one can attempt to evaluate the system as a cognitive model by judging the similarity between natural language arguments acceptable to people and formal arguments acceptable to the system.

The remainder of this chapter is divided into four sections. The first section briefly discusses the nature of natural language mathematical arguments. The second section of the chapter discusses the formal language used in the Ontic system. The third section describes the user-level interface to the system and gives several examples of arguments verified by the system. The fourth section describes the object-oriented inference mechanisms in more detail.

The relationship between Ontic and previous work in reasoning, knowledge representation, and theorem proving is discussed in detail in chapter 2. Chapter 3 presents an analysis of the Ontic system as a cognitive model giving examples of both superhuman and subhuman performance on the part of the Ontic system. Chapters 4 and 5 give a mathematically precise account of the inference mechanisms as marker propagation algorithms on certain kinds of graph structure. Chapter 6 gives a mathematically precise definition of the Ontic formal language and chapter 7 gives a mathematically precise account of the compilation process by which expressions in the formal language are converted into graph structure. Chapter 8 lists some potential applications of automated inference systems such as Ontic and chapter 9 summarizes the main features of the Ontic system.

1.1 The Nature of Natural Arguments

By a "natural mathematical argument" I mean a proof written in a natural language, such as English, that would be acceptable as a fully worked out proof in a textbook or journal article. A natural mathematical argument consists of a sequence of natural language statements and the human reader is expected to use his or her knowledge and intelligence to see that each step clearly and necessarily follows from the previous steps. As an example of a natural argument consider the following proof that the square root of 2 is irrational.

> Suppose that the square root of two were rational, i.e.
>
> $$\frac{p^2}{q^2} = 2$$
>
> The squares p^2 and q^2 must each have an even number of prime factors. Thus, if p^2/q^2 is an integer then this integer must also have an even number of prime factors. But 2 has only a single prime factor so p^2/q^2 cannot equal 2.

This argument is perfectly rigorous; every step clearly follows from the previous steps and the conclusion is clearly established; $\sqrt{2}$ must be irrational. However, understanding this argument requires knowing certain facts about arithmetic and multisets. More specifically the above argument implicitly rests on the following facts:

1. The fundamental theorem of arithmetic — every natural number has a unique multiset of prime factors.

2. The multiset of factors of p^2 is the multiset union of the prime factors of p with itself.

3. The multiset union of a multiset with itself has an even number of members (an even multiset cardinality).

4. If p/q is an integer then the multiset of prime factors of q must be a subset of the multiset of prime factors of p.

5. If p/q is an integer then the multiset of prime factors of p/q is the multiset difference of the prime factors of p and the prime factors of q.

6. If the multisets m_1 and m_2 both have an even number of members and m_2 is a subset of m_1 then the multiset difference of m_1 and m_2 has an even number of members.

The fundamental theorem of arithmetic is a deep theorem involving several induction proofs. It seems quite likely that people have simply memorized this fact and use it freely. The other facts in the above list have simpler proofs (given the fundamental theorem of arithmetic). However, an explicit proof of any one of the above facts would be at least as long as the above proof that the square root of 2 is irrational. Furthermore, each of the above facts seems to be generally useful and thus it seems likely, or at least plausible, that people have memorized each of the above facts in addition to the fundamental theorem of arithmetic. People seem capable of using facts, such as the fundamental theorem, unconsciously; when reading the above natural argument one is not consciously aware of using the fundamental theorem of arithmetic. The above example suggests that people verify mathematical arguments by using knowledge they already have about the concepts involved and by applying that knowledge unconsciously in verifying the steps of the argument.

1.2 Ontic as a Formal Language

The Ontic system cannot read natural language—before an argument can be verified it must be translated into a machine readable form. The Ontic system manipulates formulas in the formal language called Ontic. The Ontic language is a syntactic sugar for first order set theory. The design of this syntactic sugar was driven by two motivations. First, the language is designed to be as similar as possible to natural language while still being simple and mathematically precise. Most atomic formulas in the Ontic language consists of a subject "noun phrase" and a predicate "verb phrase". In addition to being similar to natural language, the syntactic structure of the Ontic formal language facilitates the object-oriented inference mechanisms used in the system. Object-oriented inference is guided by a set of focus objects. The inference mechanisms "type" the focus objects—the system assigns a set of types to each focus object. In the Ontic system a type is any predicate of one argument; the types assigned to a focus object are predicates that are known to be true of that object. The syntax of the Ontic language is designed to facilitate this typing process; most atomic formulas state that a particular type applies to a particular object.

In the Ontic language there is no distinction between types, classes, sorts, and predicates of one argument. For an object x and type τ the phrases "τ contains x", "x is an instance of τ" and "τ is true of x" all mean the same thing. The word type is used, as opposed to the word class or predicate, because Ontic types are used in much the same way that types are used in computer programming languages; functions in the formal language can only be applied to arguments of the appropriate type and thus there is a distinction between "well-typed" and "ill-formed" expressions. For example, consider a function `TOPOLOGICAL-CLOSURE` such that if `X` is a topological space and `A` is a subset of `X` then

```
(TOPOLOGICAL-CLOSURE A X)
```

denotes the topological closure of `A` as a subset of `X`. An application of the operator `TOPOLOGICAL-CLOSURE` is well typed just in case its second argument denotes a topological space and its first argument denotes a subset of that space. The above expression is well typed but the expression

```
(TOPOLOGICAL-CLOSURE X A)
```

that results from reversing the arguments is not well typed because `A` is not a topological space and `X` need not be a subset of `A`.

Rather than give a rigorous syntax and semantics for the Ontic language, this section discusses the language informally and largely by example. A more rigorous treatment is presented in chapter 6. Every expression of the Ontic language belongs to exactly one of five syntactic categories; an expression is either a term, a formula, a function expression, a type expression, or a type generator expression. Terms are expressions that denote objects.[1] A formula is an expression which denotes one of the Boolean truth values *true* or *false*.[2] A function expression denotes a mapping from objects to objects. Each function expression takes a fixed number of arguments and returns an object.[3] Type expressions are predicates of one argument.[4] A type generator expression denotes a mapping from objects to types. Each type generator expression takes a fixed number of arguments and returns a type.[5]

1.2.1 Types

Figure 1.1 lists some type expressions. The first five type expressions in figure 1.1 are type symbols. The types `THING` and `SET` are primitive type symbols in the Ontic system. The Ontic system allows for the possibility that there are instances of the universal type `THING`, such as symbols, which are not instances of the type `SET`. Each of the types `GROUP`, `TOPOLOGICAL-SPACE`, and `RIEMANNIAN-MANIFOLD` can be defined in terms of more primitive concepts.

[1]A term is an expression of kind `OBJECT`. It is consistent with axioms of the logic to assume that all objects are actually sets in a standard model of ZFC set theory. However, it is more natural, and equally consistent, to assume that there exist objects which are not sets.

[2]A formula is an expression of kind `BOOLEAN`.

[3]Function expressions have kind `OBJECT` $\times$ `OBJECT` $\times \cdots \times$ `OBJECT` $\rightarrow$ `OBJECT`.

[4]Type expressions have kind `OBJECT` $\rightarrow$ `BOOLEAN`.

[5]Type generator expressions have kind `OBJECT` $\times$ `OBJECT` $\times \cdots \times$ `OBJECT` $\rightarrow$ `TYPE`.

THING, SET, GROUP, TOPOLOGICAL-SPACE, RIEMANNIAN-MANIFOLD

(MEMBER-OF s), (LOWER-BOUND-OF s p)

(LAMBDA ((x τ)) $\Phi(x)$)

(EITHER x y)

(AND-TYPE τ_1 τ_2)

(OR-TYPE τ_1 τ_2)

Figure 1.1: Ontic Type Expressions

The next two type expressions are types that result from applying type generators to arguments. If a term s denotes a set then (MEMBER-OF s) is a type expression such that an object is an instance of the type (MEMBER-OF s) just in case it is a member of the set s.[6] Instances of the type

(LOWER-BOUND-OF s p)

are members of the partially ordered set p which are lower bounds of the subset s of p. One place lambda predicates are also type expressions. The instances of the type

(LAMBDA ((x τ)) $\Phi(x)$)

consist of exactly those instances x of the type τ which satisfy the formula $\Phi(x)$. The type (EITHER X Y) contains only the instances X and Y. The type (AND-TYPE τ_1 τ_2) contains exactly those objects which are instances of both the types τ_1 and τ_2. The type (OR-TYPE τ_1 τ_2) contains exactly those things which are instances of either of the types τ_1 or τ_2.

1.2.2 Terms

Figure 1.2 gives some Ontic terms. There are several ways of constructing terms in Ontic. The application of a function to arguments is a term. If τ

[6]The term s denotes an object while the expression (MEMBER-OF s) denotes a type; no expression is allowed to be both a term and a type.

```
( fun x1 x2 ...)

(THE-SET-OF-ALL τ)

(THE-RULE fun)

(THE τ)

'symbol
```

Figure 1.2: Ontic Terms

is a "small" type expression then the expression (THE-SET-OF-ALL τ) is a term which denotes the set of all instances of τ. The process of converting a type to a set is called *reification* and sets of the form

(THE-SET-OF-ALL τ)

are often called *reified types.* It is important to remember that there is a syntactic distinction between terms (which denote objects) and type expressions (which denote predicates). There are types, such as the type THING, which can not be converted to sets—there is no set of all things. Most of the axioms of Zermelo-Fraenkel set theory state that certain sets exist. One can view these axioms as saying that certain types can be converted to sets. In the Ontic system these axioms of set theory are incorporated into the notion of a *syntactically small* type expression; the operator THE-SET-OF-ALL can only be applied to syntactically small type expressions. The notion of a syntactically small type expression, and the relation between this notion and the axioms of set theory, are discussed in more detail in chapter 6, section 6.1.

If *fun* is a function of one argument then the term (THE-RULE *fun*) denotes the "rule" that corresponds to the function. The relationship between functions and rules is analogous to the relationship between types and sets—the expression (THE-RULE *fun*) is a term and denotes an object while *fun* is a function expression. Expressions of the form (THE-RULE *fun*) are often referred to as *reified functions.* There exist functions which can not be reified as rules, e.g any function defined on all sets, such as the function that maps an arbitrary set to its power set, is too big to be reified as a rule.

If τ is a type with exactly one instance then the expression (THE τ) is a term which denotes the single object contained in the type. For example, if

(PRIME-NUMBER-BETWEEN n m)

is a type whose instances are the prime numbers between n and m then

(THE (PRIME-NUMBER-BETWEEN 20 25))

denotes the number 23.

Expressions of the form *'symbol* are also terms. For example the expression 'FOO denotes the symbol FOO. Quoted symbols denote objects which are instances of the type SYMBOL. The Ontic system allows for the possibility that all objects are sets, i.e. that every object is an element of a model of Zermelo-Fraenkel set theory. However, the Ontic system also allows for a more natural interpretation under which rules and symbols are not sets—the types SET, RULE, and SYMBOL can be assumed to be disjoint.

1.2.3 Formulas

Figure 1.3 gives some Ontic formulas. The formula (IS x τ) is true just in case x denotes an instance of the type τ. Formulas of this form are intuitively pleasing because they seem to reflect natural language syntax—x is a subject "noun phrase" and the type τ is a predicate that applies to the subject. The formula (EXISTS-SOME τ) is true just in case there exists an instance of τ. The formula

(EXISTS ((x_1 τ_1) (x_2 τ_2) ...) $\Phi(x_1, x_2, \ldots)$)

is true just in case there exists instances $a_1, a_2 \ldots a_n$ of the types $\tau_1, \tau_2, \ldots \tau_3$ respectively such that such that Φ is true when the variables $x_1, x_2, \ldots x_n$ are interpreted as $a_1, a_2 \ldots a_n$ respectively. The formula

(FORALL ((x_1 τ_1) (x_2 τ_2) ...) $\Phi(x_1, x_2, \ldots)$)

has the obvious analogous meaning. The formula (EXACTLY-ONE τ) is true just in case there is exactly one instance of the type τ. The formula

(IS-EVERY τ_1 τ_2)

```
(IS x τ)
(EXISTS-SOME τ)
(EXISTS ((x₁ τ₁) (x₂ τ₂) ...) Φ(x₁, x₂, ...))
(FORALL ((x₁ τ₁) (x₂ τ₂) ...) Φ(x₁, x₂, ...))
(EXACTLY-ONE τ)
(IS-EVERY τ₁ τ₂)
(NOT Φ)
(AND Φ₁ Φ₂)
```

Figure 1.3: Ontic Formulas

is true just in case every instance of τ_1 is an instance of τ_2. Of course Boolean combinations of formulas are also formulas.

1.2.4 Definitions

Figure 1.4 gives some examples of definitions of functions and type generators. Functions are defined with the `DEFTERM` construct as shown in the first example. In the first example the function `POWER-SET` is defined to be equivalent to the lambda function

```
(LAMBDA ((S SET)) (THE-SET-OF-ALL (SUBSET-OF S)))
```

Thus the function `POWER-SET` takes one argument which must be a set and returns the set of all subsets of that set. Types and type generators are defined with the `DEFTYPE` construct. The second definition in figure 1.4 defines `LOWER-BOUND-OF` to be a type generator which takes two arguments: a set s and a poset p where the set s is required to be a subset of the set of elements of p. The type generator `LOWER-BOUND-OF` takes these arguments and returns a type: a predicate of one argument. An object x is an element of the type (`LOWER-BOUND-OF` s p) just in case x is an element of the underlying set of the poset p and every member of the set s is greater than or

```
(DEFTERM (POWER-SET (S SET))
  (THE-SET-OF-ALL (SUBSET-OF S)))

(DEFTYPE (LOWER-BOUND-OF
            (S (SUBSET-OF (U-SET P)))
            (P POSET))
  (LAMBDA ((X (MEMBER-OF (U-SET P))))
    (IS-EVERY (MEMBER-OF S)
              (GREATER-OR-EQUAL-TO X P))))

(DEFTYPE (GREATEST-LOWER-BOUND-OF
            (S (SUBSET-OF (U-SET P)))
            (P POSET))
  (LAMBDA ((X (LOWER-BOUND-OF S P)))
    (IS-EVERY (LOWER-BOUND-OF S P)
              (LESS-OR-EQUAL-TO X P))))

(DEFTYPE COMPLETE-LATTICE
  (LAMBDA ((P POSET))
    (FORALL ((S (SUBSET-OF (U-SET P))))
      (EXISTS-SOME (GREATEST-LOWER-BOUND-OF S P)))))
```

Figure 1.4: Some Ontic Definitions

equal to x under the ordering imposed by the poset p. The type generator GREATEST-LOWER-BOUND-OF is similar to LOWER-BOUND: it takes a set s and a poset p where s is a subset of the underlying set of p and yields a type. An object x is an element of the type (GREATEST-LOWER-BOUND-OF s p) just in case x is a lower bound of s in the poset p and every lower bound of s in p is greater or equal to x. The type COMPLETE-LATTICE is defined so that an object p is of type COMPLETE-LATTICE just in case p is a poset such that for every subset s of the underlying set of p there exists a greatest lower bound of s under the ordering imposed by p.

The type restrictions on the formal parameters of functions and type generators determine a distinction between well-typed and ill-formed expressions. The Ontic system will not invoke the definition of a function or type generator unless the arguments to the function or type generator have been proven to be of the correct type; the Ontic system effectively type-checks expressions before it expands definitions. Given the expressive power of the Ontic type system, however, one can easily show that there are well-typed expressions which fail to type check. In the Ontic system type checking involves theorem proving based on a lemma library. Many of the lemmas of the lemma library state that certain objects have certain types; not surprisingly, such lemmas play an important role in determining if an expression is well typed. It is often the case that a given expression fails to type check using one lemma library but succeeds in type checking given a stronger lemma library.

1.2.5 Summary

In addition to providing a distinction between well-typed and ill-formed expressions, the Ontic type vocabulary seems to allow for concise and natural formal statements. For example the IS-EVERY phrase constructor allows the concise expression of statements that would normally require explicit quantification. Similarly, the EXISTS-SOME phrase constructor uses the type vocabulary to make concise existential statements. Types are also used directly by the phrase constructors THE-SET-OF-ALL, THE, and EXACTLY-ONE.

The definitions in figure 1.4 should provide an indication of the conciseness and expressive power of the Ontic language. Jonathan Rees spent about a month defining various mathematical concepts in Ontic. Starting with only the fundamental notions described above, he used the Ontic language to formally define groups, rings, ideals in a ring, fields, the natural numbers, the real numbers (defined both as a totally ordered complete field and as Dedekind cuts), topological spaces, continuous functions, homotopy of maps between topological spaces, the fundamental group of a topological space, differentiable functions on the reals, the derivative of a function, the notion of a category and products and limits in arbitrary categories. The ease with which Rees expressed these concepts suggests that any mathematical concept can be readily expressed in Ontic.

1.3 Examples of Verification

Object-oriented inference operates in a context. A context consists of three things: a lemma library, a set of focus objects and set of suppositions about the focus objects. Figure 1.5 gives a block diagram of the object-oriented inference mechanisms used in the Ontic system. The inference process is forward chaining; it draws conclusions from the lemma library without being given any goal formula. It is well known that unrestricted forward chaining from a large lemma library leads to an immediate combinatorial explosion — vast numbers of formulas are generated where each formula can be derived from the given lemmas in only a few steps. The forward chaining inference mechanisms used in the Ontic system, however, are guided by the focus objects. The focus objects are Ontic terms, expressions that denote objects. The system restricts its inference process to formulas that are in some sense "about" the focus objects. There are four basic inference mechanisms: Boolean constraint propagation, congruence closure, focused binding (also called semantic modulation), and automatic universal generalization. The first two inference mechanisms are well known inference procedures for the quantifier-free predicate calculus with equality. The last two inference mechanisms are unique to the Ontic system. These four inference mechanisms are discussed in section 1.4 and again in more detail in chapters 4 and 5. In a given context the four forward chaining inference mechanisms generate a set of formulas about the focus objects called "obvious truths".

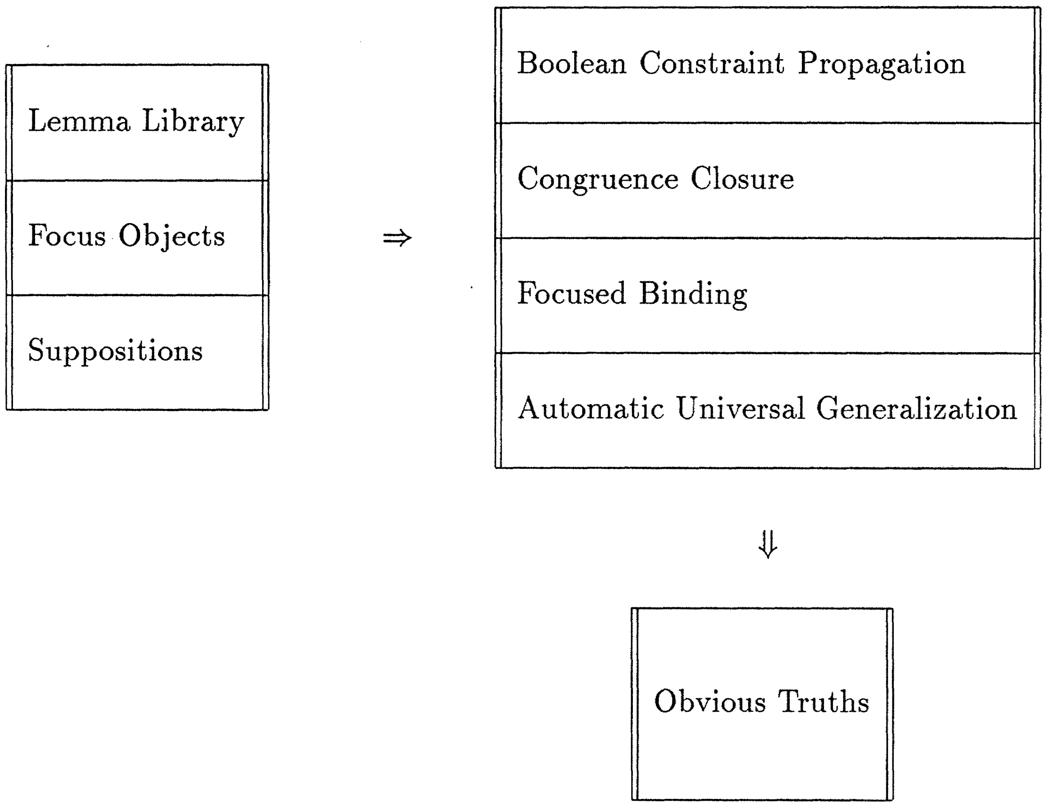
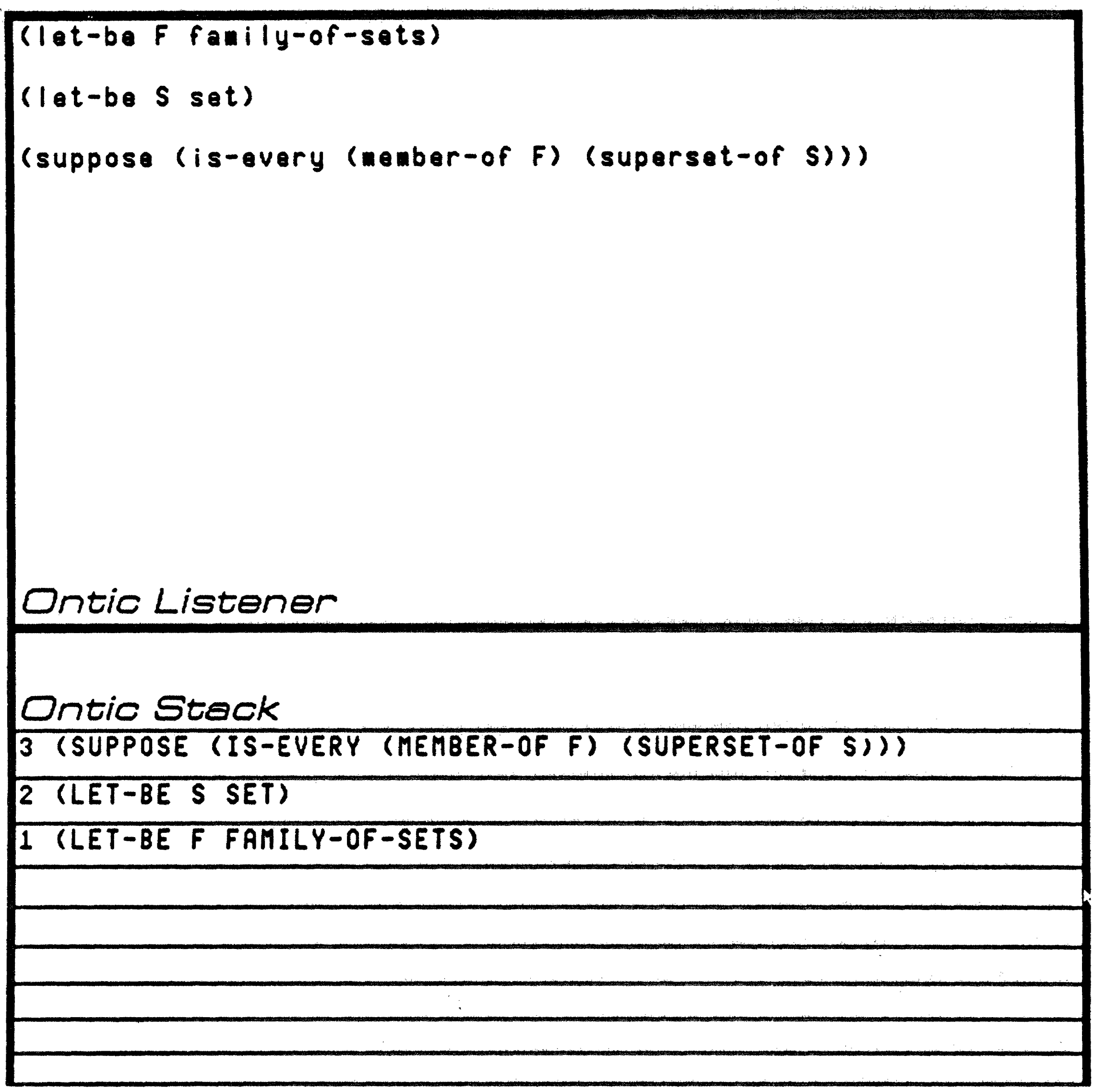


Figure 1.5: A Block Diagram of Object-Oriented Inference



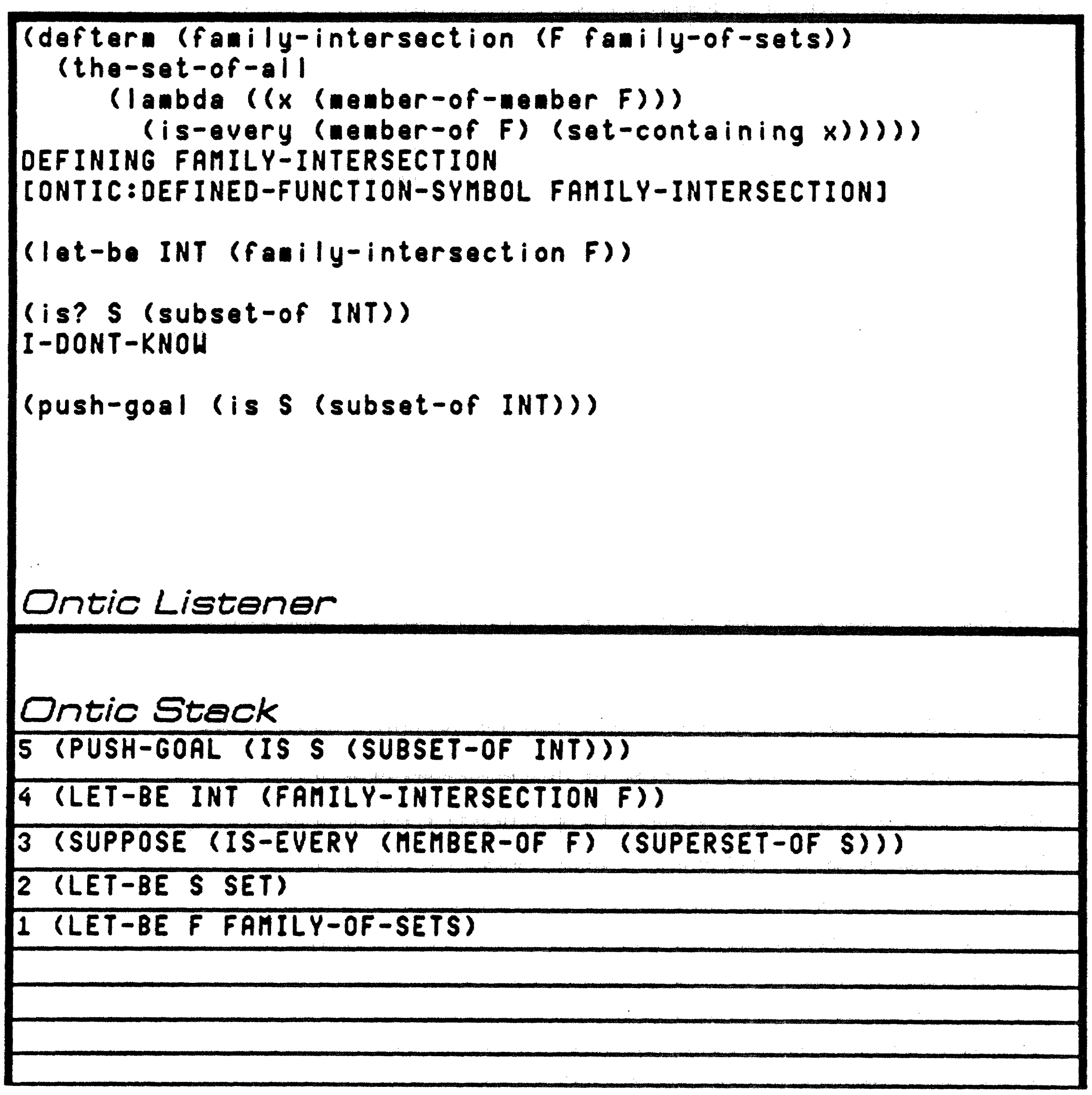
```
(let-be F family-of-sets)

(let-be S set)

(suppose (is-every (member-of F) (superset-of S)))
```

```
3 (SUPPOSE (IS-EVERY (MEMBER-OF F) (SUPERSET-OF S)))
2 (LET-BE S SET)
1 (LET-BE F FAMILY-OF-SETS)
```

Figure 1.6: The Ontic Interpreter Display

The Ontic interpreter is an interactive system for verifying proofs. Each step in an argument is associated with a context, i.e. a set of focus objects, a set of suppositions about the focus objects and the current lemma library. The user tells the system when to enter new contexts, when to leave old contexts, and when to "note" a fact that has been established in a given context. Figure 1.6 shows the display of the Ontic interpreter as seen by a user who is about to verify a fact concerning families of sets. The top half of the display is a Lisp listener: a window for interacting with a Lisp interpreter. The bottom half of the display shows the *context stack* which displays the set of suppositions and focus objects for the current context. In the example shown in figure 1.6 the user first instructs the system to let `F` be a family of sets. This caused the system to enter a context in which it is focusing on an arbitrary family of sets denoted by `F`. The user then instructs the system to let `S` be any set. This causes the system to enter a context where it is focusing on an arbitrary set `S`. Finally the user instructs the system to suppose that every set in the family `F` is a superset (i.e. contains) the set `S`. Each time a new context is entered, the instruction for entering that context is pushed onto the context stack shown in the bottom half of the display. By looking at the context stack display one can determine the set of focus objects and suppositions that are currently active.

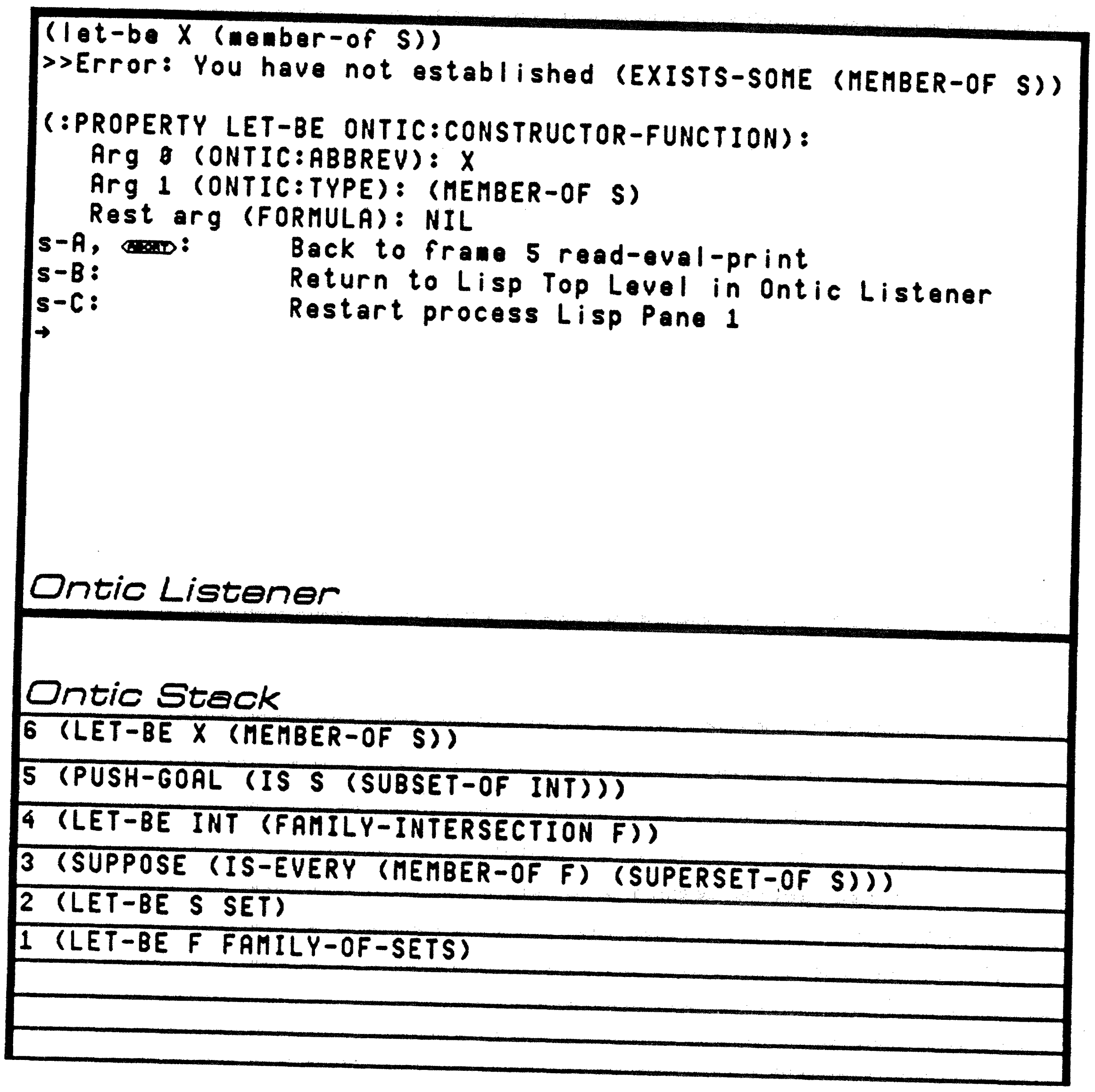
Figures 1.7 through 1.13 show successive stages in the verification of a simple fact concerning families of sets. Let `F` be a family of sets, let `S` be a set and suppose that every member of the family `F` contains the set `S`. Figures 1.7 through 1.13 present an argument showing that the set `S` must be a subset of the intersection of the members of the family `F`. Figure 1.7 shows the definition of the function `FAMILY-INTERSECTION` which takes a family of sets and returns the intersection of all its members. In Figure 1.7 the user asks the system to abbreviate the term `(FAMILY-INTERSECTION F)` with the symbol `INT`. This causes the intersection `INT` to become a focus object. The user then asks the system if the set `S` is a subset of `INT` and the system says it doesn't know. The user then states that the formula `(IS S (SUBSET-OF INT))` is a goal to be proven. This last instruction has no effect on the context; the system is not goal directed and ignores goals which appear on the context stack. Goals act as comments which improve the readability of proofs (the written form of proofs will be discussed later).



```
(defterm (family-intersection (F family-of-sets))
  (the-set-of-all
     (lambda ((x (member-of-member F)))
       (is-every (member-of F) (set-containing x)))))
DEFINING FAMILY-INTERSECTION
[ONTIC:DEFINED-FUNCTION-SYMBOL FAMILY-INTERSECTION]

(let-be INT (family-intersection F))

(is? S (subset-of INT))
I-DONT-KNOW

(push-goal (is S (subset-of INT)))
```

```
5 (PUSH-GOAL (IS S (SUBSET-OF INT)))
4 (LET-BE INT (FAMILY-INTERSECTION F))
3 (SUPPOSE (IS-EVERY (MEMBER-OF F) (SUPERSET-OF S)))
2 (LET-BE S SET)
1 (LET-BE F FAMILY-OF-SETS)
```

Figure 1.7: Statement of a New Lemma to be Proved



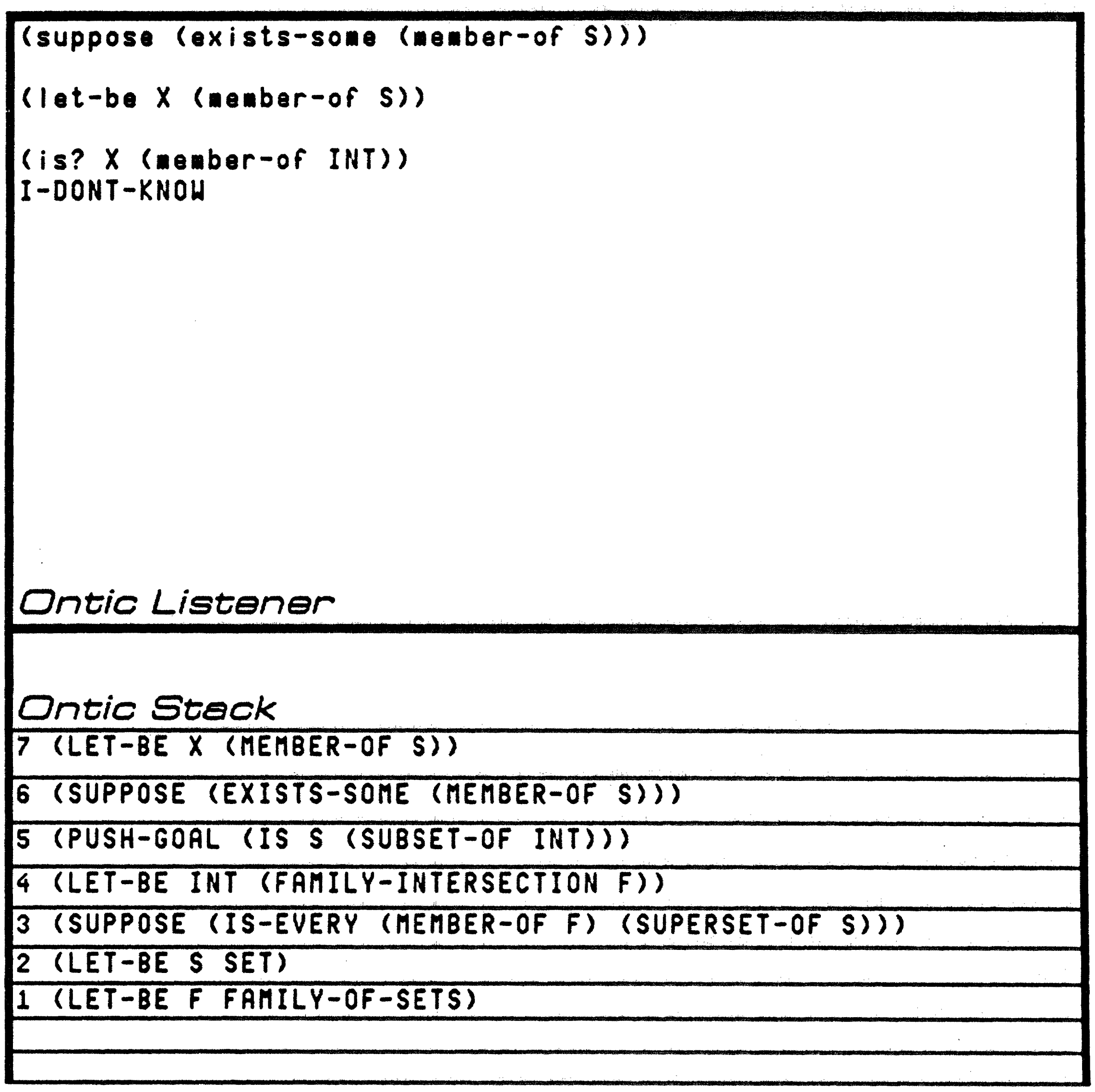
```
(let-be X (member-of S))
>>Error: You have not established (EXISTS-SOME (MEMBER-OF S))

(:PROPERTY LET-BE ONTIC:CONSTRUCTOR-FUNCTION):
   Arg 0 (ONTIC:ABBREV): X
   Arg 1 (ONTIC:TYPE): (MEMBER-OF S)
   Rest arg (FORMULA): NIL
s-A, ABORT:     Back to frame 5 read-eval-print
s-B:            Return to Lisp Top Level in Ontic Listener
s-C:            Restart process Lisp Pane 1
→
```

Ontic Listener

Ontic Stack

```
6 (LET-BE X (MEMBER-OF S))
5 (PUSH-GOAL (IS S (SUBSET-OF INT)))
4 (LET-BE INT (FAMILY-INTERSECTION F))
3 (SUPPOSE (IS-EVERY (MEMBER-OF F) (SUPERSET-OF S)))
2 (LET-BE S SET)
1 (LET-BE F FAMILY-OF-SETS)
```

Figure 1.8: A Failed Instruction to the Interpreter



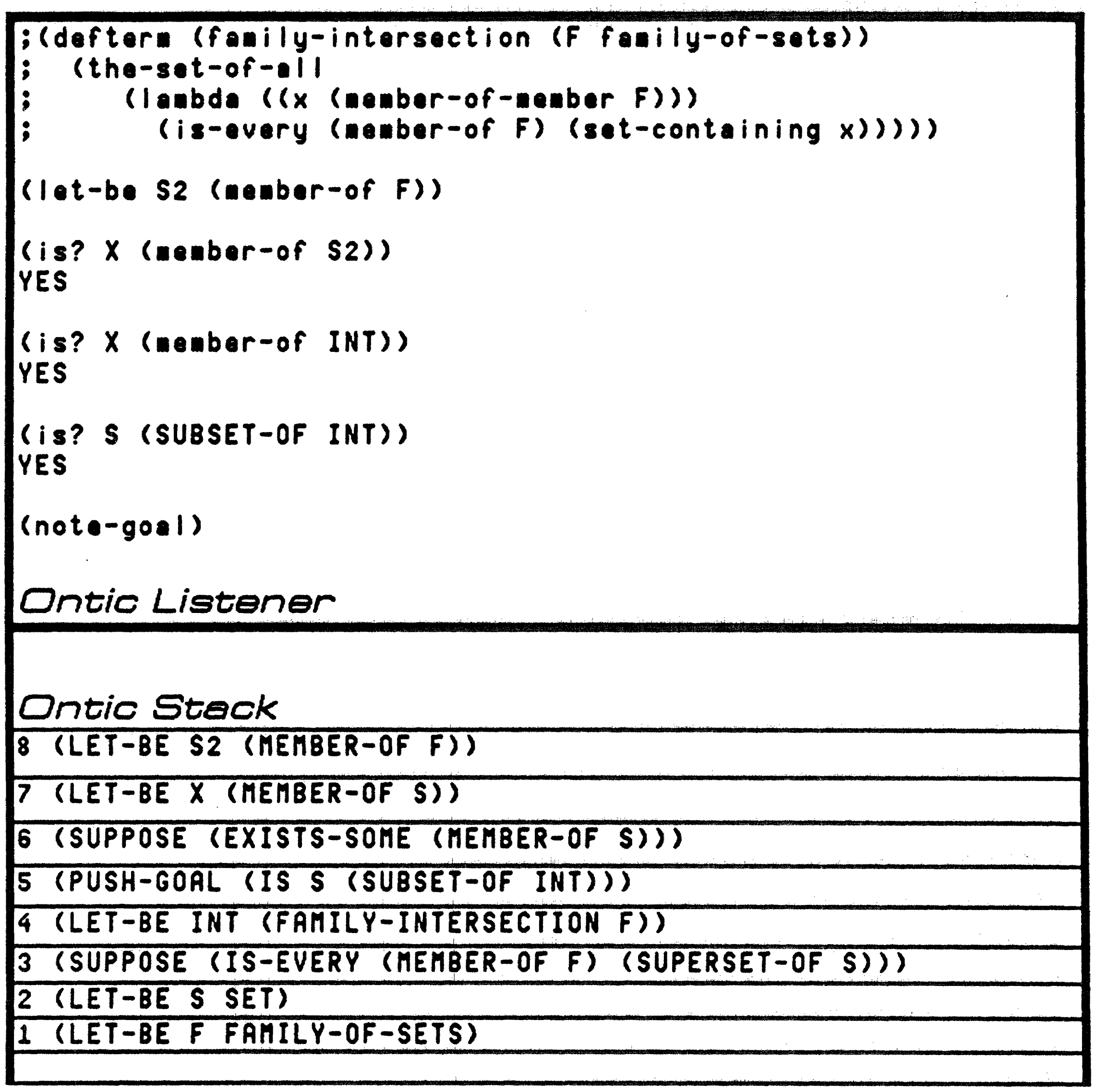
```
(suppose (exists-some (member-of S)))

(let-be X (member-of S))

(is? X (member-of INT))
I-DONT-KNOW
```

Ontic Listener

Ontic Stack

```
7 (LET-BE X (MEMBER-OF S))
6 (SUPPOSE (EXISTS-SOME (MEMBER-OF S)))
5 (PUSH-GOAL (IS S (SUBSET-OF INT)))
4 (LET-BE INT (FAMILY-INTERSECTION F))
3 (SUPPOSE (IS-EVERY (MEMBER-OF F) (SUPERSET-OF S)))
2 (LET-BE S SET)
1 (LET-BE F FAMILY-OF-SETS)
```

Figure 1.9: Supposing the Existence of Objects of Certain Kind



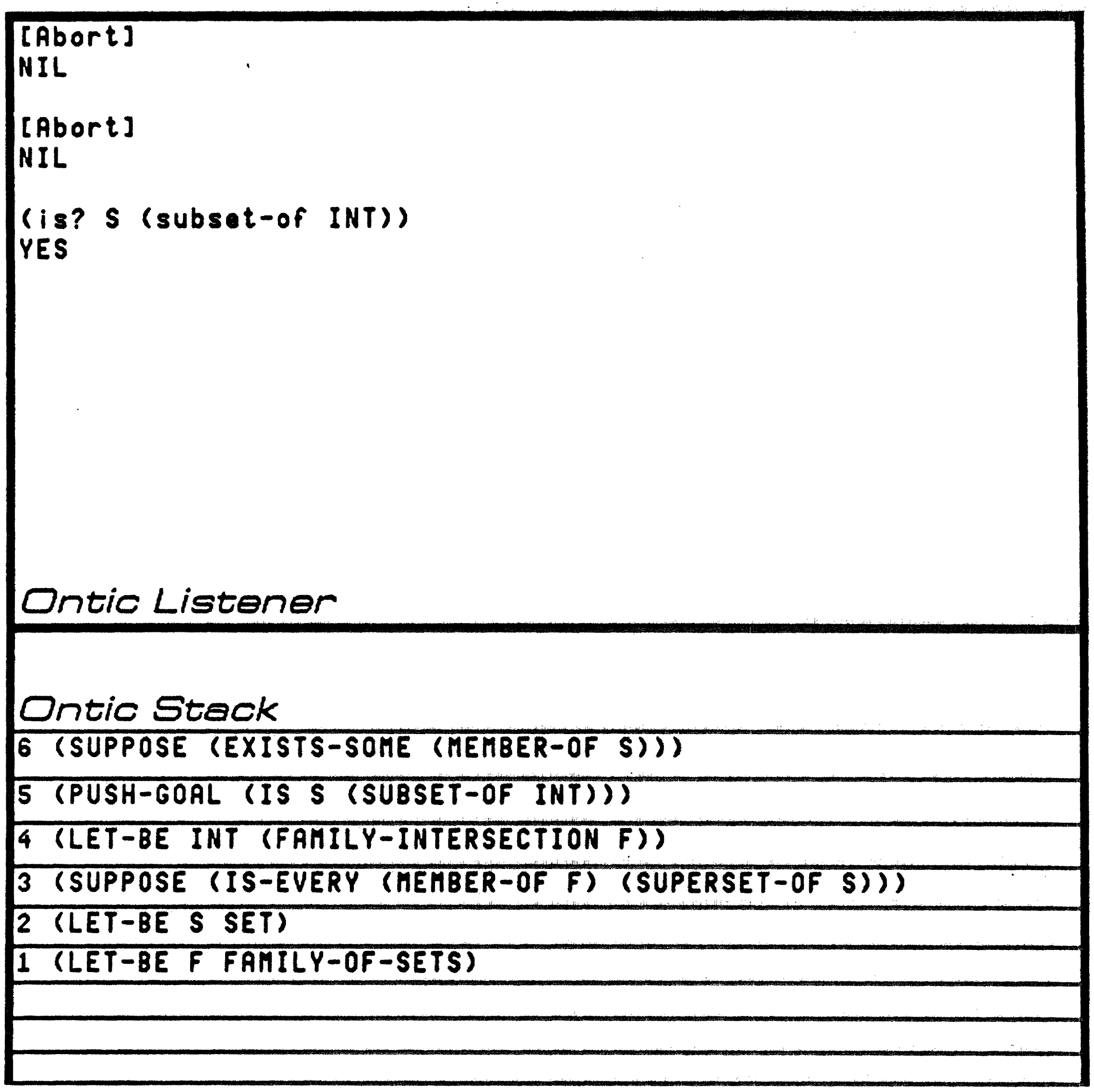
```
;(defterm (family-intersection (F family-of-sets))
;  (the-set-of-all
;     (lambda ((x (member-of-member F)))
;       (is-every (member-of F) (set-containing x)))))

(let-be S2 (member-of F))

(is? X (member-of S2))
YES

(is? X (member-of INT))
YES

(is? S (SUBSET-OF INT))
YES

(note-goal)
```

Ontic Listener

Ontic Stack

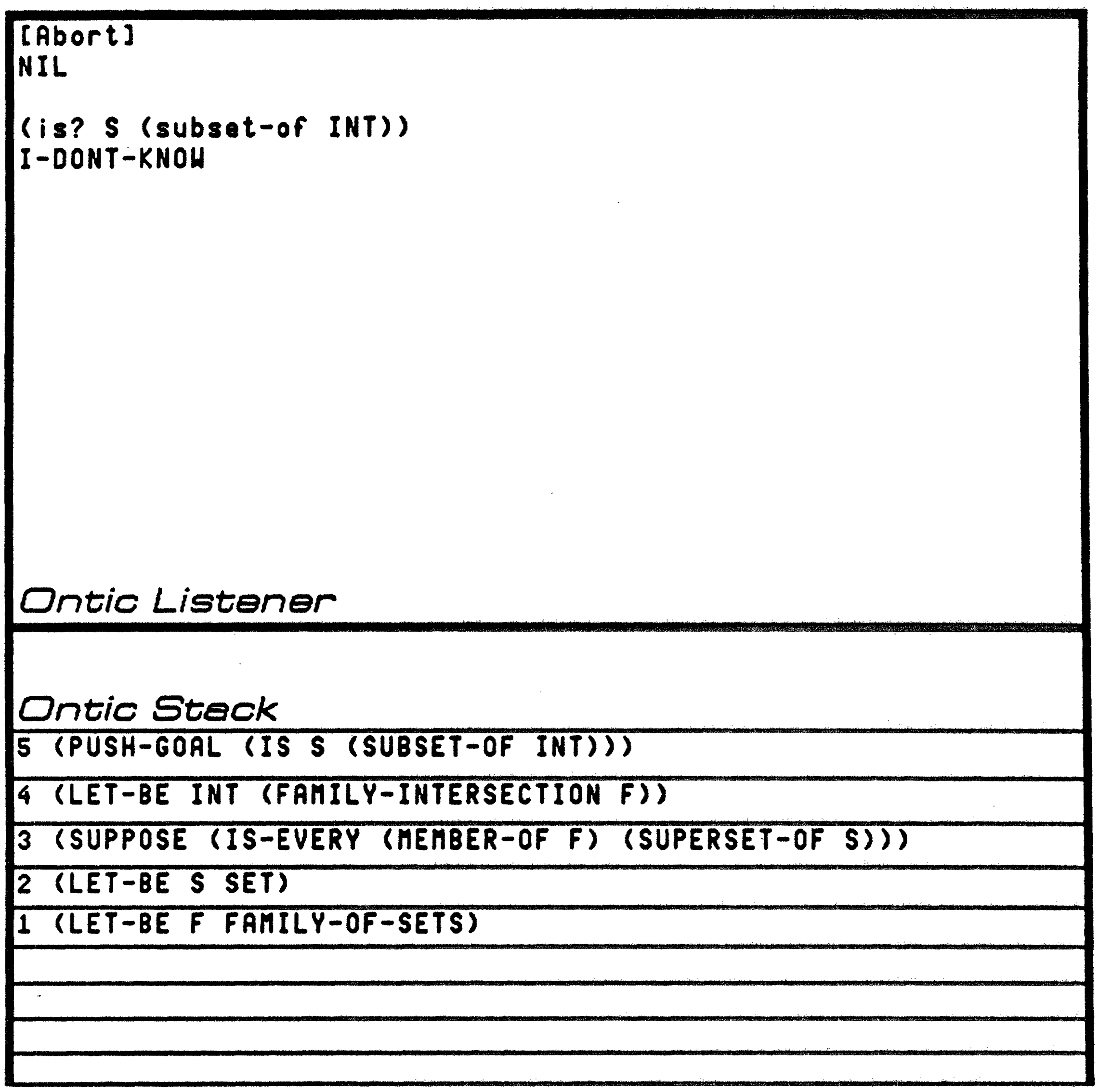
```
8 (LET-BE S2 (MEMBER-OF F))
7 (LET-BE X (MEMBER-OF S))
6 (SUPPOSE (EXISTS-SOME (MEMBER-OF S)))
5 (PUSH-GOAL (IS S (SUBSET-OF INT)))
4 (LET-BE INT (FAMILY-INTERSECTION F))
3 (SUPPOSE (IS-EVERY (MEMBER-OF F) (SUPERSET-OF S)))
2 (LET-BE S SET)
1 (LET-BE F FAMILY-OF-SETS)
```

Figure 1.10: Establishing the Goal in a Certain Context


```
[Abort]
NIL

[Abort]
NIL

(is? S (subset-of INT))
YES
```

```
6 (SUPPOSE (EXISTS-SOME (MEMBER-OF S)))
5 (PUSH-GOAL (IS S (SUBSET-OF INT)))
4 (LET-BE INT (FAMILY-INTERSECTION F))
3 (SUPPOSE (IS-EVERY (MEMBER-OF F) (SUPERSET-OF S)))
2 (LET-BE S SET)
1 (LET-BE F FAMILY-OF-SETS)
```

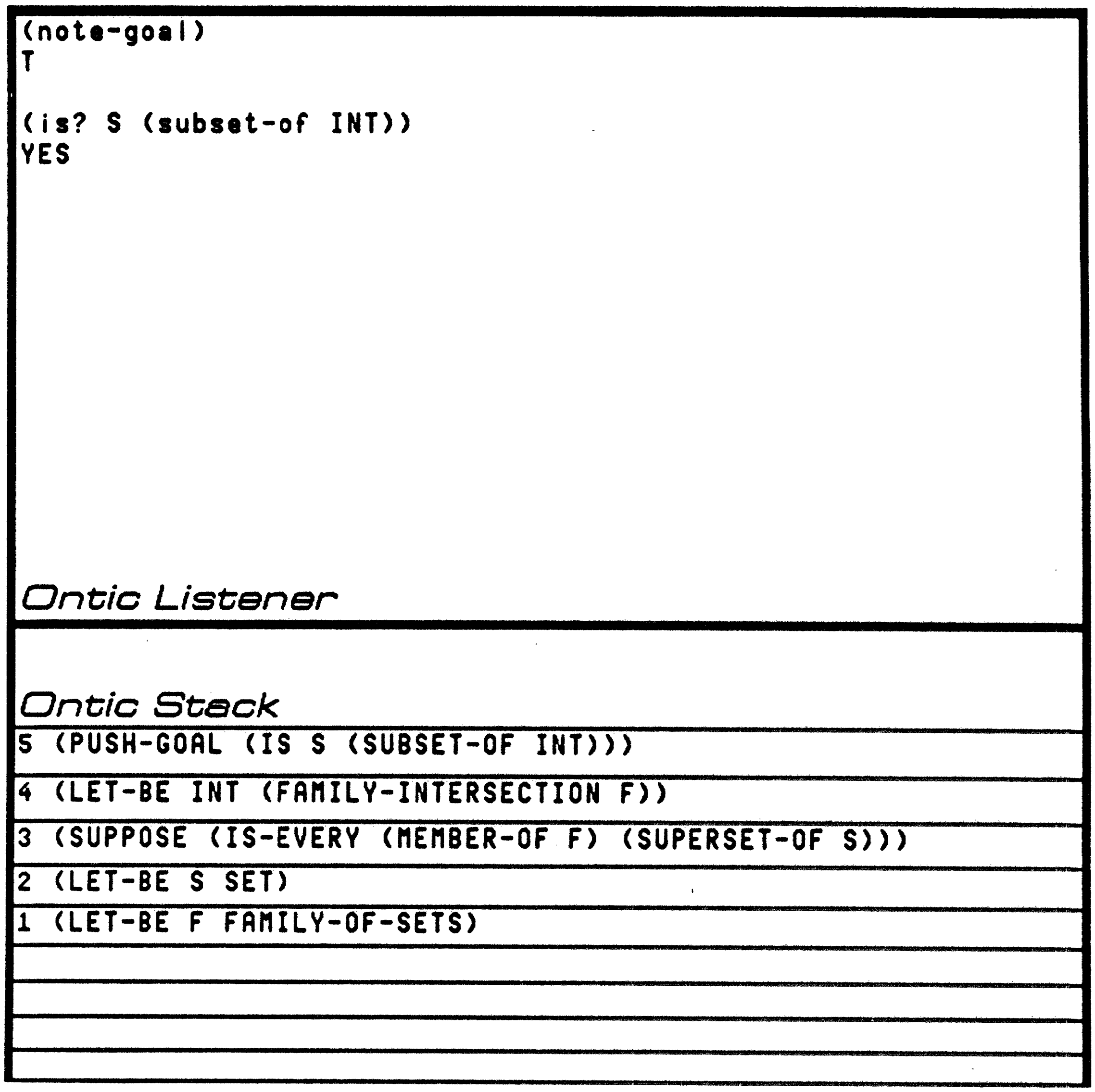
Figure 1.11: Bringing the Result Back to an Earlier Context



```
[Abort]
NIL

(is? S (subset-of INT))
I-DONT-KNOW
```

```
5 (PUSH-GOAL (IS S (SUBSET-OF INT)))
4 (LET-BE INT (FAMILY-INTERSECTION F))
3 (SUPPOSE (IS-EVERY (MEMBER-OF F) (SUPERSET-OF S)))
2 (LET-BE S SET)
1 (LET-BE F FAMILY-OF-SETS)
```

Figure 1.12: The Result Does Not Move Past Relevant Suppositions


```
(note-goal)
T

(is? S (subset-of INT))
YES
```

Ontic Listener

Ontic Stack

```
5 (PUSH-GOAL (IS S (SUBSET-OF INT)))
4 (LET-BE INT (FAMILY-INTERSECTION F))
3 (SUPPOSE (IS-EVERY (MEMBER-OF F) (SUPERSET-OF S)))
2 (LET-BE S SET)
1 (LET-BE F FAMILY-OF-SETS)
```

Figure 1.13: A Simple Automatic Refutation Finishes the Proof

To show that the set S is a subset of INT we must show that every member of S is a member of INT. To do this we can consider some arbitrary member X of the set S. In figure 1.8 the user tells the system to do so. However, the system complains that we have not yet established that such members exist; the set S might be empty. In general the system ensures that every object being considered is known to exist. In order to consider an arbitrary member of the set S we must first assume that such members exist. In figure 1.9 the user first instructs the system to suppose that there are members of the set S and then he instructs the system to consider a particular (but arbitrary) member X. The user then asks the system if X is a member of INT and the system doesn't know. At this point the user may be mystified as to why the system does not "see" the obvious fact that X is indeed a member of the family intersection INT. Before proceeding further, the user reviews the definition of the function FAMILY-INTERSECTION as shown in figure 1.10. This definition states that X is a member of the family intersection just in case X is a member of every set in the family F. In figure 1.10 the user shows that X is a member of the intersection INT by showing that X is a member of an arbitrary set S2 in the family F. This is done by considering an arbitrary member S2 of the family F. In this scenario, instances of the type FAMILY-OF-SETS are by definition non-empty and thus we do not need the additional assumption that F is non-empty. When the system focuses on the member S2 of the family F it "sees" that because X is a member of S, and S is a subset of S2, X is a member of S2. At this point the system performs an automatic universal generalization. Since S2 is an arbitrary member of F, and since X has been shown to be a member of S2, it follows that X is a member of every member of F. Furthermore since X is an arbitrary member of S the system can perform yet another automatic universal generalization and conclude that all members of S must be members of INT and thus S is a subset of INT. Asking the system a question has no effect on the state of the system; the questions shown in figure 1.10 serve only to indicate the line of reasoning used by the system. The problem was actually solved by forward chaining as soon as the last context was entered.

The forward chaining inference mechanisms establish the goal in the context shown in figure 1.10. In order to remember that the goal has been proven, the system must update the underlying lemma library. More specifically, if the lemma library were not updated, then when the user returned to a previous context, nothing would have been learned; the set of "obvious truths" in a context is determined by the lemma library, the focus objects and the suppositions. In the scenario shown in figure 1.10 the user explicitly updates the lemma library by calling the function NOTE-GOAL. In this case the system adds the following lemma:

```
(FORALL ((F FAMILY-OF-SETS)
         (S SET))
  (=> (AND (IS-EVERY (MEMBER-OF F) (SUPERSET-OF S))
           (EXISTS-SOME (MEMBER-OF S)))
      (IS S (SUBSET-OF (FAMILY-INTERSECTION F)))))
```

In any context, the user can instruct the system to note any formula that is obviously true in that context. The function NOTE-GOAL is just an abbreviation for noting the latest goal which has been pushed onto the context stack; the same effect would have been achieved if the user had typed

```
(NOTE (IS S (SUBSET-OF INT)))
```

When a formula is noted the system constructs the implication which states that suppositions active in the current context imply the noted formula. The system then adds the universal closure of that implication to the permanent lemma library. Note that in this case we have not really proven the desired lemma; we have only proven it for the case where the set S is non-empty.

Figure 1.11 shows that with the updated lemma library, the desired result is "obvious" in the context associated with stack frame 6. However, the result must still be proven for the case where S is empty; figure 1.12 shows that the result has not yet been established at stack frame 5. But the case for the empty set is trivial, and in figure 1.13 the user simply asks the system to note the goal. Since the goal is not known directly at frame 5, the system does a refutation proof; it enters a context where the goal is assumed to be false. Given the new lemma shown above, the forward chaining inference mechanisms are able to derive a contradiction from the negation of the goal, and thus the goal is established by refutation. Thus the note-goal in figure 1.13 has the effect of adding the following lemma to the lemma library.

```
(FORALL ((F FAMILY-OF-SETS)
         (S SET))
  (=> (IS-EVERY (MEMBER-OF F)
                (SUPERSET-OF S))
      (IS S (SUBSET-OF (FAMILY-INTERSECTION F)))))
```

The "proof" shown in figures 1.7 through 1.13 is automatically recorded by the system; Figure 1.14 shows an automatically generated textual representation of the complete proof. Evaluating the form shown in figure 1.14 with the Lisp interpreter causes the above two lemmas to be proved and added to the lemma library. (The second lemma makes the first one obsolete and the user can, if he wishes, explicitly delete the first lemma after the proof has been done.)

The textual representation of proofs involves `IN-CONTEXT` expressions. In general an `IN-CONTEXT` expression is composed of two parts: a "context definition" and a body; the context definition specifies the construction of a new context by giving a list of context-constructing instructions. The body is a list of instructions to be executed in the specified context. The body of an `IN-CONTEXT` expression may contain embedded `IN-CONTEXT` expressions. Embedded contexts inherit the focus objects and suppositions of outer contexts.

The two note-goal expressions in figure 1.14 correspond to the case analysis performed in the interactive proof. The first note-goal notes that if there exists a member of `S` then the theorem is true. The second note-goal invokes a refutation proof which effectively handles the case where `S` is empty. In general multiple note-goals for the same goal correspond to a case analysis. Often, as in this example, the context for the last case does not need to be explicitly constructed because an automatic refutation process initiated by the last note-goal effectively constructs the context for the last case.

The Ontic interpreter is able to use a large lemma library without human assistance; the system automatically applies facts from the lemma library whenever it enters a new context. Figure 1.15 shows the lemma established by the proof in figure 1.14 together with two other facts: for every family of sets `F`, every member of `F` contains (as a subset) the family intersection of `F`; and, for two sets, if each is a subset of the other, then the two sets

```
(IN-CONTEXT ((LET-BE F FAMILY-OF-SETS)
             (LET-BE S SET)
             (SUPPOSE (IS-EVERY (MEMBER-OF F)
                                (SUPERSET-OF S)))
             (LET-BE INT (FAMILY-INTERSECTION F))
             (PUSH-GOAL (IS S (SUBSET-OF INT))))

  (IN-CONTEXT ((SUPPOSE (EXISTS (MEMBER-OF S)))
               (LET-BE X (MEMBER-OF S))
               (LET-BE S2 (MEMBER-OF F)))
    (NOTE-GOAL))

  (NOTE-GOAL))
```

Figure 1.14: The History

```
(FORALL ((F FAMILY-OF-SETS)
         (S SET))
     (=> (IS-EVERY (MEMBER-OF F)
                   (SUPERSET-OF S))
         (IS S (SUBSET-OF (FAMILY-INTERSECTION F)))))

(FORALL ((F FAMILY-OF-SETS)
         (S (MEMBER-OF F)))
    (IS (FAMILY-INTERSECTION F)
        (SUBSET-OF S)))

(FORALL ((S1 SET)
         (S2 SET))
    (=> (AND (IS S1 (SUBSET-OF S2))
             (IS S2 (SUBSET-OF S1)))
        (= S1 S2)))
```

Figure 1.15: Some Simple Facts

```
(IN-CONTEXT ((LET-BE S SET)
             (LET-BE S2 (SUBSET-OF S))
             (LET-BE F (THE-SET-OF-ALL
                         (AND-TYPE (SUBSET-OF S)
                                   (SUPERSET-OF S2))))
   (IN-CONTEXT ((PUSH-GOAL (= S2 (FAMILY-INTERSECTION F))))
     (IN-CONTEXT ((LET-BE INT (FAMILY-INTERSECTION F))
                  (LET-BE S3 (MEMBER-OF F)))
       (NOTE-GOAL))))
```

Figure 1.16: A Proof Using Lemmas

are equal. Figure 1.16 is a proof which makes use of the facts in figure 1.15. We assume that the lemmas in figure 1.15 have been placed in the lemma library and are therefore available to the Ontic interpreter. The proof in figure 1.16 goes as follows: Let S be any set and let S2 be any subset of S. Let F be the set of all subsets of S which contain the set S2. We wish to show that the family intersection of F equals the set S2. First the user focuses on the family intersection of F by abbreviating this intersection with the symbol INT. Next the user focuses on an arbitrary member of the family F. Focusing on arbitrary member of F causes the system to "realize" various facts about F. For example every member of F is a set and thus F is a family of sets. By proving that F is a family of sets the system establishes that the term (FAMILY-INTERSECTION F) is well typed and thus the definition of FAMILY-INTERSECTION can be invoked. Furthermore S3 is a superset of S2 so S2 is a subset of S3 and by universal generalization S2 is a subset of every member of F. Once the system deduces that F is a family of sets and every member of F is a set which contains S2 the system automatically applies the first lemma in figure 1.16 and realizes that S2 is a subset of the intersection INT. The system also realizes that the set S2 is a member of the family F and applies the the second lemma in figure 14 thus realizing that the intersection INT is a subset of S2. Finally the system applies the the third fact in figure 1.15 and realizes that INT equals S2.

Actually the Ontic interpreter makes no distinction between definitions and lemmas; definitions are just universally quantified equations which are accessed in the same manner as lemmas. The proof shown in figure 1.16 relies on definitions as well as the lemmas shown in figure 1.15. The proof shown in figure 1.14 does not involve any previously proven lemmas but it does involve the definition of the intersection of a family of sets.

In general, the user need not make explicit references to definitions and lemmas. The user relies on the system to use definitions and lemmas whenever they are appropriate. For example, consider an arbitrary lemma of following form:

$$\texttt{(FORALL ((}x\ \tau_1\texttt{) (}y\ \tau_2\texttt{))}\ \Phi(x,\ y)\texttt{)}$$

This "lemma" might actually be a definition in which case Φ is an equation or logical equivalence. The Ontic system will automatically use this lemma in any context where there are two focus objects `A` and `B` such that `A` is an instance of τ_1 and `B` is an instance of τ_2. In general, a universally quantified lemma such as the one shown above will be instantiated with all combinations of focus objects that match the type restrictions of the lemma. Once the lemmas have been instantiated with the focus objects, the system applies the forward chaining inference techniques of Boolean constraint propagation, congruence closure, and automatic universal generalization. The instantiation process that invokes facts from the lemma library is a graph-theoretic marker-propagation inheritance mechanism called *focused binding* or *semantic modulation.* The focused binding mechanism achieves the effect of instantiation but avoids constructing the formulas that result from the syntactic substitutions done by normal instantiation.

One way of measuring the performance of a verification system is to compare the length of a natural argument with the length of a corresponding machine readable proof. The ratio of the length of a machine readable proof to the length of the corresponding natural argument is called the *expansion factor* for that proof. Figure 1.17 shows both an English natural argument (taken from a textbook on lattice theory, [Gratzer 78] page 24) and a corresponding Ontic proof. The natural argument contains 75 words and mathematical symbols, while the Ontic proof contains 73 symbols, yielding a word count expansion factor of about one. For the most part the "clear and necessary" steps of this particular natural argument correspond to statements that the Ontic interpreter can verify in a single step.

Proof. Let P be a poset in which $\bigvee S$ exists for all $S \subseteq P$. For $H \subseteq P$, let K be the set of all lower bounds of H. By hypothesis $\bigvee K$ exists; set $a = \bigvee K$. If $h \in H$, then $h \geq k$ for all $k \in K$; therefore $h \geq a$ and $a \in K$. Thus a is the greatest member of K, that is $a = \bigwedge H$.

```
(IN-CONTEXT ((LET-BE P POSET)
             (SUPPOSE (FORALL ((S (SUBSET-OF (U-SET P))))
                        (EXISTS (LEAST-UPPER-BOUND-OF S P))))
             (LET-BE H (SUBSET-OF (U-SET P)))
             (PUSH-GOAL
               (EXISTS (GREATEST-LOWER-BOUND-OF H P)))); #1

  (IN-CONTEXT
        ((LET-BE K (THE-SET-OF-ALL (LOWER-BOUND-OF H P)))
         (LET-BE a (THE (LEAST-UPPER-BOUND-OF K P))))

    (IN-CONTEXT ((PUSH-GOAL (IS a (LOWER-BOUND-OF H P)))); #2
      (IN-CONTEXT ((SUPPOSE (EXISTS (MEMBER-OF H)))
                   (LET-BE h0 (MEMBER-OF H)))

        (IN-CONTEXT
             ((PUSH-GOAL (IS h0 (UPPER-BOUND-OF K P)))); #3
          (IN-CONTEXT
                ((SUPPOSE (EXISTS (MEMBER-OF K)))
                 (LET-BE k0 (MEMBER-OF K)))
            (NOTE-GOAL)); #3
          (NOTE-GOAL))); #3

      (NOTE-GOAL)); #2

    (NOTE-GOAL))); #1
```

Figure 1.17: Least upper bounds yield greatest lower bounds.

The natural argument shown in figure 1.17 concerns complete lattices. A complete lattice is a partially ordered set P such that every subset of P has both a least upper bound and a greatest lower bound. The arguments in figure 1.17 show that if every subset of a partially ordered set P has a least upper bound, then every subset of P must also have a greatest lower bound. In the argument from Gratzer's book, shown in figure 1.17, the least upper bound of a set H is denoted $\bigvee H$ and the greatest lower bound of H is denoted $\bigwedge H$. In the Ontic proof the goals are numbered so that one can more easily see the association between the statement of the goal and the achievement of the goal.

A different measure of the length of an argument or proof is obtained by counting the number of type expressions rather than words. The number of type expressions used in an argument provides a rough measure of the number of "statements" involved. A direct translation of the natural argument in figure 1.17 into Ontic would contain 14 type expressions while the actual Ontic proof contains only 13 type expressions yielding an expansion factor of about one. Thus the basic result that the Ontic proof is about the same length as the English proof does not depend on the particular way in which one measures length.

In checking the proof in figure 1.17 the Ontic interpreter makes use of a large lemma library. The system uses some basic facts about partial orders together with the following facts:

1. The definitions of the concepts involved, e.g. the definition of partial orders, lower bound, least member and greatest lower bound.
2. The fact that if s is a subset of a partially ordered set p then the set of all lower bounds of s is a subset of p.
3. The fact that for any subset s of a partially ordered set p, there is at most one least upper bound of s.

One can argue that the expansion factor measured for the proof of figure 1.17 is too low because the Ontic interpreter was allowed to use preproven lemmas that are not shown in the formal proof. But all of the lemmas used

Lemma	**Predicate Count Expansion Factor**	**Word Count Expansion Factor**
If arbitrary least upper bounds exist then arbitrary greatest lower bounds also exist.	.9	1.0
Every filter is contained in an ultrafilter.	1.3	1.2
If F is an ultrafilter and $x \vee y \in F$ then $x \in F$ or $y \in F$.	2.1	2.7
Every Boolean algebra is isomorphic to a field of sets.	2.0	1.7

Table 1.1: Various Measurements of the Expansion Factor

by the Ontic interpreter in proving this theorem are of general interest and have in fact been used in several different contexts. Furthermore the last two lemmas listed above have simple one or two line proofs in the Ontic system and thus if those lemmas had not been in the lemma library the proof shown in figure 1.17 would not be much longer.

It seems likely that human mathematicians unconsciously invoke a large data base of general facts when they think about mathematical objects. Furthermore, it seems likely that in familiarizing oneself with a new domain one must verify a large body of "trivial" facts and incorporate these facts into the way one thinks about the domain.

Bell and Machover's text on mathematical logic gives a more concise proof of the lemma of figure 1.17 ([Bell & Machover 77] page 127). In the proof a least upper bound is called a supremum and a greatest lower bound is called an infimum.

> Let L be a partially ordered set in which each subset has a supremum. Let X be a subset of L, and let Y be the set of lower bounds of X in L. Then Y has a supremum z and it is not hard to see that z is the infimum of X.

A direct translation of the statements in Bell and Machover's into the language Ontic would contain 7 type expressions while the machine verifiable Ontic proof has 13 type expressions yielding a predicate count expansion factor of about two. While Bell and Machover's proof is clearly shorter than Gratzer's proof, Bell and Machover's proof includes the phrase "and it is not hard to see that". This phrase seems to be an admission that the given proof is not complete. Gratzer's proof, on the other hand, contains no such phrase and we must take Gratzer's proof as a fully expanded (complete) proof.

The appendix contains a complete listing of a mathematical development that ends with a proof of the Stone representation theorem for Boolean lattices. This appendix provides a large number of examples of Ontic proofs and these proofs can be used to evaluate the Ontic verifier. Table 1.1 shows four expansion factor measurements taken from four of the larger proofs done in the Ontic system. The table lists both a predicate count expansion factor and a word count expansion factor for each test case. Both the natural argument and the corresponding Ontic proofs for each test case can be found in the appropriate sections of the appendix.

The machine readable proofs underlying table 1.1 relied on an extensive lemma library and the expansion factor measurements are thus open to the criticism that parts of the machine readable proof have been hidden in the lemma library. However, once a sufficiently large lemma library has been constructed, it should be possible to prove new theorems without extending the basic lemma library. I believe that the numbers listed in table 1.1 are accurate in that, with a mature lemma library, new theorems can be verified with small expansion factors even if the expansion factor takes into account all lemmas added during the verification.

1.4 The Inference Mechanisms

All of the inference mechanisms used in the Ontic system manipulate labelings of a graph structure. More specifically, the Ontic system compiles the lemma library into a graph structure where the nodes in the graph structure correspond to unique expressions in the formal language. There are nodes that correspond to terms, formulas, type expressions, function expressions and type generator expressions. The graph structure has nine different kinds of "links" where each link expresses a certain way that nodes are related. For example if n is the node corresponding to the type expression `(LOWER-BOUND-OF` s p`)` then there is a subexpression link that relates n to the three nodes that correspond to the expressions `LOWER-BOUND-OF`, s and p. There are also links that express Boolean constraints among formula nodes, links that relate a lambda function to the node representing the bound variable and the body of that expression, and six other kinds of links.

A labeling of the graph structure consists of two parts: a partial truth labeling on formula nodes, and a color labeling on all nodes. For each formula node p the partial truth labeling either assigns p the label *true*, assigns p the label *false*, or leaves p unlabeled. The color nodes represent an equivalence relation on nodes: two nodes with the same color label are considered to be equivalent, i.e. proven equal in the current context. Whenever an inference is made the system updates the labeling: either a formula is assigned a truth label or two equivalence classes are merged by recoloring one class to be the same color as the other class. Any such inference process for updating labels on a fixed graph structure must terminate because there are only finitely many formula nodes which can be assigned truth labels and every merger of equivalence classes reduces the number of equivalence classes remaining and the number of equivalence classes can not drop below one.

The same underlying graph structure can be used in many different contexts. Graph structure is never thrown away: each time new graph structure is created it is saved for use in other contexts. Truth and color labels, on the other hand, are temporary; they are thrown away, for example, when the system stops considering a particular supposition or focus object.

This section presents an informal description of the inference mechanisms which operate on the graph structure and the way in which the graph structure is constructed from the lemma library. A precise description of the inference mechanisms and graph structure is presented in chapters 4 and 5. Chapter 6 contains a precise description of the Ontic language and chapter 7 contains a precise description of the way the lemma library is compiled into graph structure.

1.4.1 Inference Mechanisms for Quantifier-Free Logic

Boolean constraint propagation and congruence closure were originally designed as inference techniques for quantifier-free logic. Boolean constraint propagation adds truth labels in response to Boolean constraints and previous truth labels. For example, if the node for the implication `(=>` Φ Ψ`)` is labeled true, and the node for Φ is labeled true, then Boolean constraint propagation will ensure that the node for Ψ is labeled true. Similarly, if the node for `(=>` Φ Ψ`)` is labeled true, and the node Ψ is labeled false, then Boolean constraint propagation will ensure that the node for Φ is labeled false.

Boolean constraint propagation is also responsible for ensuring a certain relationship between color labels and the truth labels of nodes representing equalities. To ensure this relationship the system may merge equivalence classes in response to the addition of a truth label or, alternatively, add a truth label in response to the merger of equivalence classes. More specifically, let p be a node which represents an equation between the expressions represented by nodes n_1 and n_2. If the equality node p is assigned the label *true* then the system ensures that nodes n_1 and n_2 have the same color label, i.e. are in the same equivalence class. On the other hand if the nodes n_1 and n_2 are in the same equivalence class then the system ensures that p is assigned the label *true*.[7]

Congruence closure is responsible for ensuring that the equivalence relation represented by the color labels respects the substitution of equals for equals. For example consider terms `(POWER-SET` s_1`)` and `(POWER-SET` s_2`)`.

[7]If n_1 and n_2 are in the same equivalence class and the equality node p has been labeled false by some other inference process then the system signals a contradiction.

Congruence closure ensures that if the nodes representing the terms s_1 and s_2 have the same color label (are in the same equivalence class) then the nodes representing the expressions (POWER-SET s_1) and (POWER-SET s_2) also have the same color label. When two equivalence classes are merged congruence closure may merge additional equivalence classes in order to ensure that the equivalence relation respects the substitution of equals for equals.

1.4.2 Generic Individuals, Classification, and Focused Binding

Recall that a context consists of a lemma library, a set of focus objects and a set of suppositions about the focus objects. Focused binding is a way of applying the universally quantified formulas in the lemma library to the focus objects in a context. This is done using an inheritance mechanism similar in spirit to Fahlman's virtual copy mechanism based on marker propagation [Fahlman 79]. More specifically, each type τ which has been compiled into a node in the graph structure is associated with a set of (typically two or three) *generic individuals* of that type. Information that is known to hold for a given type is explicitly stated about the generic individuals of that type. A focus object which is known to be an instance of type τ becomes a "virtual copy" of one of the generic individuals of type τ and thus inherits information from that individual.

Each generic individual is a term node in the graph structure. Information which is known to hold for the type τ is explicitly stated about each generic individual of type τ. More specifically, if the system compiles into graph structure a universal formula of the form

$$\texttt{(FORALL ((}x\ \tau\texttt{))}\ \Phi(x)\texttt{)}$$

then for each generic individual g of type τ which is added to the graph structure, the system constructs a Boolean constraint equivalent to the following implication.

```
(=> (AND (FORALL ((x τ)) Φ(x))
         (EXISTS-SOME τ))
    Φ(g))
```

Given the above constraint, if the universally quantified formula is true in a context, and instances of type τ are known to exist in that context, then the body of the universal formula is known to be true for each generic individual of type τ. In this way everything that is known about the type in general is explicitly stated about the generic individuals of that type.

Classification assigns types to focus objects. Classification is needed in order for focus objects to inherit information from generic individuals. The system classifies a focus object r by collecting a set, *types*(r), of types known to hold for r according to the following rules:

1. If the node for the formula (IS r τ) is labeled true then τ is included in *types*(r).
2. If s is a term that is in the same equivalence class as the focus object r, and if the formula (IS s σ) is labeled true, then σ is included in *types*(r).
3. If τ is a member of *types*(r), and the formula (IS-EVERY τ σ) is labeled true, then σ is included in *types*(r).
4. If τ is a member of *types*(r) and σ is a type in the same equivalence class (with the same color as) τ then σ is included in *types*(r).

Focused binding causes a given focus object to inherit information from a given generic individual. More specifically, for each focus object r and each type τ in the set *types*(r) the system chooses a generic individual g of type τ and constructs the binding $g \mapsto r$. The generic individual g can be thought of as a typed variable and the binding $g \mapsto r$ can be thought of as a variable binding. In the Ontic system the variable binding $g \mapsto r$ is implemented via the color labels: when the system constructs the binding $g \mapsto r$ it assigns g and r the same color label, thereby making g equivalent to r. When g is made equivalent to r, the congruence closure mechanism is used to "unify" or "match" the expressions involving the generic individual g with the expressions involving the focus object r. In this way the focus object r becomes a virtual copy of the generic individual g. Since general knowledge about the type τ is explicitly stated about the generic individual g, general knowledge about the type τ becomes effectively stated about the focus object r. In this way general facts in the lemma library are effectively applied to focus objects of the correct type.

The focused binding process is sometimes called *semantic modulation* because it involves modulating (changing) the interpretation of a fixed generic individual. The same generic individual can be bound to different focus objects in different contexts. In this way the system modulates the semantic denotation of the generic individual, hence the term semantic modulation.

There are several subtleties involved in focused binding. First, the system must not bind the same generic individual to two different focus objects simultaneously. For example, consider a generic number g and two numbers j and k which are focus objects such that j is an even number and k is an odd number. If the system bound the generic number g to both j and k simultaneously then it could prove that g was both even and odd and thus that there exists a number which is both even and odd.

A second subtlety involves the possibility of circular bindings. Before generating a binding of the form $g \mapsto r$ the system must be sure that r does not depend on g. Any term can be given as a focus object. Generic individuals themselves correspond to terms in the Ontic language (they are Ontic variables) and thus a focus object may be a generic individual or a term that contains a generic individual.[8] For example, if g is a generic individual ranging over numbers then the term $1 + g$ might be a focus object. In this case one should prevent the binding $g \mapsto 1 + g$; no number is equal to the next number. The dependency test for avoiding circular bindings is similar to the occurs-check done in unification. Given a focus object r of type τ the system chooses a generic individual g such that g does not "occur in" r. Unfortunately the occurs-check performed by the Ontic system is somewhat

[8]By abuse of notation I will identify a generic individual with the corrosponding Ontic variable. Technically, a generic individual is a node in the graph structure while an Ontic variable is a term of the Ontic language.

complicated. Consider a generic individual y which ranges over numbers which are greater than x, where x is a generic individual ranging over all numbers (y is a generic individual of type `(GREATER-THAN` x`)`). The binding $x \mapsto 1 + y$ is illegal because it forces x to be greater than itself. However, x is not a free variable of the expression $1 + y$. Rather, x is a free variable of the *type* of y where y is a free variable of $1 + y$. We say that an expression u *depends on* a variable x if either x appears free in u or there is some free variable y of u such that the type of y depends on x. Unfortunately this notion of dependence still does not provide a sound occurs-check in the Ontic system: if x and y both range over arbitrary numbers the system must prevent the two simultaneous bindings $x \mapsto 1 + y$ and $y \mapsto 1 + x$. To prevent such circularities the system must take previous bindings into account when computing occurs-checks. It turns out that there is a subtle interaction between previous bindings and the dependencies introduced by types. More specifically, if the system has already constructed the binding $y \mapsto u$ then the type of y can be ignored in the occurs-check procedure. The resulting occurs-check procedure runs quickly but the proof that the occurs-check procedure leads to sound inference is somewhat complex (see sections 5.2 and 5.3).

1.4.3 Automatic Universal Generalization

The fourth inference mechanism used by the Ontic system is automatic universal generalization. Universal generalization can be applied when the system has deduced a fact about an arbitrary individual and no assumptions have been made about that individual. More specifically, a universal generalization inference can be made if:

- g is a generic individual of type τ.
- The system has labeled the node for a formula $\Phi(g)$ true.
- No assumptions have been made about the individual g other than the assumption that it is an instance of type τ.

- No free variable of $\Phi(g)$ has a type that depends on g. The notion of dependence used here is the same as that defined above: τ depends on x just in case x appears free in τ or some free variable of τ has a type which depends on x.

When the above conditions are met the system can infer the universal closure

$$\texttt{(FORALL ((}x\ \tau\texttt{))}\ \Phi(x)\texttt{)}$$

There are several things to note about automatic universal generalization. First, this inference mechanism does not construct new formulas or new graph structure; automatic universal generalization is only applied when the graph already contains nodes for the formulas $\Phi(g)$ and the universal closure

$$\texttt{(FORALL ((}x\ \tau\texttt{))}\ \Phi(x)\texttt{)}$$

Second, types play a central role in the automatic universal generalization mechanism. When the system proves the formula $\Phi(g)$ it is allowed to use the fact that g is an instance of the type τ, and the resulting universal statement applies to all instances of τ. Third, without the last restriction universal generalization is unsound. For example, consider a generic individual y that ranges over numbers greater than the generic number x. Without making any assumptions about x and y other than that they are both instances of their respective types, the system can deduce that x is less than y. It does not follow, however, that all numbers are less than y; there is no largest number. The fact that x is less than y does not imply that all numbers are less then y because the x "occurs in" y; x is a free variable in the type of y. The same proof that shows that the Ontic occurs-check procedure is sound for focused binding can be used to show that the Ontic occurs-check procedure leads to sound universal generalization.

The above notion of universal generalization can be made more powerful by relaxing the restriction that no assumptions have been made about the arbitrary individual being generalized over. More specifically one can perform universal generalization under the following conditions:

- g is a generic individual of type τ.

- The system has labeled the node for a formula $\Phi(g)$ true.
- The system has bound g via the binding $g \mapsto h$.
- h is a generic individual of type σ where σ has the same color label as τ in the current context.
- No assumptions have been made about h.
- h does not "occur in" any free variable of $\Phi(g)$ other than g.

When the above conditions are met the system can infer the universal closure

$$\texttt{(FORALL ((}x\ \tau\texttt{))}\ \Phi(x)\texttt{)}$$

Again, note that this inference mechanism does not construct new formulas or add new graph structure. In order for this inference mechanism to be applied, all of the formulas involved must already be compiled into nodes in the graph structure.

To see the importance of the more general automatic universal generalization mechanism, consider a subset s of a partially ordered set p and the set u of all lower bounds of s as a subset of p. Now consider a member x of s. By definition u is the set of lower bounds of s so x is an upper bound of u. It turns out that in the Ontic system proving this last statement requires universal generalization. More specifically the Ontic system must focus on an arbitrary member y of u and note that x is greater than or equal to y. Since y is an arbitrary member of u, x is greater than or equal to all members of u. In this situation the system will construct the following bindings:

$$s' \mapsto u$$

$$z \mapsto y$$

Here s' is a generic individual ranging over arbitrary subsets of p and z is a generic individual ranging over members of s'. Now y is a generic individual ranging over members of u and z is a generic individual ranging over members of s', so z and y are different generic individuals whose types happen to be equal in the current context. Furthermore z is bound to y. In this situation the system generalizes over the variable z rather than the variable y. The system must generalize over z rather than y because the definition of upper bound is stated about the generic subset s' rather than the particular subset u and thus the quantified formula in question quantifies over members of s' rather than members of u.

All of the inference mechanisms used in the Ontic system run concurrently and interact with each other. Inferences can lead to more knowledge about the types of focus objects; this can lead to more bindings, which can lead in turn to more inference. The time required to finish the overall inference process is bounded by the size of the graph structure. This is because the inference processes can only add as many truth labels as there are formula nodes and can only merge as many equivalence classes as there are nodes in total. The factors that contribute to the size of the graph structure are discussed below.

1.4.4 The Size of the Graph Structure

When a new focus object r of type τ is introduced, it is possible that all generic individuals of type τ have either already been bound to other objects or occur in the focus object r and thus can not be bound to r. In this case the system creates a new generic individual of type τ and copies all of the information known about type τ as explicit statements about that new generic individual. Once the generic individual has been constructed, however, it is saved and can be used in other contexts. For most arguments there are already enough generic individuals in the graph structure to accommodate the focus objects and no new graph structure is created. However, if there are not enough generic individuals to accommodate the focus objects, then generic individuals are created on demand as focus objects are introduced. As generic individuals are created the underlying graph structure expands.

The size of the graph structure created by the Ontic compiler is determined by the library of mathematical facts and by the number of generic individuals that have been created for each type. Fortunately, for any given bound on the level of quantifier nesting, the size of the graph structure is linear in the size of the lemma library; the amount of graph structure is the sum over all lemmas of the amount of structure created by each lemma. This fact allows the Ontic system to be used with large libraries of mathematical facts. However, the cost of an individual lemma can be quite high. Consider a lemma of the following form:

$$\texttt{(FORALL ((}x\ \tau_1\texttt{) (}y\ \tau_2\texttt{) (}z\ \tau_3\texttt{))}\ \Phi(x,y,z)\texttt{)}$$

The body of this lemma will be copied for each triple g_1, g_2, g_3 where g_1, g_2 and g_3 are generic individuals of type τ_1, τ_2 and τ_3 respectively. In general every quantified formula which is compiled into graph structure gets instantiated with every generic individual of the appropriate type. Let $|\tau_1|$, $|\tau_2|$ and $|\tau_3|$ be the number of generic individuals for τ_1, τ_2, and τ_3 respectively. The number of copies of the body of the above lemma is:

$$|\tau_1| \cdot |\tau_2| \cdot |\tau_3|$$

Generic individuals are created on demand as new focus objects are introduced. If no more than n focus objects have been introduced in any one context then there will be at most n generic individuals of each type. If the maximum number of quantifiers used in any lemma is d then there can be no more than n^d copies of the body of each lemma. Lemmas rarely involve more than three quantifiers and most sessions with the Ontic interpreter involve at most five simultaneous focus objects. Thus a typical lemma in a typical session generates no more than 5^3 or 125 instantiations. In practice this number is smaller because most lemmas quantify over highly specialized types and there are typically only a small number of generic individuals of specialized types. Again note that the size of the graph structure is *linear* in the size of the lemma library; the total amount of graph structure is just the sum over all lemmas of the amount of structure generated by each lemma. However, the size of graph structure is very sensitive to the maximum number of focus objects introduced in a given context. A good rule of thumb seems to be that the size of the graph structure is proportional to $n^3|\Sigma|$ where n is the maximum number of focus objects introduced in any one context and $|\Sigma|$ is the size of the lemma library.

Chapter 2

Comparison with Other Work

The Ontic system represents a synthesis of ideas from artificial intelligence and automated theorem proving. Constraint propagation is a forward chaining inference technique that terminates quickly because it monotonically fills a finite set of "slots"; the Ontic system monotonically generates truth and color labels for nodes in a finite graph structure. Congruence closure is a powerful theorem proving technique for reasoning about equality. Congruence closure is usually viewed as an inference procedure reasoning about equalities involving ground (variable-free) expressions. In the Ontic system, however, congruence closure is used as an integral part of general first order theorem proving. Focused binding, also known as semantic modulation, is closely related to inheritance mechanisms which have been developed for knowledge representation languages and object oriented computer programming languages. Focused binding integrates inheritance with other theorem proving mechanisms. Congruence closure is used to implement a strong virtual copy mechanism that allows focus objects to inherit from generic individuals. Automatic universal generalization is perhaps the simplest and yet the most original feature of the Ontic system. Ontic brings all these ideas together in a single integrated inference process.

The first section of this chapter relates each of the four basic inference mechanisms used in Ontic with previous work in knowledge representation and automated theorem proving. The second section of the chapter relates

Ontic's focused binding mechanism to unification. Focused binding and unification provide alternative ways of selecting and applying facts from a fact library. The third section of the chapter lists various theorem proving mechanisms other than those used in the Ontic system and attempts to show how they are related to Ontic. The final section of the chapter lists some of the general issues to be considered in constructing a proof verification system and discusses how Oxx set term Ontic and various other systems have addressed those issues.

2.1 Inference Mechanisms Similar to Ontic's

The following four sections discuss each of Ontic's four inference mechanisms in turn. The first three inference mechanisms are related to well known inference techniques. Ontic, however, brings these mechanisms together in an integrated, object oriented theorem proving process.

2.1.1 Constraint Propagation

There are many mechanisms in the artificial intelligence literature which could be described as constraint propagators. By "constraint propagation" I mean an inference process whose running time, or number of processing steps, is directly bounded by the size of a finite *constraint network*. Ontic is a constraint propagation system in two ways. First of all, one of the fundamental inference mechanisms is Boolean constraint propagation which is a special case of the arc-consistency constraint propagation technique for general constraint satisfaction problems [Mackworth 77]. Second, all of Ontic's inference mechanisms operate by labeling a graph structure. The graph structure is analogous to a constraint network in that the total number of labeling operations is directly bounded by the size of that graph structure.

Many artificial intelligence researchers have used constraint propagation. Waltz used constraint propagation to filter the possible interpretations of lines in a line drawings of polygonal physical objects [Waltz 75]. A line in a drawing of a scene can be interpreted as a convex edge on single object, a concave edge on a single object or an edge between two objects. A particular interpretation of an edge is called a "label" for that edge. Vertices between edges provide constraints on the possible interpretations of edges. In Waltz line labeling a forward chaining inference process systematically eliminates possible labelings of individual edges. The running time of the process is directly bounded by the number of edges and the number of labels that can be eliminated.

The Waltz line labeling procedure can be used in the more general setting of an arbitrary constraint satisfaction problem [Mackworth 77]. A constraint satisfaction problem consists of a set of variables each of which can be assigned one of a finite set of possible values and a set of constraints where each constraint restricts the simultaneous assignments for a given subset of the variables. The arc-consistency procedure, which is a straightforward generalization of Waltz labeling, systematically eliminates possible interpretations of variables based on local constraints. The running time of the arc-consistency procedure is directly bounded by the number of variables and the number of possible assignments for each variable. Boolean constraint propagation is a special case of the arc-consistency procedure where the variables are Boolean, i.e. they can be assigned the labels *true* or *false*, and the constraints are disjunctive clauses involving the Boolean variables. Boolean constraint propagation is described in more detail in chapter 4.

Sussman and Steele have proposed a language for expressing constraints on real valued variables and constraint propagation techniques for dealing with such constraints [Sussman & Steele 80]. The number of propagation operations performed by Sussman and Steele's system was directly bounded by the number of variables involved.

Nevins constructed a forward chaining geometry theorem prover which restricted the forward chaining inference process to an a priori fixed set of formulas [Nevins 74]. Nevins' program used a diagram to focus the system's attention on certain lines. If a geometry problem has n points then there are $\binom{n}{2}$ possible line segments between these points. A diagram, however, specifies a subset of the $\binom{n}{2}$ lines, those actually drawn in the diagram. By limiting forward chaining to statements about these focused lines, the forward chaining process does not generate large numbers of irrelevant facts.

With Nevins' focused forward chaining mechanism there is no need for the diagrammatic filter used by Gelernter [Gelernter 59].

Ontic's inference processes operate on a finite graph structure; the number of labeling operations is directly bounded by the size of that graph structure. The Ontic system can use the same graph structure in different contexts to reason about different focus objects. When a generic individual g is bound to a focus object r, a formula involving g can be viewed as a formula involving r; in the presence of bindings the formula nodes in the graph structure represent formulas about focus objects. Different bindings cause the nodes in the graph structure to represent statements about different objects.

2.1.2 Congruence Closure

Congruence closure is the process of "closing" an equivalence relation on expressions under the inference rule of substitution of equals for equals. Congruence closure was first discussed by Kozen for reasoning about finitely presented algebras [Kozen 77]. Congruence closure has also been used by Nelson and Oppen in constructing fast decision procedures for a variety of problems that arise in automatic program verification [Nelson and Oppen 80]. The congruence closure procedure used in the Ontic system, and discussed in some detail in chapter 4, is based on the procedure given by Downey, Sethi and Tarjan [Downey, Sethi & Tarjan 80].

Ontic uses congruence closure both as a mechanism for reasoning about equality and as a replacement for unification. The relationship between Ontic's use of congruence closure and traditional unification is discussed in section 2.2.

2.1.3 Focused Binding as Inheritance

Focused binding can be viewed as an inheritance mechanism: information about a type is inherited by instances of that type. Type hierarchies and inheritance also play an important role in object oriented programming languages such as Smalltalk [Ingalls 76]. In object-oriented programming, data types are organized into a hierarchy where one data type can be a subtype of another. Data objects are usually records with data fields. A given data object inherits both data fields and functional behavior from all the supertypes of its immediate type. A fairly rigorous, though not very general, treatment of some basic ideas in object-oriented programming is given in [Cardelli 84].

Type hierarchies and inheritance also play a central role in many knowledge representation systems and object oriented programming languages. Frame-based knowledge representation languages typically allow the user to define "concepts" which he or she organizes into an "is-a" hierarchy (e.g. [Brachman & Schmolze 85]). A concept represents a class of structured objects; the concept is associated with a set of "slots"; an instance of that concept is an object with specific "fillers" or "values" for the slots of the concept. For example the concept *room* might have slots *ceiling*, *floor*, *walls*, and *furniture*. Any particular room will have a particular ceiling, a particular floor, and a particular set of pieces of furniture. Furthermore, a concept can place certain constraints on the slot fillers. For example the concept *room* might specify that the furniture slot is always filled with a set of physical objects. The user could introduce the concept *auditorium* as a specialization of the concept *room* and the concept *auditorium* would then automatically "inherit" the slots and constraints of the concept *room*.

Ontic's focused binding mechanism is very similar to Fahlman's virtual copy mechanism based on marker propagation [Fahlman 76]. Fahlman proposed a semantic network formalism in which objects inherit information from classes by passing markers along links in the network. The marker passing is done in such a way that the object being considered becomes a "virtual copy" of generic objects which contain information about classes. In the Ontic system color labels are used instead of Fahlman's markers. A focus object is made into a virtual copy of a generic individual by assigning the generic individual the same color label as the focus object; congruence closure ensures that if two nodes have the same color label then they have identical properties.

In the Ontic system inheritance is just one aspect of an integrated theorem proving mechanism. Generic individuals are viewed as logical variables that range over a given type. Inheritance occurs when a generic individual g is bound to a focus object r via a binding $g \mapsto r$. Fahlman's inheritance mechanism, on the other hand, was not viewed as a formal inference mechanism and Fahlman did not propose integrating his inheritance mechanism with other formal inference techniques such as Boolean constraint propagation, congruence closure, or automatic universal generalization.

2.1.4 Automatic Universal Generalization

Automatic universal generalization arises from a very simple idea: if a fact is proven about a generic individual g of type τ and no assumptions have been made about g other than that g is an instance of τ, then the fact holds for all instances of τ. In spite of the simplicity of the underlying idea, Ontic's universal generalization technique seems to be unlike any previous automatic inference mechanism. For example, a comparison of Ontic and resolution theorem provers shows that when Ontic performs universal generalization it is treating a generic individual as a Skolem constant introduced by a universally quantified goal formula. But, unlike resolution, the Ontic system does not make any distinction between variables and Skolem constants. Generic individuals in Ontic are used in three different ways. If instances of a type τ are known to exist then each generic individual of type τ is asserted to be an instance of τ. In this way the generic individuals can be used as Skolem constants introduced by the premise that instances of τ exist. But generic individuals are also used as variables that can be bound to specific terms in much the same way that resolution variables are bound during unification. Generic individuals are used in yet a third way by the universal generalization mechanism; universal generalization treats generic individuals as Skolem constants introduced by universally quantified goal statements.

The real novelty of the Ontic system lies in the way that the above four inference mechanisms are brought together. Ontic integrates constraint propagation, congruence closure, inheritance, and universal generalization in a single object-oriented labeling process on a fixed graph structure.

2.2 Focused Binding vs. Unification

One of the most striking features of the Ontic system, as compared to other theorem proving systems, is that Ontic does not use unification. Unification is often used to access information in a data base. A Prolog interpreter, for example, takes a goal formula and finds a production in the data base whose left hand side unifies with the given goal. A rewrite system takes an expression to be simplified and finds a rewrite rule in the data base whose left hand side unifies with the expression to be simplified. Under the set-of-support heuristic a resolution theorem prover finds a clause in the data base such that a literal of that clause unifies with a subgoal in the current problem. In all these cases the system is finding an expression in the data base which unifies with an expression in the current problem.

Ontic accesses information in the lemma library via the focused binding mechanism. Both unification and focused binding generate variable bindings which are useful to produce specialized instances of the general formulas in a data base. However, unification and focused binding generate variable bindings in very different ways. Unification starts with the expressions to be matched and generates variable bindings which lead to the match. Focused binding, on the other hand, starts with focus objects then generates variable bindings (bindings of generic individuals) and relies on congruence closure to generate "matches" between expressions involving variables and expressions involving the focus objects. Unification is a local process: unification is used in the application of a single rewrite rule or in a single resolution step. Focused binding, on the other hand, is a global process involving an arbitrary number of facts from the lemma library. Focused binding is integrated into the theorem-proving process. Automated inference and knowledge from the lemma library is used both in determining the types which apply to a given object and in determining equivalences between expressions after bindings have been performed.

Considerable research has been directed toward incorporating various kinds of knowledge (axiomatic theories) into unification. Equational axioms, such as the commutativity and associativity properties of addition, can be incorporated into the unification process so that, for example, $a + x$ matches

$b+a$ with the binding $x \mapsto b$. Taxonomic information, information involving the classification of objects into types, can also be incorporated into the unification process. Because Ontic's focused binding mechanism is integrated with the theorem proving process, focused binding automatically incorporates both equational and taxonomic information into the matching process; any lemma in the lemma library may be used in Ontic's matching process. However, unlike most unification mechanisms, Ontic's matching process is not logically complete: it is possible that two expressions are provably equivalent and yet the Ontic system fails to match them. This is consistent with the overall design philosophy of the Ontic system; to ensure that the system always terminates quickly, completeness has been abandoned.

2.2.1 Unification Relative to Equational Theories

There has been a considerable amount of research dedicated to incorporating equational theories into unification. For example consider addition as an associative and commutative operator. Now consider the problem of unifying $x + (a + b)$ and $a + (c + b)$. The binding $x \mapsto c$ unifies these two terms in the sense that the equation

$$c + (a + b) = a + (c + b)$$

follows from the associative and commutative properties of $+$.

More generally, let Γ be a set of universally quantified equations between first order terms. For example Γ might consist of the associative and commutative laws for addition. A general purpose theorem prover, such as a resolution system, could handle the equations in Γ simply by adding the equations in Γ to the data base of general facts. In practice, however, it seems more efficient to incorporate certain equational facts into the unification process. Once these facts have been incorporated into the unification process they can be removed from the general data base without loss of logical completeness.

A given set of equational axioms Γ has a corresponding unification problem. For any substitution σ and any expression u we define $\sigma(u)$ to be the result of simultaneously replacing all free variables in u with their image under σ. A *unification* of two expressions s and t relative to the axioms in Γ is a substitution σ which yields a match between s and t relative to Γ, i.e. such that the equational formulas in Γ imply that $\sigma(s)$ equals $\sigma(t)$. If Γ states that $+$ is associative and commutative then the substitution $\{x \mapsto c\}$ unifies $x + (a + b)$ and $a + (c + b)$ relative to Γ. The unification problem for Γ is the problem of computing, for any given expressions s and t, a representation of all unifications of s and t relative to Γ.

If Γ consists of a single commutative operation then it is easy to determine if there exists a unification of any two given terms relative to Γ. On the other hand if Γ states that a binary operator $\cdot$ is associative, and $\cdot$ distributes over a binary operator $+$, then there is no procedure which can decide the existence of a unification of two arbitrary terms relative to Γ. These results and others are discussed in a review article by Siekmann [Siekmann 84].

Unification relative to equational theories can be compared with Ontic's focused binding mechanism. Ontic first binds variables (generic individuals) of the appropriate type to focus objects and then uses congruence closure to "match" expressions involving the variables with expressions involving the focus objects. Ontic's matching process (congruence closure) automatically incorporates equations from the lemma library. For example suppose that Ontic's lemma library contains the associative and commutative laws for addition on the natural numbers. More specifically, suppose the lemma library includes the following three lemmas:

```
(FORALL ((X NATURAL-NUMBER)
         (Y NATURAL-NUMBER))

  (= (SUM-OF X Y)
     (SUM-OF Y X)))
```

```
(FORALL ((X NATURAL-NUMBER)
         (Y NATURAL-NUMBER)
         (Z NATURAL-NUMBER))

  (= (SUM-OF X (SUM-OF Y Z))
     (SUM-OF (SUM-OF X Y) Z)))
```

```
(FORALL ((X NATURAL-NUMBER)
         (Y NATURAL-NUMBER)
         (Z NATURAL-NUMBER))
   (= (SUM-OF X (SUM-OF Y Z))
      (SUM-OF (SUM-OF Y Z) X)))
```

The first and second lemma above express the fact that addition is commutative and associative respectively. The third lemma follows from the other two. If the third lemma were not explicitly given, however, then when focusing on three generic numbers g_1, g_2 and g_3 the following equation would not be obvious to the Ontic system.

$$g_1 + (g_2 + g_3) = (g_2 + g_3) + g_1$$

To prove this equation in the absence of the third lemma, or to prove the third lemma from the other two, the system must focus on the sum $g_2 + g_3$ so that the commutative law is applied to $g_1 + (g_2 + g_3)$. The associative and commutative laws allow for twelve different ways of writing down the sum of g_1, g_2 and g_3: there are six different orders in which the numbers can appear and two different ways of parenthesizing each order. In the presence of the three lemmas given above all twelve ways of writing the sum are equivalent; the twelve nodes in the graph structure that represent the twelve different expressions for this sum are all in the same equivalence class; they have the same color label. Now suppose the user focuses on three particular numbers a, b and c. The Ontic system will bind a generic number to each of these three particular numbers; assume that the system generates the bindings

$$g_1 \mapsto a$$

$$g_2 \mapsto b$$

$$g_3 \mapsto c$$

Given that all twelve expressions for the sum of g_1, g_2 and g_3 are in the same equivalence class, congruence closure together with the above bindings ensures that the term a+(b+c) is equivalent to the term b+(c+a). By using congruence closure as a matching mechanism, and by precompiling equational theories as equations involving generic individuals, the Ontic system automatically performs theory-relative matching. Unfortunately Ontic's matching process is not complete; the incompleteness is demonstrated by the need for the third lemma given above. On the other hand, as the example shows, one can always improve the power of the matching process by adding derived equational lemmas to the lemma library.

Ontic's focused binding mechanism automatically incorporates any equational lemma whatsoever into the congruence closure process; in the Ontic system one does not have to design a new theory-relative matching process for each new theory as one must do for theory relative unification. Ontic's mechanism has the disadvantage however that there is no guarantee of completeness — congruence closure may fail to equate semantically equal terms.

2.2.2 Unification Relative to Taxonomic Theories

Several researchers have investigated unification relative to theories which are not equational. Non-equational theories incorporated into the unification process are sometimes called *taxonomic theories* because they usually encode a classification of objects into types. The separation of "taxonomic" and "assertional" information has been discussed in the knowledge representation literature [Brachman, Fikes & Levesque 82]. For example consider the axiom

$$\forall x\ \mathbf{whale}(x) \Rightarrow \mathbf{mammal}(x)$$

This axiom expresses an inclusion relation between the "type" **whale** and the type **mammal**. Inclusion relations of this kind can be incorporated into the unification process and need not be stated explicitly in the data base of a general purpose theorem prover.

Walther has given a unification algorithm which handles any taxonomic theory expressible as a partial order on class symbols [Walther 84a]. He showed that for any such taxonomic theory Γ and any two typed terms s and t the set of all unifications of s and t can be expressed with a finite set of most general unifiers (i.e. the unification problem is finitary). Furthermore he showed that if the type hierarchy is a tree then there is a single most general unifier.

Ait-Kaci and Nasr have given a unification algorithm for a more expressive class of taxonomic theories and propose using this algorithm in an implementation of the programming language PROLOG [Ait-Kaci & Nasr 86]. Stickel has investigated the use of taxonomic theories in even greater generality although Stickel does not address unification as a mechanism for generating variable bindings (only the ground case is considered as lifting to the general case is "straightforward") [Stickel 85].

Ontic's mechanism for inheritance via semantic modulation is based on taxonomic information. More specifically, the Ontic system classifies each focus object by associating each focus object with a set of types known to be true of that focus object. This classification process takes the type hierarchy into account. For example if r is a focus object, σ is a type known to hold of r, and the formula `(IS-EVERY` σ τ`)` is labeled true, then the classification process will collect τ as a type known to hold of r.

Unlike unification, Ontic's focused binding mechanism integrates the use of type information with other theorem proving mechanisms. Ontic may prove a statement about types and use that statement immediately in classifying the current focus objects. Ontic's focused binding mechanism automatically incorporates arbitrary lemmas about the types of objects. There is no guarantee, however, that Ontic's focused binding mechanism will derive all the logical consequences of taxonomic information.

2.2.3 Higher-Order Unification

Unification has been generalized to allow for higher-order variables; higher-order unification can be used to bind variables that range over functions and predicates as well as variables ranging over first order terms. For example, consider the induction schema for Peano arithmetic.

$$P(0) \;\wedge\; \forall n\,(P(n) \Rightarrow P(n+1)) \;\Rightarrow\; \forall n\, P(n) \tag{2.1}$$

In this schema P is a variable which ranges over predicates. This schema can be instantiated with any predicate P and higher-order unification can be used to find bindings for P. For example consider a function f which is known to be monotone:

$$\forall m \quad f(m+1) \geq f(m) \tag{2.2}$$

and we wish to prove

$$\forall m \quad f(m) \geq f(0) \tag{2.3}$$

To prove this last statement a backward chaining theorem prover might unify $P(n)$ from the conclusion of 2.1 with the goal $f(m) \geq f(0)$ from 2.3. This unification leads to the following bindings:

$$n \mapsto m$$

$$P \mapsto (\lambda(n)\, f(n) \geq f(0))$$

A backward chaining inference system could then establish the antecedents of 2.1 under the above binding for the predicate P.

The first complete unification procedure for higher-order logic was constructed by Gerard Huet [Huet 75]. Higher-order unification has been used effectively in at least two mathematical verification systems, Ketonen's EKL system [Ketonen 84] and Andrews' TPS [Miller et al. 82]. In both systems the higher-order unification procedure was found to terminate quickly in practice.

The Ontic system is higher-order in the same sense that axiomatic set theory is higher-order; functions and predicates can be "reified" as sets and thus first order variables can be made to range over functions and predicates. In the Ontic system the user can focus on a reified predicate Q and thus cause the system to bind variables to the predicate Q. This kind of "higher-order" binding is used many times in the mathematical development given in the appendix.

While the Ontic system does allow for higher-order reasoning, the Ontic system does not adequately handle mathematical induction. Verifying induction proofs in the Ontic system results in a large expansion factor; the machine readable proofs are significantly longer than the natural language counterpart.

Higher-order unification provides one technique for reducing the expansion factor for induction proofs. The EKL system relies on higher order unification both in establishing the well formedness of recursive definitions and in performing induction arguments to prove properties of recursively defined functions. But there seem to be other, perhaps even better, techniques for reasoning about recursive definitions. The Boyer-Moore theorem prover is extremely effective in performing induction arguments but does not use higher order unification [Boyer & Moore 79]. Ontic's weakness with regard to induction arguments and possible ways of making Ontic's induction mechanisms more powerful are discussed in section 3.2.2.

2.3 Inference Mechanisms Unlike Ontic's

This section surveys some of the general purpose inference mechanisms that have been introduced in the past thirty years and compares these mechanisms with Ontic's object-oriented inference mechanisms. Only general purpose inference mechanisms are discussed here; domain specific mechanisms, such as Chou's application of Wu's method for geometry theorem, will not be discussed [Wu 86] [Chou 84]. I will also not discuss decision procedures for particular theories or mechanisms for combining decision procedures [Nelson & Oppen 79] [Shostak 82].

This section briefly discusses some particular general purpose inference systems. The automath proof verification systems used normalization of the typed lambda calculus as an inference mechanism. The Davis-Putnam procedure was based on a direct enumeration of the Herbrand universe for a set of first order sentences. The resolution procedure and its variants improved on the Davis-Putnam procedure by introducing unification, thereby allowing a large number of ground inferences to be abbreviated with a single resolution step. The Boyer-Moore theorem prover finds induction proofs for verifying equations concerning recursive programs in pure Lisp. The Boyer-Moore theorem prover is based on user-defined (and machine verified) rewrite rules together with heuristics for generalizing induction hypotheses. The Knuth-Bendix procedure provides a way of converting a set of unordered equations into a set of rewrite rules for canonicalizing expressions. The Knuth-Bendix procedure can also be used for proving certain equations about recursive programs via an "inductionless" induction technique. Finally, a fair number of systems have been constructed which use automated theorem proving support to verify natural deduction proofs.

2.3.1 Automath

The typed lambda calculus is closely related to intuitionistic (constructive) proof theory. The analogy between typed lambda calculus and intuitionistic proof theory is based on viewing types as formulas and viewing a term of type τ as a proof of τ (where τ is viewed as a formula). If the formulas encoded by types include quantifiers, i.e., if the type system has dependent types, then it can be difficult to determine if a term u has type τ. More specifically, determining if u has type τ may involve normalizing (i.e. evaluating) the term u. This normalization process can be viewed as inference where β reductions correspond to either the inference rule of modus-ponens or the inference rule of universal instantiation.

The relationship between types and formulas of intuitionistic logic underlies one of the earlier mathematical verification systems, the Automath system [deBruijn 68], [deBruijn 73]. The Automath system has been used to verify Landau's Grundlagen, a book on the foundations of the integers, rationals, reals, and complex numbers [Jutting 79]. The book includes a very rigorous (almost formal) definition of each number system. The rationals are defined as equivalences classes of pairs of integers, the reals are defined as Dedekind cuts in the rationals, the complex numbers are defined as pairs of reals. The book also includes proofs that the basic algebraic operations on these numbers are well defined (e.g. addition of rationals, multiplication of reals). No significant theorems are proven other than the well-formedness of these basic definitions.

Even though Landau's grundlagen is an extremely rigorous (almost formal) book, the version of the book readable by the Automath system is about ten times as long as the Grundlagen itself. This indicates that the Automath verifier does not use powerful automatic inference mechanisms; there is not yet good evidence that normalization of the typed lambda calculus is a useful automated inference mechanism.

2.3.2 The Davis-Putnam Procedure

The Davis-Putnam procedure [Davis & Putnam 60] is based directly on Herbrand's theorem for the first order predicate calculus. Herbrand's theorem implies that if Σ is an unsatisfiable set of first order formulas in Skolem normal form then there exists a *finite* set Γ of *ground instantiations* of Σ such that Γ is inconsistent. It is possible to write a computer program that decides whether a set of ground formulas is consistent. To determine if the original set Σ of first order formulas is satisfiable, one can simply enumerate all finite ground instantiations Γ of Σ and test each one for consistency. If Σ is inconsistent then by Herbrand's theorem one will find a ground instantiation Γ of Σ that is inconsistent.

The Davis-Putnam procedure is not used today; resolution theorem proving is more effective [Robinson 65]. The Davis-Putnam procedure spends most of its time deciding the satisfiability of quantifier-free ground formulas. Resolution theorem proving is more effective because a large (infinite) number of of ground inferences are summarized in a single resolution step. More specifically, the formula generated by a resolution step can be viewed as a universally quantified lemma which summarizes a large number of ground statements [Robinson 65]. Because other proof mechanisms (resolution) are more effective than the Davis-Putnam procedure, the Davis-Putnam procedure will not be discussed further here.

2.3.3 Resolution and its Variants

Most research in automated theorem proving in the past twenty years has been based in some way on resolution. The basic resolution rule was introduced by Robinson in 1965 and shown to be refutation complete for first order predicate calculus [Robinson 65]. The resolution principle represented a clear advance over the Davis-Putnam procedure because a single resolution step abbreviates a large number of the ground inferences. However the number of possible n-step deductions grows exponentially in n and it soon became clear that resolution theorem provers could not, in practice, find significant theorems by searching this large space of possible deductions.

The late sixties saw the development of a large number of restrictions on the resolution principle. Each such restriction rules out certain resolution steps and thus reduces the number of possible n-step deductions. In spite of the reduction in the number of possible inferences, various restricted forms of resolution are logically complete. A description of various restrictions and modifications of the resolution rule can be found in [Loveland 78]. Connection graph resolution, a resolution restriction invented by Kowalski, is described in [Bibel 81].

One perceived difficulty with resolution theorem proving, in addition to the large search spaces encountered, is the use of normal forms. Resolution requires that first order formulas be put in normal from in three stages. First, all quantifiers are moved to the beginning of the formula resulting in a formula in *prenex normal form.* Second, existential quantifiers are replaced by skolem functions resulting in an equisatisfiable formula in prenex normal form with only universal quantifiers. Finally, the matrix of the formula (the part after the quantifiers) must be placed in conjunctive normal form resulting in a set of universally quantified clauses where each clause is a disjunction of literals. Several researchers have developed theorem proving techniques which are similar to resolution but which do not require the last normalization step: the matrix of the formula need not be in conjunctive normal form. Such "non-clausal" provers are described in [Andrews 81], [Murray 82], and [Stickel 82]. These non-clausal procedures are similar to resolution in that they use unification to find matches between formulas and matched formulas are combined to generate new formulas. The non-clausal procedures are also similar to resolution in that existential quantification is eliminated in favor of Skolem constants.

Research in resolution theorem proving and related techniques has focused on establishing logical completeness. However, logical completeness may not be important in practice. The Boyer-Moore theorem prover is clearly not complete, it often terminates in failure, and yet the Boyer-Moore prover has been been used effectively in more applications than has any other theorem proving system.

As a side effect of focusing on completeness, the resolution theorem proving community has failed to make any distinction between "obvious" and "non-obvious" inferences. The failure to distinguish obvious and non-obvious inferences makes it difficult to use resolution theorem provers in interactive proof verifiers. Any interactive proof verifier based on resolution must have some way of forcing the resolution process to terminate so that a proposed proof step can be rejected in a finite amount of time. For example Bledsoe built an interactive verifier which simply imposed a time limit on the resolution process [Bledsoe 71]. A more principled restriction of the resolution process has been introduced by Davis [Davis 81] and used in the Mizar system [Trybulec & Blair 85]. However the restriction proposed by Davis forces the decision procedure for obvious inferences to determine the satisfiability of an arbitrary set of ground clauses. Determining the satisfiability of a set of ground clauses is known to be NP-complete. Furthermore, as far as I know, there has never been a detailed comparison of natural arguments and theorems provable under Davis' suggestion.

2.3.4 Rewriting Mechanisms

Automated inference systems often have a hard time dealing with equality and equational axioms. Directed rewrite systems provide one approach to reasoning about equality. The process of rewriting expressions is also known as *simplification*, *symbolic evaluation* or *demodulation*. Rewrite systems iteratively simplify a given expression until it is in canonical form. A statement can be proved by rewriting it to the constant **true**.

Some of the most effective theorem proving systems are based on rewrite mechanisms. Most notably, the Boyer-Moore theorem prover uses a simplification mechanism guided by user defined (but machine verified) rewrite rules [Boyer & Moore 79]. The Boyer-Moore theorem prover has been used to verify a wide variety of theorems from number theory, recursive function theory, formal logic and software and hardware verification [Boyer & Moore 84], [Shankar 85], [Russinoff 85], [Boyer & Moore 86]. The real power of the Boyer-Moore prover comes from its ability to perform induction proofs. However the simplification (rewrite) mechanism is central to the system.

The Boyer-Moore prover is primarily used to prove equations between terms defined in pure Lisp. Once an equation has been proven it is treated as a rewrite rule to be used in future proofs. The direction of each newly proven rewrite rule is provided by the human user, e.g. when the system proves an equation $s = t$ the human user specifies whether this equation should be treated as $s \rightarrow t$, which rewrites s to t, or as $t \rightarrow s$, which rewrites t to s.

Ketonen's EKL system is another example of a verification system based on user defined rewrite rules [Ketonen 84]. As in the Boyer-Moore prover, the direction of EKL rewrite rules are specified by the human user. Unlike the Boyer-Moore prover however, the EKL system uses Huet's higher order unification procedure to perform induction proofs. The EKL system lacks the facility for generalizing induction hypotheses used in the Boyer-Moore prover.

Knuth and Bendix developed a powerful method for constructing decision procedures for certain equational theories [Knuth & Bendix 69]. Unlike the Boyer-Moore prover and the EKL system, the Knuth-Bendix procedure can be used to *automatically* convert undirected equations to directed rewrite rules. More specifically, equations can be ordered via a general (but user specified) order $\succ$ on terms. If $s \succ t$ then the equation $s = t$ becomes the rule $s \rightarrow t$; if $t \succ s$ then the equation $s = t$ becomes $t \rightarrow s$. The partial order $\succ$ used in the Knuth-Bendix procedure must be well founded, respect term structure, and obey substitutions (see [Knuth & Bendix 69] for details).

After ordering equations into rewrite rules, the Knuth-Bendix procedure can also be used to automatically construct additional "derived" rewrite rules. More specifically, given a set of unordered equations, and an acceptable partial order $\succ$ on terms, the Knuth-Bendix procedure both converts equations to rewrite rules and constructs additional rewrite rules whose validity follows from the original equations. The set of rewrite rules that results from applying the Knuth-Bendix procedure to a set of Σ is often much larger than

Σ. If the Knuth-Bendix procedure terminates with success it generates a set of rewrite rules that completely canonicalize expressions relative to the given equations; by canonicalizing expressions one can determine if two terms can be proven equal from the original set of equations. Unfortunately, however, the Knuth-Bendix procedure does not always succeed; it can either terminate in failure or fail to terminate.

The Knuth-Bendix procedure has been used extensively in system which manipulate equational specifications of computer programs and equational programming languages [Kapur et al. 86] [Lescanne 86] [Huet 86]. These systems are based on an equational view of programming in which computer data structures are viewed as terms constructed from atomic symbols (Lisp atoms) and "data constructor functions" such as the Lisp function `CONS`. Recursive functions can be defined via equations involving the defined function symbols [Guttag & Horning 78] [O'Donnell 85].

The Knuth-Bendix procedure can also be used to generate "induction arguments" of the type performed by the Boyer-Moore theorem prover [Huet & Hullot 83]. More specifically, consider the closed (variable free) terms which can be constructed from a set of "atoms" (constructor functions of no arguments), constructor functions (functions such as `CONS` which construct data objects), and defined functions. A "data object" is a term with no defined functions. Let Σ be a set of equations which *defines* the defined function symbols as operations on the data objects, i.e. no two data objects can be proven equal from Σ and every closed term involving defined functions can be proven (under Σ) to be equal to some data object. Now suppose we wish to prove some equation $s = t$ where s and t are distinct terms involving defined functions and free variables. For example the equation $s = t$ might state the associativity of the `APPEND` function on lists. The equation $s = t$ holds in the data object universe just in case there is no counter example, i.e. no ground variable substitution σ such that $\sigma(s)$ denotes a different data object from $\sigma(t)$. If there exists a counter example to the equation $s = t$ then adding this equation to Σ would allow one to prove an equation between two distinct data objects. The Knuth-Bendix procedure can be used (in some cases) to convert $\Sigma \cup \{s = t\}$ to a complete set of rewrite rules. By examining this set of rewrite rules it is possible to determine whether $\Sigma \cup \{s = t\}$ allows one to prove an equation between distinct data objects. If such equation is provable then the equation $s = t$ has a counter example. If no such equation between distinct data objects is provable from $\Sigma \cup \{s = t\}$ then the equation $s = t$ has no counter examples and must be true in the data object universe. In general it may be possible to show that $s = t$ has counter examples at an intermediate point in the Knuth-Bendix procedure; thus a complete set of rewrite rules for $\Sigma \cup \{s = t\}$ may not be required.

One problem with the Knuth-Bendix procedure however is the need for a single partial order on all expressions. There may be domain specific intuitions about how terms should be rewritten and it is difficult to incorporate such knowledge into a single uniform term ordering. While some sophisticated partial orders have been developed [Dershowitz 79], it is not yet clear whether a uniform term ordering can be used for the large verifications that have been done with the Boyer-Moore prover.

Like unification research, research on term rewriting systems using the Knuth-Bendix mechanism has centered on the notion of logical completeness. There are many equational theories Σ with an undecidable set of logical consequences (an undecidable word problem) and in this case the Knuth-Bendix procedure either terminates in failure or fails to terminate. In systems based on the Knuth-Bendix procedure it is not clear what to do when the procedure fails. Even if a complete set of reductions is found, the time required to perform the rewriting may be prohibitively large. The rigid framework of the Knuth-Bendix procedure may make it difficult to perform the large verifications that have been done with the Boyer-Moore prover; it is not clear that a Knuth-Bendix based system could verify the RSA encryption algorithm or the undecidability of the halting problem as has been done with the Boyer-Moore system [Boyer & Moore 84] [Boyer & Moore 86].

Rewrite systems are designed to handle equational theories. The Ontic system handles equality with its congruence closure mechanism; rewrite rules are not used. The congruence closure mechanism can be quite powerful in practice. Figure 2.1 gives an example of an inference done using Ontic's congruence closure mechanism. Consider a distributive lattice with a least member 0 and a greatest member 1 (a lattice with a least and greatest member is called *bounded*). If x and y are members of the lattice L then we say that x and y are complements if the meet of x and y is 0 and the join of

```
(IN-CONTEXT ((LET-BE L (AND-TYPE DISTRIBUTIVE-LATTICE
                                 BOUNDED-LATTICE))
             (LET-BE X (IN-U-SET L))
             (PUSH-GOAL
               (AT-MOST-ONE (COMPLEMENT-OF X L))))
   (IN-CONTEXT ((SUPPOSE (EXISTS (COMPLEMENT-OF X L)))
                (LET-BE Y1 (COMPLEMENT-OF X L))
                (LET-BE Y2 (COMPLEMENT-OF X L)))
     (NOTE-GOAL))
   (NOTE-GOAL))
```

Ontic "sees" this theorem using its congruence closure mechanism as follows:

$$\begin{aligned}
y_1 &= y_1 \wedge 1 && \text{A previously established fact.}\\
&= y_1 \wedge (y_2 \vee x) && \text{Because } y_2 \text{ is a complement of } x.\\
&= (y_1 \wedge y_2) \vee (y_1 \wedge x) && \text{By definition of a distributive lattice.}\\
&= (y_1 \wedge y_2) \vee 0 && \text{Because } y_1 \text{ is a complement of } x.\\
&= (y_1 \wedge y_2) \vee (y_2 \wedge x) && \text{Because } y_2 \text{ is a complement of } x.\\
&= (y_2 \wedge y_1) \vee (y_2 \wedge x) && \text{Because } \wedge \text{ is commutative.}\\
&= y_2 \wedge (y_1 \vee x) && \text{By definition of a distributive lattice.}\\
&= y_2 \wedge 1 && \text{Because } y_1 \text{ is a complement of } x.\\
&= y_2 && \text{Because } y_2 = y_2 \wedge 1
\end{aligned}$$

Figure 2.1: A statement that is obvious to Ontic but not obvious to people

x and y is 1. It was obvious to the Ontic interpreter that in any bounded distributive lattice a given member x has at most one complement. Ontic's proof of this fact, also shown in figure 2.1, uses congruence closure.

Figure 2.1 shows that congruence closure is a powerful technique for reasoning about equality. Because Ontic handles equality with congruence closure rather than rewrite rules, there is no need for the user to specify rewrite directions for equations; the Ontic system can handle undirected declarative equations. The value of declarative as opposed to procedural representations is discussed in more detail in section 2.4.2.

2.3.5 Natural Deduction Systems

Natural deduction systems are based on "natural" rules of inference. A given rule says that a goal G of a certain form can be proven by reducing the goal G to the subgoals $G_1, G_2 \ldots G_n$. Different rules provide different ways of achieving a goal where the success of any one rule is sufficient. The earliest natural deduction system was Newell, Shaw and Simon's Logic Theorist [Newell, Shaw & Simon 57]. This system used natural deduction rules and backward chaining to prove theorems in Whitehead and Russell's Principia Mathematica. Soon after the construction of the Logic Theorist, Gelernter constructed his program for finding proofs in Euclidean geometry [Gelernter 59]. Gelernter's system also used backward chaining and natural deduction rules but the subgoals were pruned by the use of a diagram, i.e. a model of the assumptions in the proof. If a subgoal was false in the diagram then the system could infer that the subgoal could not be achieved and thus should be abandoned.

During the sixties research in automatic theorem proving focused primarily on resolution theorem proving. However, during the early seventies frustration with resolution systems lead to a renewed interest in natural deduction systems [Bledsoe 77]. Natural deduction systems from the seventies include [Bledsoe 71], [Nevins 72], [Bledsoe et al. 72], [Reiter 73], [Ernst 73], [Goldstien 73], [Bledsoe & Bruell 73], and [deKleer et al. 77]. These later natural deduction systems often used resolution as a subroutine for proving subgoals. A time limit was imposed on resolution proofs to force the resolution theorem prover to terminate quickly [Bledsoe 71].

One of the major problems with using resolution as a test for "obvious" subgoals was the tendency of resolution to get lost when it was given too many initial facts. In other words resolution was not able to automatically find the relevant facts in a large lemma library. As Bledsoe says in [Bledsoe 71]:

> One of the more serious [problems is referencing]. The computer should be able to bring to bear "all it knows" (all definition axioms and previously proven theorems) ... But if one attempts a resolution proof on a large number of formulas, the result is the production of a glut of irrelevant clauses and sure failure, even when the best known search strategies are used. Thus the crucial part of a resolution proof is the *selection* of the reference theorems by the *human* user; the human, by this one action, usually employs more skill than that used by the computer in the proof.

It is useful to remember that this was written in 1971, well after most of the refinements to resolution had been developed. These comments about the ineffectiveness of resolution on large lemma libraries are probably as true today as they were in 1971. The Ontic interpreter on the other hand seems to handle large lemma libraries without difficulty. It would be interesting to reconstruct these old natural deduction systems using the Ontic interpreter rather than resolution to test for obvious subgoals.

The Seventies also saw a development of basic natural deduction proof checking systems that did not provide much automated reasoning support. For example McDonald and Suppes developed an interactive proof checking system for teaching an introductory logic course [McDonald & Suppes 84]. Richard Weyhrauch also developed the FOL system for checking first order logic proofs [Weyhrauch 77].

While the FOL system does not provide sophisticated general purpose theorem proving, it does provide a uniform mechanism for associating any given predicate or function symbol with a computer program for computing the value of the predicate or function on "semantic" arguments. It seems clear that mathematical verification systems could benefit from the addition of computational oracles. Along with procedures for basic arithmetic (addition multiplication etc.) one can imagine incorporating procedures for symbolic integration, series summation, or polynomial manipulation. No attempt has been made to incorporate such features into the Ontic system.

Procedural attachment is part of a general focus on "metatheory" within the FOL system [Weyhrauch 80]. While procedural attachment has clear potential value, I think the emphasis on metatheory is misplaced. There seems to be a fundamental unity in all mathematics; there is no fundamental distinction between "metamathematics", number theory, graph theory, finite combinatorics, or real analysis. A system which reason about numbers, graphs, and ordered sets can just as easily reason about formulas, models, and Tarskian truth functions.

During the late seventies and into the eighties there has been an emphasis on "programmable" natural deduction systems. These systems provide a mechanism for adding user defined inference rules. The first programmable natural deduction system was Edinburgh LCF [Gordon, Milner & Wadsworth 79]. A more recent programmable natural deduction system is the Nuprl system developed by Bates and Constable [Constable et al. 86] [Howe 86]. The Nuprl system grew out of research in interactive verifications systems [Constable et al. 82] and their use in teaching formal logic and formal approaches to program verification. The Nuprl system is based on constructive type theory and places particular emphasis on finding constructive proofs. The system provides a facility for converting a constructive proof that a certain number exists into a program for computing that number.

Backward chaining natural deduction systems use rules of inference to convert a given goal to a set of subgoals. In the Nuprl system the user can define new inference rules, or "tactics", for converting a goal to a set of subgoals. When a tactic replaces a goal G by a set of subgoals G_1, G_2, ... G_n the tactic must construct a proof showing that the replacement is sound, i.e. that the subgoals G_1, G_2, ... G_n imply the goal G. One could write a tactic for showing that any given set S is a subset of U by supposing that S is non-empty and then considering an arbitrary member of S. One could then use this tactic as a subroutine and write another tactic for showing that two sets are equal by showing that each is a subset of the other. In the

Ontic system one has to repeat this style of argument every time one wants to prove set equality. It seems likely that tactics could be used in the Ontic system to reduce the length of machine readable proofs. On the other hand it seems likely that Ontic's object oriented inference mechanisms could be used to reduce the length of proofs in the Nuprl system.

2.4 Issues in Automated Reasoning

There are several general issues involved in the construction of proof verification systems. First, in designing a verification system one should consider the expressive power of the formal language involved. Does the language allow one to express a wide variety of formal concepts and arguments? Second, one should consider the extent to which the knowledge base contains procedural as opposed to declarative information. Procedural information may help make the system run more effectively but procedural information is harder to construct and a reliance on procedural information makes automatic discovery of useful information more difficult. Third, one should consider whether the system should rely on backward or forward chaining. It is not clear whether forward chaining has any intrinsic advantage over backward chaining or vice versa. In both cases the basic problem is to control the generation of facts or subgoals. Simplification seems to be effective as a guiding principle in backward chaining while focus seems to be effective as a guiding principle in forward chaining.

2.4.1 Expressive Power

Some very restricted formal languages have tractable inference problems: there exists a tractable procedure for determining the validity of any statement expressible in the language. Thus there seems to be a trade off between expressive power and computational tractability in knowledge representation languages [Levesque & Brachman 85]. However this "trade off" is misleading. In order to design a language with a tractable inference problem one must design a language in which hard questions can not be asked. But this does not produce the result one really wants; rather than making it easier to answer hard questions, limiting the expressive power of a language simply makes it impossible to *ask* hard questions. On the other hand, increasing the expressive power of the reasoning language can make it easier to reason about hard questions.

Natural mathematics (mathematics done in natural language) seems to have a notion of "well typed" expressions. For example consider the well typed phrase

"the value of the map f on the point x"

as opposed to the "garbled" phrase

"the value of topological space X on the point x"

The notion of a well typed natural phrase seems to correspond to the notion of a well typed formal expression. Mathematicians talk about groups, rings, fields, topological spaces, differentiable manifolds, groups homomorphisms, differentiable maps and much more. It seems that in natural mathematics any definable set (or class) can be used as a type in determining the set of well typed phrases. Most strongly typed formal systems, however, do not allow arbitrary predicates to be used as types.

In designing a type system there appears to be a trade off between expressive power and computational tractability. One can ensure computational tractability by restricting the type system so that only certain simple predicates can be used as types. Restricted type systems can not express natural types such as "prime number", "symmetric matrix", or "transitive reduced graph". While the inability to express such types makes type-checking tractable, it prevents the type-checking process from even attempting to verify certain semantic properties of programs. It seems likely that one could construct a quickly terminating type-checking procedure which could verify all simple types and could also verify *some* more difficult "semantic" types. Restrictions on the vocabulary of types does not make it easier to answer hard questions, it only makes hard questions impossible to ask.

2.4.2 Declarative Representations

Many automated inference systems require every declarative fact to be augmented with procedural information: information about how the declarative fact is to be used in the inference process. Purely declarative facts, facts not augmented with procedural instructions, have the advantage that they are easier to generate — it seems easier for people to write down a set of purely declarative facts than to write down both the declarative facts and additional information about how those facts are to be used. The ease of generating purely declarative facts may be particularly important in discovery systems — systems which automatically generate new lemmas. The task of discovering and using new facts is easier if one does not have to specify procedural information each time a new fact is discovered.

Unfortunately, purely declarative facts have the disadvantage that they are more difficult to compute with. Ketonen has discussed the difficulty of constructing effective theorem provers that use purely declarative information [Ketonen 84]. In supporting the use of procedural information Ketonen considers the following formula:

$$P(x) \Rightarrow A = B$$

He argues that there is no single way to use this formula and lists the following possible procedural interpretations:

1. Replace $P(x) \Rightarrow A = B$ by **true** whenever it appears.
2. Replace $A = B$ by **true** if one can prove $P(x)$ in the current situation.
3. Replace $P(x)$ by *false* if one can prove $A \neq B$.
4. Replace A by B whenever one can prove $P(x)$.
5. Replace B by A whenever one can prove $P(x)$.
6. Replace A by B whenever one can prove $P(x)$ but not in terms resulting from this substitution.

Ketonen argues that one must choose between the above procedural interpretations. Interpretations (4) and (5) seem opposite in intent. Furthermore formulas involving quantifiers would have an even greater number of different interpretations. Ketonen concludes that the user must specify how formulas are to be used.

It seems that Ketonen's difficulty with purely declarative representation comes from his commitment to rewrite systems. Ontic's inference mechanism effectively uses interpretations (1) through (5) simultaneously. Replacing a formula Φ by **true** in a rewrite system is analogous to putting the label **true** on the node for Φ in the Ontic's marker propagation mechanism. In the Ontic system Boolean constraint propagation handles the procedural interpretations (1) through (3) above. In the Ontic system equalities between nodes are represented by giving those nodes the same color label. This representation of equality together with the congruence closure mechanism effectively handles both procedural interpretations (4) and (5). The 6th procedural interpretation seems a little strange and is not handled in the Ontic system — congruence closure effectively performs all substitutions.

One of the primary features of the Knuth-Bendix procedure is that equations are automatically converted to rewrite rules using a single partial order that is defined for all terms. Thus, once the partial order has been defined, purely declarative equations are automatically given procedural interpretations. However the Knuth-Bendix procedure is not guaranteed to succeed: it may terminate without producing a complete set of rewrite rules or it may run forever in attempting to generate such a set. Furthermore, because the Knuth-Bendix procedure produces rewrite rules, it must choose either procedural interpretation (4) or interpretation (5) — the Ontic system effectively does both simultaneously. The effectiveness of the Knuth-Bendix procedure in large verification applications has not yet been established.

Further experimentation is needed to see if systems which use purely declarative information, such as Ontic, can be made as effective as systems which are based on rewrite rules, such as the Boyer-Moore theorem prover.

2.4.3 Forward Chaining

Forward chaining systems start with a set of premises and derive conclusions from those premises. Backward chaining systems start with a goal and reduce that goal to subgoals. It is not clear whether forward chaining has any intrinsic advantage over backward chaining or vice versa. In both cases the basic problem is to control the generation of facts or subgoals. Both forward chaining and backward chaining systems can become swamped in a sea of derived facts or derived subgoals. Certain sources of guidance seem to work for backward chaining and other sources of guidance seem to work for forward chaining.

Simplicity seems to work as a guiding principle in backward chaining. Rewrite systems are backward chaining because they start with the expression to be proved and rewrite that expression in an attempt to show it equivalent to the constant **true**. Rewrite systems are guided by some notion of simplicity: a goal expression is always replaced by a simpler goal. The notion of simplicity is either implicit in the user specified rewrite rules, as in the Boyer-Moore prover, or explicitly defined as an ordering on expressions, as in Knuth-Bendix based systems. In both cases however a notion of simplicity guides the generation of subgoals.

Focus seems to work as a guiding principle in forward chaining. Ontic's object oriented inference mechanisms are guided by the restriction that derived facts must be about the focus objects. A similar restriction is used in other forward chaining systems such as Nevins' geometry theorem prover [Nevins 74], constraint systems such as Waltz labeling [Waltz 75], and constraint languages such as that described by Sussman and Steele [Sussman & Steele 80].

It should be possible to integrate both backward and forward chaining in a single system. In such a system simplification should be used as a guiding principle in backward chaining and focus should be used as a guiding principle in forward chaining.

Chapter 3

Ontic as a Cognitive Model

One can attempt to evaluate Ontic as a model of human mathematical cognition by comparing the formal "proofs" that are acceptable to the Ontic system with the natural language proofs that are acceptable to people. There are some clear differences between Ontic proofs and natural arguments. In certain cases the Ontic system can verify proof steps that are not obvious to people; we say that Ontic exhibits superhuman performance. In other cases there are statements which are obvious to people but which require multi-step proofs in the Ontic system; we say that Ontic exhibits subhuman performance. The superhuman performance and much of the subhuman performance can be attributed to specific computational aspects of the Ontic system.

Ontic's congruence closure mechanism provides a clear example of superhuman performance. The Ontic system can use its congruence closure mechanism to "see" that in a distributive lattice complements are unique. This fact is not obvious to people. The appendix contains several examples of superhuman performance based on congruence closure. All of the examples involve lattice theoretic identities. One example is the proof of de Morgan's laws from the the algebraic axioms for a Boolean lattice.

After giving examples of superhuman inference based on congruence closure, a very fast computationally limited architecture is proposed for massively parallel computation. Boolean constraint propagation can be easily implemented in this massively parallel architecture but congruence closure can not. Substitution constraints are then proposed as an alternative to congruence closure. Substitution constraints perform many of the substitution inferences normally done by congruence closure. Furthermore, substitution constraints can be handled by Boolean constraint propagation and thus can be implemented on the proposed massively parallel architecture. However, substitution constraints do not generate the given examples of superhuman performance.

Of course the Ontic system also exhibits subhuman performance. Some cases of subhuman Ontic performance can be traced to weaknesses in the lemma library. Several proofs could be shortened by adding lemmas which introduce the principle of duality for Boolean lattices and the algebraic "definition" of a lattice. A more significant set of examples of subhuman Ontic performance involve mathematical induction. Although the Ontic system can be used to verify induction arguments, the expansion factor is large. In natural mathematics induction arguments are often unstated and unnoticed even though people understand the arguments and agree to their validity. For example consider a graph where the nodes of the graph are colored such that any two nodes with an arc between them have the same color. Clearly if nodes n and m have different colors then there is no path between them in the graph. To verify this clear and obvious fact with the Ontic system would require an induction on the length of paths. There are many other examples from both mathematics and common sense where induction arguments seem to be carried out at a subconscious level.

Future experimentation will certainly turn up additional ways in which the Ontic system exhibits subhuman performance; hopefully examples of subhuman performance will lead to the discovery of additional inference mechanisms that bring the system closer to human ability in verifying natural arguments.

3.1 Superhuman Performance

Congruence closure accounts for all the examples of superhuman performance of the Ontic system. The mathematical development given in the appendix contains six examples of superhuman performance based on congruence closure. All of these examples involve reasoning about lattice identities.

3.1.1 Examples of Superhuman Performance

The first example of superhuman Ontic performance is the proof that in a distributive lattice complements are unique. This example is given in chapter 2 and is discussed in more detail below. The second example is the proof of de Morgan's laws for complemented distributive lattices. De Morgan's laws are straightforward if one assumes that Boolean operations have their standard meaning as operators on sets, or equivalently, if Boolean operations have their standard meaning as operations on truth functions. However, until one has proven the Stone representation theorem one must consider the possibility that there exist pathological complemented distributive lattices in which the Boolean operations can not be viewed as operations on sets or as truth functions. The Ontic proof of de Morgan's laws and an analysis of that proof are shown in figure 3.1. Given several previously established simple identities for Boolean lattices the Ontic system immediately "sees" that de Morgan's laws are true in an arbitrary complemented distributive lattice.

The mathematical development in the appendix also contains a proof that for any elements x and y of a complemented distributive lattice the following are equivalent:

1. $x \leq y$
2. $y^* \leq x^*$
3. $x \wedge y^* = 0$
4. $x^* \vee y = 1$

An Ontic Proof:

```
(IN-CONTEXT ((LET-BE B BOOLEAN-LATTICE)
             (LET-BE X (IN-U-SET B))
             (LET-BE Y (IN-U-SET B))
             (LET-BE CX (COMPLEMENT X B))
             (LET-BE CY (COMPLEMENT Y B))
             (LET-BE M (MEET X Y B))
             (LET-BE J (JOIN CX CY B)))
     (NOTE (IS J (COMPLEMENT-OF M B))))
```

A Corresponding Natural Argument:

Let x^* and y^* be the complements of x and y respectively. Let m be the meet of x and y and let j be the join of x^* and y^*. We must show that m and j are compliments, i.e. that $m \wedge j = 0$ and $m \vee j = 1$. This can be done as follows:

$$\begin{aligned} m \wedge (x^* \vee y^*) &= (m \wedge x^*) \vee (m \wedge y^*) && \text{By distributivity of } \wedge \text{ over } \vee. \\ &= ((x \wedge x^*) \wedge y) \vee ((y \wedge y^*) \wedge x) && \text{By assoc. and comm. of } \wedge. \\ &= (0 \wedge y) \vee (0 \wedge x) && \text{By definition of complement.} \\ &= 0 && \text{By algebraic properties of 0.} \end{aligned}$$

$$\begin{aligned} (x \wedge y) \vee j &= (x \vee j) \wedge (y \vee j) && \text{By distributivity of } \vee \text{ over } \wedge. \\ &= (y^* \vee (x^* \vee x)) \wedge (x^* \vee (y^* \vee y)) && \text{By assoc. and comm. of } \vee. \\ &= (y^* \vee 1) \vee (x^* \vee 1) && \text{By definition of complement.} \\ &= 1 && \text{By algebraic properties of 1.} \end{aligned}$$

Figure 3.1: An example of superhuman Ontic performance.

The Ontic proof of the equivalence of the above facts is done by showing that 1) $\Rightarrow$ 2) $\Rightarrow$ 3) $\Rightarrow$ 4) $\Rightarrow$ 1). This is done in a context where the uniqueness of complements and de Morgan's laws have already been established. For each implication there is a set of four focus objects which makes the implication obvious to the Ontic system. The proof of each implication shows superhuman performance involving congruence closure.

3.1.2 A Very Fast Parallel Architecture

This section proposes an architecture for massively parallel computation and argues that, unlike Boolean constraint propagation, congruence closure is difficult to implement on this architecture. [1] People make truth judgments about obvious statements in about a second. Although the computation performed by neurons is not well understood, it is clear that neurons run very slowly. It seems likely that neurons would require one to ten milliseconds to compute the logical and of two Boolean signals. If people are computing truth judgments with Boolean circuitry, and if the gate delay for neuronal hardware is on the order of one to ten milliseconds, then people make truth judgments about obvious statements in 100 to 1000 gate delays. Computing complex truth judgments in only 100 to 1000 gate delays requires massive parallelism.

Consider a finite state machine where the state of the machine at time i is given by an n-bit bit vector D_i. The state transition table of the machine can be given by a Boolean circuit Φ of n inputs and n outputs where the state transitions of the machine are governed by the equation

$$D_{i+1} = \Phi(D_i)$$

To make the finite state machine run quickly the Boolean circuit Φ should have low depth, say ten gates. If Φ has depth ten then a state transition can

[1]It is easy to show that Boolean constraint propagation is polynomial time complete and thus "unparallelizable"; the worst case running time on a parallel machine is linear in the size of the graph. In many cases however, a parallel implementation would run much faster than a serial implementation; a parallel implementation runs in time proportional to the longest single inference chain while a serial implementation runs in time proportional to the total number of inferences.

be computed in ten gate delays. However, the bit vector defining the state of the machine can be very large: millions or tens of millions of bits, and the circuit Φ can involve millions or tens of millions of gates.

It seems possible to compile an Ontic graph structure into a Boolean circuit governing a finite state machine. More specificly, a labeling of an Ontic graph could be encoded in the state bit vector of the machine. The basic inference operations on graph labels could be incorporated into a Boolean circuit Φ governing state transitions. Two bits are needed for each formula node to represent the three possible labeling states of the node: true, false and unknown. Boolean constraints on formula nodes could be compiled directly in the structure of the Boolean circuit Φ. Every node in an Ontic graph is also associated with a color label. The color label for a given node in the graph could be represented with a set of bits in the machine's state vector. The Boolean circuit governing state transitions could be designed in such a way that if an equation node became true then the color labels of the equated nodes at time $i + 1$ would each be set to the maximum of the two labels at time i. In this way the color labels could be made to respect the truth of equality formulas. With the exception of congruence closure, all of the inference techniques used in the Ontic system seem to be amenable to a massively parallel implementation in a low-depth Boolean circuit governing a finite state machine.

The implementation of congruence closure described in chapter 5 uses a hash table to map color tuples to colors. In order to implement a hash table one needs to be able to compute memory addresses for a random access memory. I don't see any way of implementing parallel access to a large hash table in a low depth Boolean circuit governing a large finite state machine.

Congruence closure can be replaced with substitution constraints as described in the next section. Substitution constraints are Boolean constraints involving equality formulas; such constraints can be compiled directly into a low-depth Boolean circuit governing a finite state machine.

3.1.3 Substitution Constraints

Substitution constraints provide an alternative to congruence closure for reasoning about equality. Substitution constraints rely on Boolean constraint propagation's ability to handle certain equality inferences. Boolean constraint propagation ensures a simple relationship between the truth of equality formulas and the color labels encoding equivalence. Boolean constraint propagation, however, does not automatically handle the substitution of equals for equals; in the Ontic system substitution is handled by congruence closure. On the other hand, Boolean constraint propagation can be made to handle substitution by adding certain Boolean constraints called substitution constraints. Boolean constraint propagation with substitution constraints is weaker than congruence closure in that it generates fewer obvious truths in a given context.

As a simple example of a substitution constraint consider a term $f(c)$ which consists of an operator f applied to a specific argument c. We can assume that the operator f is defined on objects of a certain type τ and that c is an instance of τ. Suppose that g is a generic individual of type τ. To ensure that inheritance works properly one can add the Boolean constraint

$$g = c \quad \Rightarrow \quad f(g) = f(c)$$

Now if the system ever generates a binding $g \mapsto c$ then g and c will get the same color label and Boolean constraint propagation will ensure that the equation $g = c$ gets labeled true and thus, by the above substitution constraint, the equation $f(g) = f(c)$ will be labeled true. Independent of congruence closure, if $f(g)$ has the same color label as $f(c)$ then certain facts about $f(g)$ can be inherited by $f(c)$. For example if $f(g)$ is known to be an instance of a type σ then $f(c)$ will also be known to be an instance of the type σ. Thus the above Boolean constraint allows the binding $g \mapsto c$ to cause c to inherit facts that are stated in terms of g.

Substitution constraints can be used to perform inferences based on the substitution of equals for equals. Suppose that c is known to be equal to b and consider the terms $f(c)$ and $f(b)$. Furthermore assume the graph structure underlying Boolean constraint propagation includes the following substitution constraints

$$g = c \;\Rightarrow\; f(g) = f(c)$$

$$g = b \;\Rightarrow\; f(g) = f(b)$$

Now suppose that the system focuses on c and generates the binding $g \mapsto c$. Since c and b are known to be equal, the nodes for g, c, and b will all get the same color label. Thus the equations $g = c$ and $g = b$ will become true. Thus both the equations $f(g) = f(c)$ and $f(g) = f(b)$ will become true and the nodes for $f(g)$, $f(c)$ and $f(b)$ will all get the same color label. Thus focusing on c causes the system to deduce that $f(c)$ equals $f(b)$. This scheme for handling substitution of equals for equals via substitution constraints can be suitably generalized to handle operators of more than one argument.

Unlike congruence closure, substitution constraints combined with focused binding and Boolean constraint propagation will only substitute equals for equals when the expressions being substituted for are focus objects. All of the examples of superhuman Ontic performance involve substitutions of non-focused expressions.

3.1.4 Superhuman Performance Re-Examined

It is important to note that the scheme for equality inference based substitution constraints is not as powerful as the full congruence closure mechanism. More specifically, using substitution constraints the substitution of equals for equals can only be done when the substituted expressions are equal to some focus object. All of the examples of superhuman performance discussed above involve substitution for non-focused objects. For example consider the proof shown in chapter 2 that in a distributive lattice complements are unique. The uniqueness of complements is obvious to the Ontic system.

Figure 2.1 in chapter 2 shows the Ontic "proof" that complements are unique together with an expanded derivation showing how the Ontic system proved that if y_1 and y_2 are both complements of x then y_1 must equal y_2. The second line in the expanded derivation is derived by replacing 1 with $(y_2 \vee x)$ even though neither 1 nor $(y_2 \vee x)$ is a focus object. If congruence inference required focusing on the substituted expression then the second line could only be derived by focusing on $y_2 \vee x$. Similarly, line four is derived by substituting 0 for $y_1 \wedge x$ even though $y_1 \wedge x$ is not a focus object. Lines five and seven also involve substitution for non-focused expressions.

Even the weaker scheme based on substitution constraints could prove that complements are unique in a single inference step if the system focused on x, y_1, y_2, $y_2 \vee x$, $y_1 \wedge x$, $y_2 \wedge x$ and $y_2 \vee x$ all at the same time. However, it seems that people have a hard time focusing on seven objects simultaneously. The ability of the Ontic system to focus on a large number of objects simultaneously is perhaps another source of superhuman performance.

3.2 Subhuman Performance

Some proofs in the appendix exhibit subhuman performance which can be attributed, at least in part, to weaknesses in the lemma library. Other examples, not given in the appendix, indicate weaknesses in the fundamental inference architecture. It is hoped that examples of subhuman performance lead to new inference techniques which increase the usefulness of verification systems.

3.2.1 Weaknesses in the Lemma Library

The lemma library developed in the appendix does not include a duality principle for Lattices. Given an appropriate duality principle the proof of any identity in lattice theory would lead immediately to a proof of the dual identity. For example consider de Morgan's laws. A first de Morgan law can be phrased as follows.

$$(x \vee y)^* = x^* \wedge y^*$$

A second de Morgan's law can be derived from the first via a duality principle for Boolean lattices: the result of switching $\vee$ and $\wedge$ (and 1 and 0) in any Boolean lattice identity leads to another Boolean lattice identity. Given the duality principle for Boolean lattices the validity of the above de Morgan law leads immediately to the validity of the dual law:

$$(x \wedge y)^* = x^* \vee y^*$$

One could incorporate the duality principle into the Ontic system by defining the dual of a lattice. Given any lattice (or any partial order) the dual of the lattice is defined to be that lattice which has the same elements but in which the partial order has been reversed. Using the Ontic system one could easily define a function which mapped any lattice to its dual lattice. Furthermore one could prove that if L' is the dual of a Boolean lattice L then L' is a Boolean lattice such that the meet operation in L' equals the join operation in L, the join operation in L' equals the meet operation of L, and L' has the same complement operation as L. Given a Boolean lattice identity I one could then prove that the dual identity I' must hold in an arbitrary Boolean lattice L by considering the dual lattice L' and noting that I' holds in L just in case the lattice identity I holds in the dual L'.

Another example where standard notions could be added to the lemma library to reduce the length of proofs involves the algebraic characterization of a lattice. It turns out that the partial order of a lattice is determined by the meet and join operations and in fact one can define a Boolean lattice to be a set together with meet, join and complement operations that satisfy certain equational axioms. This algebraic view of a lattice is described in textbooks on lattice theory and could be added to Ontic's lemma library. The algebraic view of a lattice would allow a shorter machine readable proof of one of the lemmas given in the appendix. More specifically, the algebraic view of a lattice provides a short proof that if S is a subset of a Boolean lattice L such that S is closed under the meet, join and complement operations of L then the set S together with the partial order of L restricted to S forms a lattice with the same lattice operations as L.

3.2.2 Mathematical Induction

The clearest examples of subhuman behavior on the part of the Ontic system involve mathematical induction. Many common sense inferences appear to involve induction. Consider the following examples:

- Consider a colored graph in which adjacent nodes have the same color, i.e. if there is an arc between nodes n and m then n and m have the same color. If nodes n and m have different colors then there is no path between them in the graph. A formal proof requires induction on the length of paths in the graph.

- Consider a chess board. The white pawns start on the second rank and never move backward. Therefore no white pawn can ever appear on the first rank. A formal proof of this statement requires induction on the number of steps in the game.

- Consider two containers for holding marbles. Initially each container is empty. Marbles are then placed in the containers in pairs; one marble from each pair is placed in each container. No matter how many times this is done, assuming the containers do not overflow, there will be the same number of marbles in each container. A formal proof of this statement requires an induction on the number of marbles placed in the containers.

- Consider Rubic's cube. Suppose the cube starts in a solved position and is scrambled by some number of rotations of faces of the cube. There exists a set of steps that unscrambles the cube. A formal proof of this statement requires an induction on the number of rotations used to scramble the cube.

- Consider a mouse running in a maze. Suppose the maze is arranged inside a box such that there are no openings in the walls of the box and the mouse can not jump over the walls. No matter how long the mouse runs, and no matter where it goes inside the maize, the mouse will not get outside the box. A formal proof of this statement requires induction on the number of "moves" the mouse makes in the box.

In each of the above examples the conclusion is obvious to people. In each example, if the concepts involved were approximated by mathematically precise notions, then any mathematician would accept the conclusion as obvious and would not ask for further proof.

Ontic can be used to perform induction proofs. However induction proofs must be done explicitly: one must explicitly formulate the induction hypothesis and explicitly verify the induction step. For example, consider verifying that white pawns in a game of chess can not get to the first rank. This fact can be verified using the following induction principle for natural numbers.

```
(DEFTYPE SET-OF-NATNUMS
  (LAMBDA ((S SET))
    (IS-EVERY (MEMBER-OF S) NATURAL-NUMBER)))

(LEMMA
  (FORALL ((S SET-OF-NATNUMS))
    (=> (AND (IS ZERO (MEMBER-OF S))
             (FORALL ((N (MEMBER-OF S)))
               (IS (SUCCESSOR N) (MEMBER-OF S))))
        (IS-EVERY NATURAL-NUMBER (MEMBER-OF S)))))
```

The above induction principle says that if a set S contains zero and is closed under successor then it contains all numbers. The set S represents an induction hypothesis; S is the set of numbers which satisfy the hypothesis.

In the chess example one must prove that white pawns never end up on the first rank. More formally, let an instance of the type CHESS-GAME be a particular games of chess, i.e. a particular sequence of moves. If G is a particular chess game and N is some natural number then

```
(WHITE-PAWN-ON-BOARD G N)
```

denotes the type whose instances are the white pawns which are on the chess board after then N'th move of the game G. We let

```
(RANK-OF P G N)
```

be the rank occupied by the pawn P immediate after the N'th move of the game G. Figure 3.2 contains statements which follow form the rules of chess. An Ontic proof that pawns never get to the first rank is given in figure 3.3. The goals in the proof are numbered and the NOTE-GOAL steps are labeled

```
(FORALL ((G CHESS-GAME)
         (N NATURAL-NUMBER))
  (IS-EVERY (WHITE-PAWN-ON-BOARD G (SUCCESSOR N))
            (WHITE-PAWN-ON-BOARD G N)))

(FORALL ((G CHESS-GAME)
         (N NATURAL-NUMBER)
         (P (WHITE-PAWN-ON-BOARD G (SUCCESSOR N))))
   (IS (RANK-OF P G (SUCCESSOR N))
       (GREATER-OR-EQUAL-TO (RANK-OF P G N))))

(FORALL ((P (WHITE-PAWN-ON-BOARD G ZERO)))
  (IS (RANK-OF P G ZERO)
      (EQUAL-TO TWO)))
```

Figure 3.2: Statements which follow from the rules of chess.

with the number of the goal being noted. The proof uses the facts listed in table 3.2 together with simple facts about the ordering of natural numbers.

The proof starts by considering an arbitrary chess game G. The proof shows that the following induction hypothesis holds for any number N.

```
(FORALL ((P (WHITE-PAWN-ON-BOARD G N)))
    (IS (RANK-OF P G N)
        (GREATER-OR-EQUAL-TO TWO)))
```

The induction principle for natural numbers states that if a set of numbers contains zero and is closed under successor then it contains all numbers. If the induction hypothesis is Φ(N) then one should consider the set of all N such that Φ(N). For the above induction hypothesis one should consider the following set:

```
(THE-SET-OF-ALL
  (LAMBDA ((N NATURAL-NUMBER))
    (FORALL ((P (WHITE-PAWN-ON-BOARD G N)))
      (IS (RANK-OF P G N)
          (GREATER-OR-EQUAL-TO TWO)))))
```

```
(IN-CONTEXT ((LET-BE G CHESS-GAME)
             (LET-BE HYP-SATISFIERS
               (THE-SET-OF-ALL
                 (LAMBDA ((N NATNUM))
                   (FORALL ((P (WHITE-PAWN-ON-BOARD G N)))
                     (IS (RANK-OF P G N)
                         (GREATER-OR-EQUAL-TO TWO))))))
             (PUSH-GOAL
               (IS-EVERY NATURAL-NUMBER
                         (MEMBER-OF HYP-SATISFIERS)))) ;#1
  (IN-CONTEXT ((PUSH-GOAL
                 (IS ZERO (MEMBER-OF HYP-SATISFIERS)))) ;#2
    (IN-CONTEXT ((LET-BE ZEROVAR ZERO))
      (IN-CONTEXT ((SUPPOSE
                     (EXISTS-SOME (WHITE-PAWN-ON-BOARD G ZERO)))
                   (LET-BE P (WHITE-PAWN-ON-BOARD G ZERO))
                   (LET-BE TWOVAR TWO))
        (NOTE-GOAL)) ;#2
      (NOTE-GOAL))) ;#2
  (IN-CONTEXT ((PUSH-GOAL
                 (FORALL ((N (MEMBER-OF HYP-SATISFIERS)))
                   (IS (SUCCESSOR N) (MEMBER-OF HYP-SATISFIERS)))) ;#3
               (LET-BE SATISFIER (MEMBER-OF HYP-SATISFIERS))
               (LET-BE NEXT-SATISFIER (SUCC SATISFIER)))
    (IN-CONTEXT ((PUSH-GOAL
                   (FORALL ((P (WHITE-PAWN-ON-BOARD G NEXT-SATIFIER)))
                     (IS (RANK-OF P G NEXT-SATISFIER)
                         (GREATER-OR-EQUAL-TO TWO))))) ;#4
     (IN-CONTEXT ((SUPPOSE
                    (EXISTS-SOME
                      (WHITE-PAWN-ON-BOARD G NEXT-SATISFIER)))
                   (LET-BE P (WHITE-PAWN-ON-BOARD G NEXT-SATISFIER))
                   (LET-BE R1 (RANK-OF P G SATISFIER))
                   (LET-BE R2 (RANK-OF P G NEXT-SATISFIER))
                   (LET-BE TWOVAR TWO))
        (NOTE-GOAL)) ;#4
      (NOTE-GOAL)) ;#4
     (NOTE-GOAL)) ;#3
  (IN-CONTEXT ((LET-BE N (MEMBER-OF HYP-SATISFIERS)))
    (NOTE (IS HYP-SATISFIERS SET-OF-NATNUM)))
  (NOTE-GOAL)) ;#1
```

Figure 3.3: The proof that white pawns never get to the first rank.

The Ontic proof in figure 3.3 focuses on the set representing the induction hypothesis. It then proceeds to prove the base case and induction step. The base case uses the fact that the rank of a white pawn at time zero equals two and every number is greater than or equal to itself. In order to apply the fact that every number is greater than equal to itself one must focus on the number two. The induction step uses the fact that the rank of the pawn at time n is greater or equal to two and the rank of the pawn at time $n + 1$ is greater or equal to the rank at time n. To invoke the transitivity of the ordering on natural numbers one must focus on the three numbers given by the rank of pawn at times n and $n + 1$ together with the number two.

The proof shown in figure 3.3 is clearly much longer than a natural language argument which simply states that white pawns never get to the first rank. This example indicates that without additional theorem proving mechanisms the Ontic system will exhibit a large expansion factor on many induction proofs.

One possible mechanism for reducing the expansion factor in induction proofs would be a backward chaining procedure (a tactic) for automatically generating proofs such as the one shown in the figure 3.3. It would be easy to automatically convert the induction hypothesis into a set of numbers and automatically focus on that set of numbers. Furthermore one could automatically attempt to prove the base and induction cases of the argument. As figure 3.3 shows however, proving the base and induction cases with the Ontic system may require focusing on additional objects. In figure 3.3 the user focuses on an arbitrary white pawn and the number two. In the induction case the user focuses on the rank of the pawn at two different times. It seems that it might be difficult to automatically generate these additional focus objects.

Several automated inference systems include inference mechanisms for handling mathematical induction [Boyer & Moore 79] [Huet & Hullot 83] [Ketonen 84]. Research is needed to determine if these, or other, induction mechanisms can be incorporated into the Ontic system. These inference mechanisms are all backward chaining; the induction hypothesis is taken from the goal statement. It would be interesting to see if some forward chaining induction mechanism could be found that was more in the spirit of

Chapter 4

Quantifier Free Inference

Each context in the Ontic system is specified by a lemma library, a set of focus objects, and a set of assumptions. Given a lemma library, an assumption set, and a focus set the Ontic system uses focused forward chaining inference mechanisms to generate a set of "obvious truths" for the given context. In any given context the operations `NOTE` and `NOTE-GOAL` can be used to make permanent additions to the lemma library.

Each lemma, focus object and assumption is an expression in the formal language Ontic. Rather than manipulate Ontic expressions directly, the Ontic system compiles these expressions into graph structure where there is a one to one correspondence between graph nodes and Ontic expressions. Compilation and inference are separate processes; compilation generates a graph structure and inference manipulates graph labelings without creating additional graph structure. For efficiency reasons the graph constructed by the Ontic system is saved and used repeatedly in many different contexts.

In the Ontic system the current context is specified by incrementally adding and removing suppositions and focus objects. The system maintains a stack discipline with respect to the addition and removal of focus objects: the last supposition or focus object added must be the first one removed. The graph labeling of a given context is determined by the lemma library, focus objects and suppositions; the graph labeling does not depend on how the context was constructed. Labelings can be computed incrementally however. When a focus object or supposition is added Ontic's inference mechanisms extend the labeling to include more truth labels and to satisfy more equivalences. The system also maintains an "undo list" so that when a focus object or supposition is removed the previous context can be restored and then updated to reflect additions to the lemma library.

Chapters 6 and 7 specify the formal language Ontic and the way in which the graph structure is generated from the lemma library. This chapter, and the one that follows, specify the formal structure of the graph and the mechanisms for labeling that graph. The graphs constructed by the Ontic compiler have five different kinds of nodes and nine different kinds of "links" between nodes. However, this chapter discusses only those kinds of nodes and links that are used in Boolean constraint propagation and congruence closure. These node types and link types are introduced in three stages by defining three progressively more sophisticated types of graphs.

The first two sections of this chapter discuss graph structure and inference mechanisms that are relevant to Boolean constraint propagation. Boolean constraint propagation is responsible for enforcing certain Boolean constraints on formula nodes and for enforcing certain relationships between truth labels of equation nodes and color labels representing equivalences. Congruence closure ensures that the color labels that represent equivalences respect the substitution of equals for equals.

4.1 Boolean Constraint Graphs

This section describes Boolean constraint graphs and the inference mechanisms that apply to them. Sections 4.1.2 and 4.1.3 can be safely ignored by readers who are not interested in correctness proofs; the graph structure and inference mechanisms are fully specified by the end of section 4.1.1.

Boolean constraint graphs are a very simple approximation of the graphs produced by the Ontic compiler; Boolean constraint graphs have only a single kind of node and a single kind of link. The nodes represent formulas and each link is a disjunctive constraint on truth values assigned to the nodes.

> **Definition:** Let $\mathcal{N}$ be a set of formula nodes. A *literal* Ψ over $\mathcal{N}$ is either a node n in $\mathcal{N}$ or the negation $\neg n$ of some node n in $\mathcal{N}$.
>
> A *clause* over $\mathcal{N}$ is a disjunction of the form
>
> $$\Psi_1 \vee \Psi_2 \vee \ldots \Psi_n$$
>
> where each Ψ_i is a literal over $\mathcal{N}$.
>
> A *Boolean constraint graph* $\mathcal{B}$ consists of a set of formula nodes and a set of clauses over those nodes.

The Boolean constraint propagation algorithm manipulates partial truth labelings of Boolean constraint graphs. More specifically, the propagation algorithm extends partial truth labelings in a manner justified by the clauses in the graph.

> **Definition:** A *partial truth labeling* γ of Boolean constraint graph $\mathcal{B}$ is a partial map from the nodes in $\mathcal{B}$ to the set $\{\textbf{true}, \textbf{false}\}$; if n is a node in $\mathcal{B}$ then $\gamma(n)$ is either **true**, **false** or undefined.
>
> A partial truth labeling γ on $\mathcal{B}$ determines a partial truth labeling on all literals Ψ over $\mathcal{B}$ as follows:
>
> $$\gamma(\neg n) = \begin{cases} \textbf{false} & \text{if } \gamma(n) = \textbf{true} \\ \textbf{true} & \text{if } \gamma(n) = \textbf{false} \\ \text{undefined} & \text{if } \gamma(n) \text{ is undefined} \end{cases}$$

Each clause is a disjunction of the form

$$\Psi_1 \vee \Psi_2 \vee \ldots \Psi_n$$

which states that one of the literals must be true. The propagation algorithm is based on the notion of a *unit clause*; Boolean constraint propagation extends partial truth labels by identifying unit clauses in the graph structure. The notion of a unit clause is defined relative to the partial truth labeling γ. Consider a clause of the form

$$\Psi_1 \vee \Psi_2 \vee \ldots \Psi_n$$

and a partial truth label γ. If $\gamma(\Psi_i)$ is **false** then the above clause expresses the constraint that one of the other literals must be true. In general one should only pay attention to the non-false literals in a clause. A clause with only a single non-false literal is called a unit clause.

Definition: A clause $\Psi_1 \vee \Psi_2 \vee \ldots \Psi_n$ is called a γ-*unit-clause* if there is exactly one literal Ψ_i such that $\gamma(\Psi_i)$ is not **false**. The single non-false literal is called the *unit literal* of the clause.

An *open* γ-*unit-clause* is a γ-unit-clause where the unit literal has no truth label under γ, i.e. $\gamma(\Psi)$ is undefined for the unit literal Ψ.

An open γ-unit-clause provides grounds for extending the partial truth labeling γ; if there is only one non-false literal in a clause C then the remaining literal, the unit literal of the clause, must be true. Boolean constraint propagation uses open unit clauses to extend the truth labeling until either an inconsistency is discovered or there are no remaining open unit clauses.

Definition: Let $\mathcal{B}$ be a Boolean constraint graph and let γ be a partial truth labeling on $\mathcal{B}$.

The partial labeling γ will be called $\mathcal{B}$-*inconsistent* if there is some clause

$$\Psi_1 \vee \Psi_2 \vee \ldots \Psi_n$$

in $\mathcal{B}$ such that $\gamma(\Psi_i)$ is **false** for each literal Ψ_i in the clause. If γ is not $\mathcal{B}$-inconsistent we say that γ is $\mathcal{B}$-consistent.

Let Ψ be any literal over the nodes in $\mathcal{B}$ such that $\gamma(\Psi)$ is undefined. The labeling $\gamma[\Psi := \mathbf{true}]$ is the partial truth labeling which agrees with γ on all nodes other than that appearing in Ψ and such that $\gamma[\Psi := \mathbf{true}](\Psi)$ equals **true**. $\gamma[\Psi := \mathbf{false}]$ is defined similarly.

Boolean constraint propagation starts with an arbitrary partial labeling γ of a Boolean constraint graph $\mathcal{B}$ and returns a new partial labeling $N_{\mathcal{B}}(\gamma)$. The Boolean constraint propagation procedure can be defined as follows:

Definition: A partial truth labeling γ of a Boolean constraint graph $\mathcal{B}$ is called *normalized* if either it is $\mathcal{B}$-inconsistent or there are no open unit clauses in $\mathcal{B}$ under γ.

Procedure for Computing $N_{\mathcal{B}}(\gamma)$:

If γ is normalized then return γ, otherwise choose an open γ-unit-clause in $\mathcal{B}$ with unit literal Ψ and return the labeling $N_{\mathcal{B}}(\gamma[\Psi := \mathbf{true}])$.

Since there are only finitely many formula nodes in $\mathcal{C}$ the partial truth labeling can not be extended indefinitely and the recursion in the above procedure must terminate. Furthermore the labeling returned by the above procedure is always normalized.

The normalization of a labeling of a Boolean constraint graph involves inference. If a labeling γ' can be derived via a single inference from a labeling γ then we write $\gamma \rightarrow_{\mathcal{B}} \gamma'$. In analyzing Ontic's inference mechanisms the one step inference relation $\rightarrow_{\mathcal{B}}$ is easier to think about than the normalization function $N_{\mathcal{B}}$. More formally, for any Boolean constraint graph $\mathcal{B}$ the relation $\rightarrow_{\mathcal{B}}$ is defined on the labelings of $\mathcal{B}$ as follows:

Definition: Let γ and γ' be two partial truth labelings of a Boolean constraint graph $\mathcal{B}$. We write $\gamma \rightarrow_{\mathcal{B}} \gamma'$ if γ is $\mathcal{B}$-consistent and γ' can be derived in a single unit inference from γ, i.e. if there is some open γ-unit-clause in $\mathcal{B}$ with unit literal Ψ and such that γ' equals $\gamma[\Psi := \mathbf{true}]$.

The relation $\rightarrow_{\mathcal{B}}$ should be viewed as a reduction relation analogous to reduction relations in the lambda calculus or term rewriting systems. For any labeling γ of $\mathcal{B}$ the normalized labeling $N_{\mathcal{B}}(\gamma)$ is the normalization of γ under the reduction relation $\rightarrow_{\mathcal{B}}$.

4.1.1 Compiling Boolean Combinations

The graph structure used in semantic modulation is constructed by compiling expressions in the Ontic language; the compilation process translates the Ontic expressions into graph structure. The utility of Boolean constraint propagation is best understood in light of this compilation process. The full Ontic compiler is precisely defined in chapter 7. However this section describes the compilation of Boolean combinations of formulas.

The compilation process converts an Ontic formula Φ to a formula node n_Φ. Certain Ontic formulas are associated with clauses called *meaning postulates.* When the node n_Φ is constructed the meaning postulates for Φ are added to the graph. For example suppose that the formula Φ is a Boolean combination of the formulas Θ_1 and Θ_2, e.g. Φ might be the formula `(OR` Θ_1 Θ_2`)`. The meaning postulates for Φ are clauses that relate the node n_Φ to the nodes n_{Θ_1} and n_{Θ_2}. The exact nature of the clauses relating n_Φ to n_{Θ_1} and n_{Θ_2} depends on the Boolean connective used in Φ. Table 4.1 shows the meaning postulates for the Boolean connectives used in the Ontic system.

Boolean constraint propagation generates a normalized partial truth labeling of the constraint graph generated by the compilation process. If the normalized labeling is $\mathcal{B}$-consistent then the meaning postulates for Boolean connectives ensure certain relationships between Boolean formulas and their subformulas. For example consider the following meaning postulate for im-

Formula Φ	**Meaning Postulates for n_Φ**	
`(AND` $\Theta_1\ \Theta_2$`)`	$\neg n_{\mathtt{(AND}\ \Theta_1\ \Theta_2\mathtt{)}} \vee n_{\Theta_1}$	i.e. $n_{\mathtt{(AND}\ \Theta_1\ \Theta_2\mathtt{)}} \Rightarrow n_{\Theta_1}$
	$\neg n_{\mathtt{(AND}\ \Theta_1\ \Theta_2\mathtt{)}} \vee n_{\Theta_2}$	i.e. $n_{\mathtt{(AND}\ \Theta_1\ \Theta_2\mathtt{)}} \Rightarrow n_{\Theta_1}$
	$\neg n_{\Theta_1} \vee \neg n_{\Theta_2} \vee n_{\mathtt{(AND}\ \Theta_1\ \Theta_2\mathtt{)}}$	i.e. $n_{\Theta_1} \wedge n_{\Theta_2} \Rightarrow n_{\mathtt{(AND}\ \Theta_1\ \Theta_2\mathtt{)}}$
`(OR` $\Theta_1\ \Theta_2$`)`	$\neg n_{\Theta_1} \vee n_{\mathtt{(OR}\ \Theta_1\ \Theta_2\mathtt{)}}$	i.e. $n_{\Theta_1} \Rightarrow n_{\mathtt{(OR}\ \Theta_1\ \Theta_2\mathtt{)}}$
	$\neg n_{\Theta_2} \vee n_{\mathtt{(OR}\ \Theta_1\ \Theta_2\mathtt{)}}$	i.e. $n_{\Theta_1} \Rightarrow n_{\mathtt{(OR}\ \Theta_1\ \Theta_2\mathtt{)}}$
	$\neg n_{\mathtt{(OR}\ \Theta_1\ \Theta_2\mathtt{)}} \vee n_{\Theta_1} \vee n_{\Theta_2}$	i.e. $n_{\mathtt{(OR}\ \Theta_1\ \Theta_2\mathtt{)}} \Rightarrow n_{\Theta_1} \vee n_{\Theta_2}$
`(IMPLIES` $\Theta_1\ \Theta_2$`)`	$\neg n_{\Theta_2} \vee n_{\mathtt{(IMPLIES}\ \Theta_1\ \Theta_2\mathtt{)}}$	i.e. $n_{\Theta_1} \Rightarrow n_{\mathtt{(IMPLIES}\ \Theta_1\ \Theta_2\mathtt{)}}$
	$n_{\Theta_1} \vee n_{\mathtt{(IMPLIES}\ \Theta_1\ \Theta_2\mathtt{)}}$	i.e. $\neg n_{\Theta_1} \Rightarrow n_{\mathtt{(IMPLIES}\ \Theta_1\ \Theta_2\mathtt{)}}$
	$\neg n_{\mathtt{(IMPLIES}\ \Theta_1\ \Theta_2\mathtt{)}} \vee \neg n_{\Theta_1} \vee n_{\Theta_2}$	i.e. $n_{\mathtt{(IMPLIES}\ \Theta_1\ \Theta_2\mathtt{)}} \wedge n_{\Theta_1} \Rightarrow n_{\Theta_2}$
`(IFF` $\Theta_1\ \Theta_2$`)`	$\neg n_{\mathtt{(IFF}\ \Theta_1\ \Theta_2\mathtt{)}} \vee \neg n_{\Theta_1} \vee n_{\Theta_2}$	i.e. $n_{\mathtt{(IFF}\ \Theta_1\ \Theta_2\mathtt{)}} \wedge n_{\Theta_1} \Rightarrow n_{\Theta_2}$
	$\neg n_{\mathtt{(IFF}\ \Theta_1\ \Theta_2\mathtt{)}} \vee n_{\Theta_1} \vee \neg n_{\Theta_1}$	i.e. $n_{\mathtt{(IFF}\ \Theta_1\ \Theta_2\mathtt{)}} \wedge \neg n_{\Theta_1} \Rightarrow \neg n_{\Theta_2}$
	$\neg n_{\Theta_1} \vee \neg n_{\Theta_2} \vee n_{\mathtt{(IFF}\ \Theta_1\ \Theta_2\mathtt{)}}$	i.e. $n_{\Theta_1} \wedge n_{\Theta_2} \Rightarrow n_{\mathtt{(IFF}\ \Theta_1\ \Theta_2\mathtt{)}}$
	$n_{\Theta_1} \vee n_{\Theta_2} \vee n_{\mathtt{(IFF}\ \Theta_1\ \Theta_2\mathtt{)}}$	i.e. $\neg n_{\Theta_1} \wedge \neg n_{\Theta_2} \Rightarrow n_{\mathtt{(IFF}\ \Theta_1\ \Theta_2\mathtt{)}}$
`(NOT` Θ`)`	$n_{\Theta} \vee n_{\mathtt{(Not}\ \Theta\mathtt{)}}$	
	$\neg n_{\Theta} \vee \neg n_{\mathtt{(Not}\ \Theta\mathtt{)}}$	

Table 4.1: Meaning postulates for Boolean connectives

plications of the form (IMPLIES Θ_1 Θ_2)

$$\neg n_{(\texttt{IMPLIES}\ \Theta_1\ \Theta_2)} \vee \neg n_{\Theta_1} \vee n_{\Theta_2}$$

Now suppose γ is a $\mathcal{B}$-consistent normalized partial truth labeling such that $\gamma(n_{(\texttt{IMPLIES}\ \Theta_1\ \Theta_2)})$ is **true** and $\gamma(n_{\Theta_1})$ is **true**. In this case the first two literals in the above clause are labeled **false** under γ. By assumption γ is $\mathcal{B}$-consistent so the last literal is not false. Furthermore since γ is assumed to be normalized the above clause can not be an open γ-unit-clause so the last literal must be labeled **true**. In summary:

If γ is a $\mathcal{B}$-consistent normalized labeling such that

$$\gamma(n_{(\texttt{IMPLIES}\ \Theta_1\ \Theta_2)}) = \textbf{true}$$

and

$$\gamma(n_{\Theta_1}) = \textbf{true}$$

then

$$\gamma(n_{\Theta_2}) = \textbf{true}$$

Thus $\mathcal{B}$-consistent normalized labelings are closed under the inference rule of modus ponens. A similar argument can be used to prove the following:

If γ is a $\mathcal{B}$-consistent normalized labeling such that

$$\gamma(n_{(\texttt{IMPLIES}\ \Theta_1\ \Theta_2)}) = \textbf{true}$$

and

$$\gamma(n_{\Theta_2}) = \textbf{false}$$

then

$$\gamma(n_{\Theta_1}) = \textbf{false}$$

A similar argument concerning the meaning postulates for negations shows that if γ is a $\mathcal{B}$-consistent normalized partial truth labeling and the nodes

n_Θ and $n_{(\text{NOT } \Theta)}$ have been constructed in the graph then either γ does not provide a truth label for either of these nodes or the γ assigns these nodes opposite labels.

Now let op be any binary Boolean operator listed in table 4.1 and let γ be a $\mathcal{B}$-consistent normalized truth labeling. The meaning postulates ensure the following conditions:

- If the nodes n_{Θ_1} and n_{Θ_2} both have truth labels then any node of the form $n_{(\text{op } \Theta_1 \ \Theta_2)}$, also has a truth label; $n_{(\text{op } \Theta_1 \ \Theta_2)}$ has the truth label given by the meaning of op.

- If the meaning of op allows the truth of (op Θ_1 Θ_2) to be derived from either the truth label for n_{Θ_1} or the truth label for (or n_{Θ_2}) then $n_{(\text{op } \Theta_1 \ \Theta_2)}$ has the appropriate truth label. For example a disjunction is true whenever one of its disjuncts is true and a conjunction is false whenever one of its conjuncts is false.

- If the meaning of op allows the truth of n_{Θ_1} to be derived from the truth label of $n_{(\text{op } \Theta_1 \ \Theta_2)}$ then n_{Θ_1} has the appropriate truth label. For example if a conjunction is true then each conjunct is true and if a disjunction is false then each disjunct is false. If an implication is false then its antecedent is true and its consequent is false.

- If the meaning of op allows the truth of n_{Θ_1} to be derived from both the truth label of $n_{(\text{op } \Theta_1 \ \Theta_2)}$ and the truth label of n_{Θ_2} then n_{Θ_1} has the appropriate truth label. An analogous statement holds for deriving labelings of n_{Θ_2} from labelings of $n_{(\text{op } \Theta_1 \ \Theta_2)}$ and n_{Θ_1}. For example if a conjunction is labeled **false** and one of its conjuncts is labeled **true** then other will be labeled **false**. If a disjunction is labeled **true** and one of its disjuncts are labeled **false** then the other disjunct will be labeled **true**.

The above properties of a $\mathcal{B}$-consistent normalized labeling γ do not guarantee that γ is closed under all possible Boolean inferences. Boolean constraint propagation constructs a normalized labeling in time proportional to the number of nodes in the graph; assuming $P \neq NP$ any logically complete Boolean inference mechanism requires exponential time. Thus it is not surprising that Boolean constraint propagation is logically incomplete. More specifically, Boolean constraint propagation does not perform case analyses. For example there exists a $\mathcal{B}$-consistent normalized labeling γ with the following properties:

$$\gamma(n_{(\texttt{OR}\ \Theta_1\ \Theta_2)}) = \mathbf{true}$$

$$\gamma(n_{(\texttt{IMPLIES}\ \Theta_1\ \Theta_3)}) = \mathbf{true}$$

$$\gamma(n_{(\texttt{IMPLIES}\ \Theta_2\ \Theta_3)}) = \mathbf{true}$$

$$\gamma(n_{\Theta_3}) \text{ is undefined}$$

In the above situation Boolean constraint propagation does not generate truth labels for any of the nodes n_{Θ_1}, n_{Θ_2} or n_{Θ_3}.

4.1.2 Order Independence for Boolean Inference

The Boolean constraint propagation procedure defined above is non-deterministic; the procedure extends a partial truth labeling by non-deterministically choosing an open unit clause. Fortunately however, one can prove that the labeling generated by the propagation procedure is independent of the order in which open unit clauses are chosen.

Definition: Two partial labelings γ_1 and γ_2 of a Boolean constraint graph $\mathcal{B}$ will be called *$\mathcal{B}$-equivalent* if either γ_1 equals γ_2 or both γ_1 and γ_2 are $\mathcal{B}$-inconsistent.

Normalization Theorem: For any partial labeling γ of a Boolean constraint graph $\mathcal{B}$ the Boolean constraint propagation procedure terminates and all possible values of $N_{\mathcal{B}}(\gamma)$ are $\mathcal{B}$-equivalent.

This theorem can be proven by examining the inference relation $\rightarrow_{\mathcal{B}}$. Viewing $\rightarrow_{\mathcal{B}}$ as a reduction relation, the above theorem is implied by the fact that the relation $\rightarrow_{\mathcal{B}}$ satisfies a certain Church-Rosser property. The

Church-Rosser property of $\rightarrow_{\mathcal{B}}$ is proven using general lemmas that apply to any reduction relation.

Definition: For any binary relation $\rightarrow$ we write $x \rightarrow^* y$ if either x equals y or there exists some z such that $x \rightarrow z$ and $z \rightarrow^* y$.

We say that $\rightarrow$ is *well founded* if there is no infinite sequence

$$x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow \ldots$$

We say that y is a *normal form* under $\rightarrow$ if there is no z such that $y \rightarrow z$. We say that y is a normal form of x under $\rightarrow$ if y is a normal form under $\rightarrow$ and $x \rightarrow^* y$.

We say say that $\rightarrow$ is a *terminating normalizer* modulo an equivalence relation $\approx$ if $\rightarrow$ is well founded and normalizations under $\rightarrow$ are unique up to $\approx$, i.e. if y and z are both normal forms of x then $y \approx z$.

$\rightarrow_{\mathcal{B}}$ **Normalization Lemma:** $\rightarrow_{\mathcal{B}}$ is a terminating normalizer modulo $\mathcal{B}$-equivalence.

To prove the normalization lemma first note that whenever $\gamma \rightarrow_{\mathcal{B}} \gamma'$ the labeling γ' provides more truth labels than does γ. Since there are only finitely many nodes in $\mathcal{B}$ there can not be any infinitely long reduction chains under the relation $\rightarrow_{\mathcal{B}}$. Thus $\rightarrow_{\mathcal{B}}$ is well founded. Thus, to prove that $\rightarrow_{\mathcal{B}}$ is a terminating normalizer it suffices to show that normal forms are unique up to $\mathcal{B}$-equivalence.

Definition: We say that $\rightarrow$ satisfies the *diamond property* modulo an equivalence relation $\approx$ if for every x, y and z such that $x \rightarrow y$ and $x \rightarrow z$ there exists a w and w' such that $y \rightarrow^* w$, $z \rightarrow^* w'$ and $w \approx w'$.

Diamond Lemma: If $\rightarrow$ is well founded and satisfies the diamond property modulo $\approx$ then for any object x in the domain of the relation $\rightarrow$, all normal forms of x under $\rightarrow$ are equivalent under $\approx$, i.e. $\rightarrow$ is a terminating normalizer modulo $\approx$.

The diamond lemma as stated above is a straightforward modification of a theorem proved by Knuth and Bendix for term rewrite systems [Knuth & Bendix 69]. The diamond property for a given relation can be proven by showing that individual inferences commute. More specifically if there are two open unit clauses which each can be used to extend the partial truth labeling in two different ways then one can perform both inferences and the result is the same no matter which inference is performed first. Unfortunately the situation is complicated by the possibility of contradictions but the basic result holds: $\rightarrow_{\mathcal{B}}$ satisfies the diamond property modulo $\mathcal{B}$-equivalence of partial truth labelings.

Lemma: $\rightarrow_{\mathcal{B}}$ satisfies the diamond property modulo $\mathcal{B}$-equivalence.

Proof: Suppose $\gamma_0 \rightarrow_{\mathcal{B}} \gamma_1$ and $\gamma_0 \rightarrow_{\mathcal{B}} \gamma_2$ where γ_1 is a different labeling from γ_2. From the definition of $\rightarrow_{\mathcal{B}}$ there must exist distinct literals Ψ_1 and Ψ_2 such that

$$\gamma_1 = \gamma_0[\Psi_1 := \mathbf{true}]$$

and

$$\gamma_2 = \gamma_0[\Psi_2 := \mathbf{true}]$$

Let c_1 be the clause in $\mathcal{B}$ which is an open γ_0-unit-clause with unit literal Ψ_1 and let c_2 be the clause in $\mathcal{B}$ which is an open γ_0-unit-clause with unit literal Ψ_2.

First suppose that Ψ_1 and Ψ_2 are opposite literals for the same formula node. In this case the assignment $\Psi_1 :=\mathbf{true}$ will cause Ψ_2 to be false. Thus every literal in c_2 will be false under γ_1 so in this case γ_1 is $\mathcal{B}$-inconsistent. Similarly every literal in c_1 will be false under γ_2 and so in this case γ_2 is $\mathcal{B}$-inconsistent. But if

γ_1 and γ_2 are both $\mathcal{B}$-inconsistent then they are $\mathcal{B}$-equivalent so the diamond property holds.

Now suppose that the literals Ψ_1 and Ψ_2 involve different formula nodes. Let γ_3 be the labeling

$$(\gamma[\Psi_1 := \textbf{true}])[\Psi_2 := \textbf{true}]$$

Since Ψ_1 and Ψ_2 involve different formula nodes γ_3 can also be written as

$$(\gamma[\Psi_2 := \textbf{true}])[\Psi_1 := \textbf{true}]$$

Since Ψ_1 and Ψ_2 involve different formula nodes the clause c_2 is still an open γ_1-unit-clause. Thus if γ_1 is $\mathcal{B}$-consistent $\gamma_1 \rightarrow_{\mathcal{B}} \gamma_3$. Similarly if γ_2 is $\mathcal{B}$-consistent then $\gamma_2 \rightarrow_{\mathcal{B}} \gamma_3$. Thus if both γ_1 and γ_2 are $\mathcal{B}$-consistent then they both reduce to γ_3 so the diamond property holds. If both γ_1 and γ_2 are $\mathcal{B}$-inconsistent then they are $\mathcal{B}$-equivalent so the diamond property holds. Now suppose that γ_1 is $\mathcal{B}$-consistent but γ_2 is not. In this case γ_1 reduces to γ_3. But γ_3 is a proper extension of γ_2 and γ_2 is $\mathcal{B}$-inconsistent so γ_3 must also be $\mathcal{B}$-inconsistent. But this implies that γ_3 is $\mathcal{B}$-equivalent to γ_2 so the diamond property holds.

Since $\rightarrow_{\mathcal{B}}$ is well founded and satisfies the diamond property modulo $\mathcal{B}$-equivalence for partial truth labelings the Knuth-Bendix diamond lemma implies that normalizations are unique up to $\mathcal{B}$-equivalence and thus $\rightarrow_{\mathcal{B}}$ is a terminating normalization relation modulo $\mathcal{B}$-equivalence. Thus, up to $\mathcal{B}$-equivalence, there is only one possible value of $N_{\mathcal{B}}(\gamma)$.

4.1.3 Semantic Soundness

For any Boolean constraint graph $\mathcal{B}$ the relation $\rightarrow_{\mathcal{B}}$ can be viewed as an inference relation. It is possible to provide a simple semantics for Boolean constraint graphs and prove that the relation $\rightarrow_{\mathcal{B}}$ is sound modulo this semantics. For the most part the soundness of $\rightarrow_{\mathcal{B}}$ is self evident. However the semantics given here provides groundwork that will be needed to prove the soundness of semantic modulation inference relations.

Any semantic interpretation of a set of formula nodes provides a way of assigning every node a truth value, either **true** or **false**. Thus any semantic interpretation of a set of formula nodes yields a *complete* truth labeling of those nodes.

> **Definition**: A partial truth labeling of a Boolean constraint graph $\mathcal{B}$ is called *complete* if it assigns every node a truth label. Complete labelings will be called *Boolean interpretations* and will be denoted with the greek letter ω.

Clauses in a Boolean constraint graph and any partial truth labelings express constraints on possible interpretations.

> **Definition**: Let $\mathcal{B}$ be a Boolean constraint graph, let γ be a partial truth assignment on the nodes in $\mathcal{B}$, and let ω be a Boolean interpretation of the nodes in $\mathcal{B}$.
>
> We say that ω *satisfies* a clause
>
> $$\Psi_1 \vee \Psi_2 \vee \ldots \Psi_k$$
>
> if ω makes at least one of the literals Ψ_i true. We say that ω satisfies the Boolean constraint graph $\mathcal{B}$ just in case ω satisfies every clause in $\mathcal{B}$.
>
> We say that ω *satisfies* the partial truth labeling γ if every node that is assigned a truth label by γ is assigned the same truth label by ω.

The reduction relation $\rightarrow_{\mathcal{B}}$ can be viewed as a sound inference relation in the sense that if $\gamma_1 \rightarrow_{\mathcal{B}} \gamma_2$ then every constraint in γ_2 is implied by the constraints in γ_1 and $\mathcal{B}$, i.e. if ω satisfies γ_1 and $\mathcal{B}$ then ω also satisfies γ_2.

> $\rightarrow_{\mathcal{B}}$ **Soundness Lemma**: If ω is a Boolean interpretation that satisfies a Boolean constraint graph $\mathcal{B}$ and a partial truth labeling γ, and if $\gamma \rightarrow_{\mathcal{B}} \gamma'$, then ω satisfies γ'.

4.2 Equality Constraint Graphs

This section describes equality constraint graphs and the inference mechanisms that apply to them. Sections 4.2.1 and 4.2.2 can be safely ignored by readers who are not interested in correctness proofs.

As the name implies, equality constraint graphs are used to reason about equality. In addition to clause links equality graphs have equality links. An equality expresses the fact that a certain formula node represents an equation between two other nodes. Equality constraint graphs have both formula and non-formula nodes. The non-formula nodes in an equality constraint graph are divided into two types: quotation nodes and non-formula non-quotation nodes. No two quotation nodes should ever be equal. If there are n quotation nodes then there are order n^2 potential equalities between these nodes; the existence of quotation nodes eliminates the need to explicitly state that these n^2 equalities are all false. In the Ontic compilation process quotation nodes are used to represent quotation expressions of the form `(QUOTE` *symbol*`)`.

> **Definition:** An *equality constraint graph* $\mathcal{E}$ consists of a set of formula nodes, a set of clause links over the formula nodes, a set of quotation nodes, a set of non-formula non-quotation nodes, and a set of equality links of the form
>
> $$p \Leftrightarrow n = m$$
>
> where p is a formula node in $\mathcal{E}$ and n and m are any nodes in $\mathcal{E}$.
>
> Let $\mathcal{B}$ be the Boolean constraint graph consisting of the formula nodes and clause links in an equality constraint graph $\mathcal{E}$. We say that $\mathcal{B}$ is the Boolean constraint graph *underlying* $\mathcal{E}$.

An equality link of the form $p \Leftrightarrow n = m$ says that the formula node p represents the equality between nodes n and m. The Ontic compiler creates an equality link every time it compiles an equality formula. More specifically, every time a node of the form $n_{\texttt{(= } a \ b\texttt{)}}$ is created the system constructs the equality link

$$n_{(= \; a \; b)} \;\Leftrightarrow\; n_a = n_b$$

where n_a is the node representing the expression a and n_b is the node representing the expression b.

The labelings of equality graphs contains both a partial truth labeling of formula nodes and a color labeling of all nodes. The color labeling represents information about the equality of nodes; two nodes with the same color are considered equal.

Definition: A *labeling* $\mathcal{L}$ of a colorable node set $\mathcal{E}$ is a pair $<\gamma, \; \kappa>$ where γ is a partial truth labeling of the formula nodes in $\mathcal{E}$ and κ is a color labeling which maps every node in $\mathcal{E}$ to a color.

The notion of a labeling as defined above is meaningful independent of the links in the graph structure $\mathcal{E}$. A labeling contains information about which formula nodes are true (or false) and information about equivalences between nodes (both equivalences between formula nodes and equivalences between non-formula nodes). However the links in an equality constraint graph $\mathcal{E}$ can be thought of as constraints on labelings. More specifically, we have the following definition of a $\mathcal{E}$-inconsistent labeling.

Definition: We say that a labeling $<\gamma, \; \kappa>$ of $\mathcal{E}$ is *$\mathcal{E}$-inconsistent* if any of the following conditions hold:

- γ is $\mathcal{B}$-inconsistent where $\mathcal{B}$ is the Boolean constraint graph underlying $\mathcal{E}$.
- There is some equality link $p \;\Leftrightarrow\; n = m$ in $\mathcal{E}$ such that $\kappa(n) = \kappa(m)$ but $\gamma(p) =$ **false**.
- There are two distinct quotation nodes n and m in $\mathcal{E}$ such that $\kappa(n) = \kappa(m)$.

- There are two formula nodes p and q such that $\kappa(p) = \kappa(q)$, both $\gamma(p)$ and $\gamma(q)$ are defined but $\gamma(p)$ is the opposite of $\gamma(q)$.

If a labeling $\mathcal{L}$ is not $\mathcal{E}$-inconsistent then we say that the labeling $<\gamma, \kappa>$ is *$\mathcal{E}$-consistent.*

A given equality constraint graph $\mathcal{E}$ is associated with an inference relation $\rightarrow_{\mathcal{E}}$ on labelings. The inference relation $\rightarrow_{\mathcal{E}}$ can extend a labeling in one of two ways: it can add a new truth label on a formula node or it can merge two equivalence classes by assigning both classes the same color label. When two equivalence classes are merged the smaller class is recolored to be the color of the larger class. This class merger operation can be defined as follows:

Definition: If κ is a color labeling of the nodes in $\mathcal{E}$, and n and m are nodes in $\mathcal{E}$ then the color map $\kappa[\text{union}(n, m)]$ is a color map which yields the same equivalence relation as κ except that the equivalence classes of n and m have been merged. More specifically, if the size of the equivalence class of n under κ is less than or equal to the size of the class of m under κ then the map $\kappa[\text{union}(n, m)]$ is defined as follows:

$$\kappa[\text{union}(n,m)](q) = \begin{cases} \kappa(m) & \text{if } \kappa(q) = \kappa(n) \\ \kappa(q) & \text{otherwise} \end{cases}$$

The above definition specifies that the union operation recolors the class of n to be the same color as the class of m. If the size of the class of n under κ is larger than the size of the class of m under κ then $\kappa[\text{union}(n, m)]$ equals $\kappa[\text{union}(m, n)]$. The union operation always recolors the smaller equivalence class.

It is now possible to define the inference relation $\rightarrow_{\mathcal{E}}$.

Definition: Let $\mathcal{L}$ be a labeling of $\mathcal{E}$ which is equal to the pair $<\gamma, \kappa>$. Let $\mathcal{L}$' be a labeling of $\mathcal{E}$ which is equal to the pair $<\gamma', \kappa'>$. We write $\mathcal{L} \rightarrow_{\mathcal{E}} \mathcal{L}'$ if one of the following conditions hold:

- $\kappa = \kappa'$ and γ' is derived from γ via unit inference, i.e. $\gamma \rightarrow_{\mathcal{B}} \gamma'$ where $\mathcal{B}$ is the Boolean constraint graph underlying $\mathcal{E}$.
- $\mathcal{E}$ contains the link $p \Leftrightarrow n = m$ and each of the following conditions hold
  - $\gamma(p) = \textbf{true}$
  - $\kappa(n) \neq \kappa(m)$
  - $\gamma' = \gamma$ and $\kappa' = \kappa[\text{union}(n, m)]$
- $\mathcal{E}$ contains the link $p \Leftrightarrow n = m$ and each of the following conditions hold
  - $\kappa(n) = \kappa(m)$
  - $\gamma(p)$ is undefined
  - $\kappa' = \kappa$ and $\gamma' = \gamma[p := \textbf{true}]$
- $\mathcal{E}$ contains two formula nodes p and q such that the following conditions hold:
  - $\kappa(p) = \kappa(q)$
  - $\gamma(p)$ is defined but $\gamma(q)$ is not.
  - $\kappa' = \kappa$ and $\gamma' = \gamma[q := \gamma(p)]$

4.2.1 Semantic Soundness

Any semantic interpretation of an equality constraint graph provides both a truth labeling and a color labeling where two nodes have the same color just in case they denote the same semantic object. A labeling that corresponds to a semantic interpretation must be complete in that every formula node must have a truth label.

Definition: A labeling $\mathcal{L}$ of an equality constraint graph $\mathcal{E}$ is called *complete* if $\mathcal{L}$ assigns every formula node in $\mathcal{E}$ a truth label, either the label **true** or the label **false**. Complete labels are also called *possible worlds.*

The term "possible world" comes from modal logic; there is a strong similarity between the semantics of the graphs described in chapter 5 and the possible world semantics of modal logic. Clause links and equality links can both be viewed as constraints on possible worlds. A partial labeling can also be viewed as a constraint on possible worlds.

Definition: A possible world w *satisfies* an equality constraint graph $\mathcal{E}$ just in case the truth labeling of w satisfies every clause link in $\mathcal{E}$, no two quotation nodes of $\mathcal{E}$ are assigned the same color by w, any two formula nodes which are assigned the same color label by w are assigned the same truth label by w, and for every equality link $p \Leftrightarrow n = m$ in $\mathcal{E}$, the world w assigns p the label **true** just in case w assigns n and m the same color label.

A possible world w satisfies a labeling $\mathcal{L}$ of an equality constraint graph $\mathcal{E}$ just in case every formula node which is assigned a truth value by $\mathcal{L}$ is assigned the same truth value by w and if two nodes n and m are assigned the same color by $\mathcal{L}$ then n and m are assigned the same color by w.

The reduction relation $\rightarrow_{\mathcal{E}}$ can be viewed as a sound inference relation in the sense that if $\mathcal{L}_1 \rightarrow_{\mathcal{E}} \mathcal{L}_2$ then every constraint in $\mathcal{L}_2$ is implicitly present in $\mathcal{E}$ and $\mathcal{L}_1$, i.e. if an interpretation satisfies $\mathcal{E}$ and $\mathcal{L}_1$ then it also satisfies $\mathcal{L}_2$.

$\rightarrow_{\mathcal{E}}$ **Soundness Lemma**: If w is a possible world that satisfies the equality constraint graph $\mathcal{E}$ and the labeling $\mathcal{L}$, and if $\mathcal{L} \rightarrow_{\mathcal{E}} \mathcal{L}'$, then w satisfies $\mathcal{L}'$.

4.2.2 Termination and Order Independence

Note that if $\mathcal{L} \rightarrow_{\mathcal{E}} \mathcal{L}'$ then either $\mathcal{L}'$ provides more truth labels than $\mathcal{L}$ or $\mathcal{L}'$ has fewer colors (equivalences classes) than $\mathcal{L}$. Since there are only finitely many formula nodes that can take truth labels, and since the number of equivalence classes can not be reduced below one, the inference process must terminate, i.e. there are no infinite inference chains of the form

$$\mathcal{L}_1 \rightarrow_{\mathcal{E}} \mathcal{L}_2 \rightarrow_{\mathcal{E}} \mathcal{L}_3 \rightarrow_{\mathcal{E}} \ldots$$

Thus the relation $\rightarrow_{\mathcal{E}}$ is well founded.

To prove that $\rightarrow_{\mathcal{E}}$ yields a well defined normalization operation one must show that all normal forms of a labeling $\mathcal{L}$ are equivalent modulo some equivalence relation. This equivalence of normal forms can be established under the following equivalence relation.

Definition: Two labelings $\mathcal{L}$ and $\mathcal{L}'$ of a colorable node set $\mathcal{E}$ are called $\mathcal{E}$-*equivalent* if either both $\mathcal{L}$ and $\mathcal{L}'$ are $\mathcal{E}$-inconsistent or if they both provide the same partial truth labeling on the formula nodes in $\mathcal{E}$ and the color labelings in $\mathcal{L}$ and $\mathcal{L}'$ determine the same equivalence relation on $\mathcal{E}$.

$\rightarrow_{\mathcal{E}}$ **Normalization Lemma:** $\rightarrow_{\mathcal{E}}$ is a terminating normalizer relative to $\mathcal{E}$-equivalence.

The proof of the above theorem uses the Knuth-Bendix diamond lemma. The proof that $\rightarrow_{\mathcal{E}}$ satisfies the diamond property relative to $\mathcal{E}$-equivalence is similar to the proof that $\rightarrow_{\mathcal{B}}$ satisfies the diamond property relative to $\mathcal{B}$-equivalence; both proofs are based on the commutativity of individual inference reductions.

4.2.3 Running Time

The union operation used to construct $\kappa[\text{union}(n, m)]$ recolors the the smaller of the two equivalence classes. This has the important consequence that every time the color label of a node n changes the size of n's equivalence class at least doubles. Let $|\mathcal{E}|$ be the number of nodes in $\mathcal{E}$. The color label for a given node n can change at most $\lfloor \log_2 |\mathcal{E}| \rfloor$ times because if the color of n changed more than $\lfloor \log_2 |\mathcal{E}| \rfloor$ times the equivalence class of n would be larger than $|\mathcal{E}|$. Since the color of a given node n can change at most $\lfloor \log_2 |\mathcal{E}| \rfloor$ times the total number of coloring operations required to normalize a labeling $\mathcal{L}$ is at most $|\mathcal{E}| \lfloor \log_2 |\mathcal{E}| \rfloor$. Since the number of truth labeling operations is at most $|\mathcal{E}|$ the total number of labelings operations is order $|\mathcal{E}| \log |\mathcal{E}|$.

4.3 Congruence Constraint Graphs

This section describes congruence constraint graphs and the inference mechanisms that apply to them. Sections 4.3.1 and 4.3.2 can be safely ignored by readers who are not interested in correctness proofs.

Congruence constraint graphs are just like equality graphs except that they contain subexpression links. Subexpression links relate a node for a composite expression to nodes for its subexpressions. For example a subexpression link might relate the node representing the expression `(FOO A)` to the nodes representing `FOO` and `A`. The labeling process which uses subexpression links is called *congruence closure.* Congruence closure effectively performs the substitution of equals for equals. For example consider a color labeling such that the node for `A` and the node for `B` are assigned the same color and yet the nodes for `(FOO A)` and `(FOO B)` have different colors. This labeling would not respect the substitution of equals for equals. A color labeling is said to be *congruence closed* if it does respect the substitution of equals for equals.

Definition: A *congruence constraint graph* $\mathcal{C}$ is of an equality constraint graph augmented with a set of *subexpression links* of the form

$$(m_1\, m_2\, \ldots\, m_k) = n$$

where n and each m_i are nodes in $\mathcal{C}$.

Let $\mathcal{E}$ be the equality constraint graph derived from a congruence constraint graph $\mathcal{C}$ by deleting all subexpression links. We say that $\mathcal{E}$ is the equality constraint graph underlying $\mathcal{C}$.

A labeling of a congruence constraint graph is a labeling of the underlying equality constraint graph.

A subexpression link of the form $(m_1\, m_2\, \ldots\, m_k) = n$ says that the node n represents the application of the operator m_1 to the arguments $m_2\, \ldots\, m_k$. The Ontic compiler generates subexpression links whenever it compiles an applicative expression. Subexpression links can be used to define a new inference relation on labelings.

Definition: A labeling $\mathcal{L}$ of a congruence constraint graph $\mathcal{C}$ is called $\mathcal{C}$-consistent just in case $\mathcal{L}$ is $\mathcal{E}$-consistent where $\mathcal{E}$ is the equality constraint graph underlying $\mathcal{C}$.

For any two labelings $\mathcal{L}$ and $\mathcal{L}'$ of a congruence constraint graph $\mathcal{C}$ we write $\mathcal{L} \rightarrow_{\mathcal{C}} \mathcal{L}'$ just in case $\mathcal{L}$ is equality consistent and either:

- $\mathcal{L} \rightarrow_{\mathcal{E}} \mathcal{L}'$ where $\mathcal{E}$ is the equality constraint graph underlying $\mathcal{C}$.
- $\mathcal{L}'$ can be derived from $\mathcal{L}$ via a congruence inference, i.e. $\mathcal{L}$ is a pair $<\gamma,\ \kappa>$ such that there are two subexpression links $(n_1\, n_2\, \ldots\, n_k) = m$ and $(p_1\, p_2\, \ldots\, p_k) = q$ in $\mathcal{S}$ such that for each pair m_i and q_i of corresponding subnodes $\kappa(m_i) = \kappa(q_i)$ but $\kappa(n) \neq \kappa(p)$ and $\mathcal{L}'$ is the pair $<\gamma,\ \kappa[\text{union}(n, p)]>$.

If a labeling $\mathcal{L}$ is normalized relative to $\rightarrow_{\mathcal{C}}$ then there is no pair of subexpression links satisfying the conditions for congruence inference given in the definition of $\rightarrow_{\mathcal{C}}$. This implies that if $\mathcal{L}$ is normalized under $\rightarrow_{\mathcal{C}}$ then $\mathcal{L}$ is congruence closed.

4.3.1 Semantic Soundness

Recall that a *possible world* is a complete labeling, i.e. a color and truth labeling which assigns every formula node a truth label. The links in a congruence constraint graph can be viewed as constraints on possible worlds.

Definition: A possible world w *satisfies* a congruence constraint graph $\mathcal{C}$ just in case w satisfies the underlying equality constraint graph and for any two subexpression links

$$(m_1\, m_2\, \ldots\, m_k) = n$$

and

$$(p_1\, p_2\, \ldots\, p_k) = q$$

if for each m_i the world w assigns m_i and p_i the same color then w assigns n and q the same color.

The reduction relation $\rightarrow_{\mathcal{C}}$ can be viewed as a sound inference relation in the sense that if $\mathcal{L}_1 \rightarrow_{\mathcal{C}} \mathcal{L}_2$ then the constraints in $\mathcal{C}$ and $\mathcal{L}$ semantically imply the constraints in $\mathcal{L}'$.

$\rightarrow_{\mathcal{C}}$ **Soundness Lemma**: If w is a possible world that satisfies both a congruence constraint graph $\mathcal{C}$ and a labeling $\mathcal{L}$ of $\mathcal{C}$, and if $\mathcal{L} \rightarrow_{\mathcal{C}} \mathcal{L}'$, then w satisfies $\mathcal{L}'$.

4.3.2 Termination and Order Independence

If $\mathcal{L} \rightarrow_{\mathcal{C}} \mathcal{L}'$ then either $\mathcal{L}'$ provides more truth labels than $\mathcal{L}$ or $\mathcal{L}'$ provides fewer color labels, and thus allows fewer equivalence classes than $\mathcal{L}$. Since there can not be more truth labels than there are formula nodes, nor fewer equivalence classes than one, every reduction chain must terminate. Thus the relation $\rightarrow_{\mathcal{C}}$ is well founded.

To prove that $\rightarrow_{\mathcal{C}}$ yields a well defined normalization operation one must show that all normal forms of a labeling $\mathcal{L}$ are equivalent modulo some given equivalence relation.

> $\rightarrow_{\mathcal{C}}$ **Normalization Lemma:** $\rightarrow_{\mathcal{C}}$ is a terminating normalizer modulo $\mathcal{E}$-equivalence where $\mathcal{E}$ is the equality constraint graph underlying $\mathcal{C}$.

The above theorem is proved via the Knuth-Bendix diamond lemma and the proof that $\rightarrow_{\mathcal{C}}$ satisfies the diamond property is based on the commutativity of individual inferences.

4.3.3 Implementation Techniques

For any labeling $\mathcal{L}$ of a congruence constraint graph $\mathcal{C}$ we can define $N_{\mathcal{C}}(\mathcal{L})$ to be any normal form of $\mathcal{L}$ under the reduction relation $\rightarrow_{\mathcal{C}}$. The definition of $\rightarrow_{\mathcal{C}}$ specifies the value of $N_{\mathcal{C}}(\mathcal{L})$ up to $\mathcal{E}$-equivalence where $\mathcal{E}$ is the equality constraint graph underlying $\mathcal{C}$. Furthermore, because the size of a node's equivalence class at least doubles every time the node is assigned a new color, the normalization procedure involves at most order $|\mathcal{C}| \log |\mathcal{C}|$ labeling operations. The above specification however does not provide a complete description of an efficient implementation of the normalization function $N_{\mathcal{C}}$. More specifically no procedure has been given for finding the clauses, equality links, and subexpression links involved in a single step of the normalization process.

Most labeling inferences involve a single link in the graph structure; the inference is justified by a single link and the label of the nodes in that link. Boolean constraint propagation based on clause links, for example, always involves a single clause. There are certain inferences, however, that involve two objects that are not connected by any single link. For example, to test for consistency the system must determine if two quotation nodes have the same color label. To quickly test for the presence of two quotation nodes with the same color label one can maintain a hash table with entries of the form $c \mapsto n$ where c is a color and n is a quotation node. Every time a quotation node n is assigned a color c one checks the hash table to see if some other quotation node has been labeled with color c. If there is such a node, an inconsistency is flagged. If there is no such node then one adds a new entry to the hash table. This hash table can be maintained during the inference process. Assuming hash lookup takes constant time, the time needed to maintain this hash table is proportional to the number of color labeling operations.

Another example of an inference that involves two objects not related by a single link is congruence inference. Congruence inference, as defined in the previous section, requires finding two subexpression links which together justify a congruence inference. Let s be the number of subexpression links. Searching all pairs of subexpression links for a possible congruence inference might require order s^2 comparisons. Fortunately an additional data structure can be used to eliminate the need for s^2 comparisons.

Each labeling of a congruence constraint graph can be augmented with a hash table that maps tuples of colors to nodes. More specifically each labeling is associated with a set of hash table entries of the form

$$<c_1\, c_2 \ldots c_n> \mapsto n$$

where each c_i is a color and n is a node. Such a table entry corresponds to a subexpression link of the form

$$(m_1\, m_2\, \ldots\, m_k) = n$$

where each node m_i has color c_i. Using this hash table it is possible to quickly determine if there are two subexpressions links satisfying the conditions for congruence inference. Such a hash table can be incrementally maintained as a labeling is normalized.

Given the hash tables described above it is possible to determine if a labeling can be further reduced by independently examining individual links. If a given link ℓ can not be used to generate an inference then ℓ need not be checked again until some label changes for some node in ℓ. The total number of labeling operations performed on any given node is order $\log(n)$ where n is the number of nodes in the graph. If there is some upper bound on the number of nodes that appear in any given link then the number of times a given link needs to be checked is also order $\log(n)$. Thus, if e is the number of links in the graph, and n is the number of nodes, the total number of link checks is order $e \log(n)$ and the total number of labeling operations is order $n \log(n)$. Efficient congruence closure algorithms are described in [Downey, Sethi, & Tarjan 80].

Chapter 5

Inference with Quantifiers

Focused binding and automatic universal generalization are graph labeling inference processes that construct binding environments and quantified formulas. Certain nodes in the graph structure are identified as variable nodes. Graph labelings are used to represent variable bindings. For example if n is a variable node and r is some other node then the binding $n \mapsto r$ can be represented in a graph labeling by merging the equivalence classes of n and r. This graph theoretic binding mechanism forms the basis for an inheritance mechanism; a binding of the form $n \mapsto r$ causes information known to be true of the variable (or generic individual) n to be inherited by the particular instance r.

Ontic's inference mechanisms are fully described in sections 5.1, 5.4, 5.5 and 5.6; sections 5.2 and 5.3 can be safely ignored by readers who are not interested in correctness proofs.

5.1 Semantic Modulation Graphs

Semantic modulation graphs have two new kinds of nodes: variable nodes which represent variables and type nodes which represent types. Semantic modulation graphs also have two new kinds of links: type declaration links that associate a variable with a type and type assertion links each of which states that a certain formula node represents the statement that a certain object (node) is an instance of a certain type.

This section describes the inference relation $\rightarrow_{\mathcal{S}}$. The inference relation $\rightarrow_{\mathcal{S}}$ both performs inference and generates variable bindings. However, the relation $\rightarrow_{\mathcal{S}}$ is not guided by focus objects. Section 5.4 describes the relation $\rightarrow_{\mathcal{S}\mathcal{F}}$ which is similar to $\rightarrow_{\mathcal{S}}$ except that the generation of variable bindings is guided by a set $\mathcal{F}$ of focus objects.

Before defining semantic modulation graphs we define the preliminary notion of a variable graph. A semantic modulation graph is a variable graph that satisfies a certain non-circularity constraint.

Definition: A *variable graph* consists of a congruence constraint graph together with the following:

- a classification of the non-formula non-quotation nodes into variable nodes, type nodes, and unclassified nodes.

- A set of *free variable links* of the form

  $$n \ll r$$

  Where n is a variable node. Such a link says that n represents a variable that appears free in the expression represented by r.

- A set of *type declaration links*; for each variable node n there is exactly one type declaration link of the form

  $$n : m$$

  The node m is called the *type node* of n and n is called a variable of type m.

- A set of *type formula links* of the form

  $$p \Leftrightarrow r\!:\!m$$

  where p is a formula node, r is any node, and m is a type node. Such a link says that formula node p represents the statement that node r is an instance of the type represented by m.

- A set of *subtype links* of the form

  $$q \Leftrightarrow m \prec m'$$

  where q is a formula node and m and m' are type nodes. Such a link says that q represents the formula that m is a subtype of m', i.e. every instance of m is an instance of m'.

Let $\mathcal{C}$ by the congruence constraint graph derived from a variable graph $\mathcal{V}$ by removing all free variable links, type declaration links, type formula links, and subtype links. We say that $\mathcal{C}$ is the congruence constraint graph underlying $\mathcal{V}$.

It may seem that the free variable links are redundant; it seems that one could define the free variables of a node in terms of the subexpression links discussed in chapter 4. Since a semantic modulation graph is just a congruence graph with additional structure these subexpression links are part of a semantic modulation graph. Unfortunately the graph may contain nodes that represent lambda closures (functions, types, and type generators). These nodes represent expressions that contain free variables but these nodes are not involved in subexpression links in a way that allows the free variables to be determined from the subexpression links. Thus explicit free variable links are needed.

The semantic modulation inference mechanisms manipulate bindings of the form $n \mapsto r$ where n is a variable node. A binding of the form $n \mapsto r$ can be viewed as an instruction to set the value of the variable n to the node r. Changing the value of a given variable forces the values of certain other nodes to change. In ordinary predicate calculus changing the value of a variable x causes changes in the meanings of terms that contain x as a free variable; the meaning of expressions which do not contain x as a free variable will not change when x is changed. The situation in Ontic is slightly more complex. Suppose that x is a variable ranging over sets and that y is a variable of type (MEMBER-OF x). In this case changing the meaning of the variable x may force a change the meaning of the variable y even though x is not a free variable of y. In general if x is a variable which appears free in the type node of of another variable y then we say that y *depends on* x. This notion of dependency can be defined in terms of the structure of a variable graph.

> **Definition**: Let s be a node in a variable graph $\mathcal{V}$ and let n be a variable node in $\mathcal{V}$. We say that n is a *free variable of* s just in case $\mathcal{V}$ contains the free variable link $n \ll s$. We say that s *depends on* n just in case n is a free variable of s or there is some free variable n' of s such that the type node of n' depends on n.

The soundness (or validity) of the semantic modulation inference process relies on an additional property of graphs. More specifically, the soundness of the semantic modulation inference process requires that the type node of a variable n does not depend on n. Intuitively this condition allows one to assign the value of a variable without changing the type of the variable.

> **Definition**: A *semantic modulation graph* $\mathcal{S}$ is a variable graph such that for every variable node n the type node of n does not depend on n.

In addition to manipulating truth and color labels, the semantic modulation inference process manipulates *variable bindings.* More specifically, a state of the semantic modulation inference process contains both a truth and color labeling $\mathcal{L}$ and a binding set β where β contains bindings of the form $n \mapsto r$ where n is a variable node.

> **Definition**: Let $\mathcal{S}$ be a semantic modulation graph. A *binding set* β over $\mathcal{S}$ is a set of bindings of the form $n \mapsto r$ where n is

a variable node and r is any node in $\mathcal{S}$. We say that a variable node n in $\mathcal{S}$ is *bound under* β if β contains a binding of the form $n \mapsto r$. If n is not bound under β then n is called *β-free*.

In order to define the inference relation on semantic modulation graphs the notion of dependence needs to be defined relative to a binding set β. Recall that if s depends on n then changing the value of n may force a change in the value of s. Consider a binding of the from $n \mapsto r$. In the presence of the binding $n \mapsto r$ changing the value of r forces a change in the value of n; in the presence of the binding $n \mapsto r$ the variable n depends on r. This observation leads to the notion of β-dependence where β is any binding set. If s β-depends on n then, in the presence of the binding set β, changing the value of n may force a change in the value of s. The precise semantic significance of the following syntactic definition will be discussed in more detail in later sections.

Definition: Let β be a binding set over a semantic modulation graph $\mathcal{S}$.

We say that a node s β-depends on a variable node n if one of the following conditions hold:

- n is a free variable of s.
- There exists a free variable n' of s such that n' is bound under β with binding $n' \mapsto r$ and r β-depends on n.
- There exists a free variable n' of s such that n' is not bound under β, i.e. is β-free, and the type node for n' β-depends on n.

I will use the term *direct dependence* to refer to the standard notion of dependence as distinct from β-dependence. If β is empty then β-dependence is the same as direct dependence. In the definition of β-dependence the presence of a binding of the form $n \mapsto r$ causes the variable node n to be treated as a copy of the node r.

The inference relation $\rightarrow_{\mathcal{S}}$ for semantic modulation graphs operates on binding labelings where each binding labeling consists of a truth and color labeling together with a binding set.

Definition: Let $\mathcal{S}$ be a semantic modulation graph.

A *truth and color labeling* of $\mathcal{S}$ is a labeling $\mathcal{L}$ of the congruence constraint graph underlying $\mathcal{S}$.

A *binding labeling* $\mathcal{T}$ of $\mathcal{S}$ consists of a truth and color labeling $\mathcal{L}$ of $\mathcal{S}$ together with a binding set β over $\mathcal{S}$.

Before generating a binding of the form $n \mapsto r$ the system must be sure that r is an instance of the type of n. More specifically, for any given truth and color labeling $\mathcal{L}$ and any node r it is possible to collect a set of types known to contain r as an instance. These types are called the *established types* for r.

bf Definition: Let $\mathcal{L}$ be a truth and color labeling of a semantic modulation graph $\mathcal{S}$ and let r be any node in $\mathcal{S}$. The set of *$\mathcal{L}$-established-type-nodes* for r is the least set of type nodes satisfying the following conditions:

- If there exists a type formula link $p \Leftrightarrow r : m$ in $\mathcal{S}$ such that $\mathcal{L}$ assigns p the label **true** then the node m is an $\mathcal{L}$-established-type-node for r.
- If r' is a node which is assigned the same color as r under the labeling $\mathcal{L}$ then all $\mathcal{L}$-established-type-nodes for r' are also $\mathcal{L}$-established-type-nodes for r.
- If m is an $\mathcal{L}$-established-type-node for r and m' is assigned the same color as m under $\mathcal{L}$ then m' is also an $\mathcal{L}$-established-type-node for r.
- If m is an $\mathcal{L}$-established-type-node for r and $\mathcal{S}$ contains a subtype link $p \Leftrightarrow m \prec m'$ such that $\mathcal{L}$ assigns p the label **true** then m' is an $\mathcal{L}$-established-type-node for r.

Before generating a binding of the form $n \mapsto r$ the system must be sure that this binding can be *satisfied.* For example suppose that n ranges over numbers and consider the binding $n \mapsto n+1$. This binding is well typed because n ranges over numbers and $n+1$ is always a number. However there is no interpretation which assigns n the same number as $n+1$. The system ensures that a binding of the form $n \mapsto r$ can be satisfied by checking that r does not depend on n, i.e. that it is possible to set the value of n to the value of r without changing the value of r. It is now possible to define the inference relation $\rightarrow_{\mathcal{S}}$.

Definition: Let $\mathcal{T}$ be a binding labeling of $\mathcal{S}$ which consists of the truth and color labeling $\mathcal{L}$ and the binding set β. let $\mathcal{T}'$ be a binding labeling of $\mathcal{S}$ which consists of the truth and color labeling $\mathcal{L}'$ and the binding set β'.

We write $\mathcal{T} \rightarrow_{\mathcal{S}} \mathcal{T}'$ if $\mathcal{L} \rightarrow_{\mathcal{C}} \mathcal{L}'$ where $\mathcal{C}$ is the congruence constraint graph underlying $\mathcal{S}$ and $\beta = \beta'$ or if there exists a node r in $\mathcal{S}$, an $\mathcal{L}$-established-type-node m for r, a variable n of type m such that the following conditions hold:

- r does not β-depend on n.
- n is β-free (i.e. not bound under β).
- $\beta' = \beta \bigcup \{n \mapsto r\}$ and $\mathcal{L}'$ is the truth and color labeling which results from $\mathcal{L}$ by merging the equivalence classes of n and r.

The bindings generated by $\rightarrow_{\mathcal{S}}$ can not be deduced from information in the graph; the process which generates bindings is non-deductive. However it is possible to assign semantic meaning to binding labelings of semantic modulation graphs in such a way that the relation $\rightarrow_{\mathcal{S}}$ can be proven to be semantically sound.

5.2 Semantic Soundness

This section proves the semantic soundness of the inference relation $\rightarrow_{\mathcal{S}}$. The inference relation $\rightarrow_{\mathcal{S}}$ is fully specified in section 5.1 and those readers not interested in correctness proofs can safely ignore this section.

Before one can prove a soundness theorem for the relation $\rightarrow_{\mathcal{S}}$ one must define a semantics for semantic modulation graphs. A semantics for a semantic modulation graph is a set of possible worlds analogous to the possible worlds in a model of modal logic. Given this semantics it is easy to state the soundness theorem for the inference relation $\rightarrow_{\mathcal{S}}$. The proof of the $\rightarrow_{\mathcal{S}}$ soundness theorem requires the notion of a $\mathcal{W}$-valid binding labeling; the relation $\rightarrow_{\mathcal{S}}$ preserves the $\mathcal{W}$-validity of binding labelings. Unfortunately the definition of a $\mathcal{W}$-valid binding labeling is fairly complex. Furthermore the proof that $\rightarrow_{\mathcal{S}}$ preserves $\mathcal{W}$-validity is quite long and has been relegated to a separate section. This section defines the semantics of semantic modulation graphs, states the $\rightarrow_{\mathcal{S}}$ soundness theorem, and defines the notion of $\mathcal{W}$-validity which is preserved by $\rightarrow_{\mathcal{S}}$.

5.2.1 Semantics

Semantic modulation graphs have a more sophisticated semantics than any of the graphs used for purely quantifier free inference. The soundness results for Boolean constraint graphs, equality constraint graphs and congruence constraint graphs were stated in terms of a single possible world w. On the other hand the soundness result for semantic modulation graphs is stated in terms of a set $\mathcal{W}$ of possible worlds. The set $\mathcal{W}$ of possible worlds is analogous to a semantic model of a modal logic.

The graphs generated by the Ontic compiler have an intended semantics which is a special case of the general semantics defined in this section. Each node in a graph generated by the Ontic compiler is associated with an expression in the formal language Ontic. Expressions in the language Ontic have a semantics which is defined in terms of a universe of sets. More specifically, the meaning of an Ontic expression is defined relative to a universe and an interpretation of each variable as an object in that universe which is an instance of the type of the variable. Consider a fixed universe and consider all the type-respecting variable interpretations over that universe. Each type-respecting variable interpretation over a fixed universe determines a truth value for every Ontic formula and a meaning (value) for every Ontic expression. The meanings can be treated as colors and thus each type-respecting variable interpretation provides a truth and color labeling the graph generated by the Ontic compiler. Each such truth and color labeling is complete in that every formula node has a truth label. The set of truth and color labelings that correspond to the different type-respecting variable interpretations over a fixed universe determines a set $\mathcal{W}$ of possible worlds.

Definition: Let $\mathcal{S}$ be a semantic modulation graph.

A *semantics for* for $\mathcal{S}$ is a set $\mathcal{W}$ of possible worlds (complete truth and color labelings) for nodes in $\mathcal{S}$ together with a binary relation ":" on the color labels that appear in words in $\mathcal{W}$.

The *semantic domain* of a semantics $\mathcal{W}$ for $\mathcal{S}$ is the set of all color labels which appear in the worlds in $\mathcal{W}$.

If c and c' are colors in the semantic domain of a semantics $\mathcal{W}$ and if $c{:}c'$ (i.e. c is related to c' under the relation ":") then we say that c is *an instance* of the type color c'.

A color c in the semantic domain of a semantics $\mathcal{W}$ is called a *type color* if there exists a type node m and a world w in $\mathcal{W}$ such that m has color c in w. The relation ":" on colors allows a type color (or any color) to be viewed as a set. More specifically a type color c can be viewed as the set of all instances of c. Worlds assign colors to type nodes. Thus each world provides a way of interpreting each type node as a set; the set associated with type node m in world w is the set of all instances of the color of m in w. Note that the set associated with a given type node can be different in different worlds.

Definition: The color c is said to be an instance of a type node

m in a world w just in case $c\colon c_m$ where c_m is the color of m in the world w.

A type node m is said to be a *subtype* of a type node m' in world w just in case every instance of m in w is also an instance of m' in w.

Variables are nodes whose interpretation can be varied. More specifically suppose that n is a variable node with type node m. Furthermore suppose that w is a world such that c is an instance of the type of m in w. In this case it should be possible in interpret the variable n as the color c, i.e. one should be able to assign n the value c. Changing the interpretation of a variable n forces changes in the interpretation of expressions that depend on n. These intuitions are formally captured in the following semantic definition of an assignment.

Definition: Let $\mathcal{W}$ be a semantics for a semantic modulation graph $\mathcal{S}$.

We say that two worlds w and w' in $\mathcal{W}$ *agree* on a node s if w and w' assign s the same color label and if s is a formula node then w and w' assign s the same truth label.

Let n be a variable node in $\mathcal{S}$, let c be a color in the semantic domain of $\mathcal{W}$, and let w be any world in $\mathcal{W}$. An *assignment* of n to c in w is a world $w[n := c]$ which assigns n the color c and which agrees with w on all nodes that do not depend on n.

The links in a semantic modulation graph can be viewed as constraints on possible worlds. More specifically a semantics $\mathcal{W}$ is called a *satisfactory semantics* for a semantic modulation graph $\mathcal{S}$ if the information in the links in $\mathcal{S}$ holds true under the semantics $\mathcal{W}$.

Definition: We say that a semantics $\mathcal{W}$ for a semantic modulation graph $\mathcal{S}$ is a *satisfactory semantics* for $\mathcal{S}$ if the following conditions hold:

- Every world in $\mathcal{W}$ satisfies the congruence constraint graph underlying $\mathcal{S}$.
- The labels of a node are determined by the labels of the free variables of that node, i.e if w and w' are two worlds in $\mathcal{W}$ such that w and w' agree on all free variables of a node s, then w and w' agree on s (in particular if s has no free variables then all worlds in $\mathcal{W}$ must agree on s).
- If $p \Leftrightarrow\ r\!:\!m$ is a type formula link in $\mathcal{S}$ and w is a world in $\mathcal{W}$ then w assigns p the label **true** just in case the color of r in w is an instance of m in w.
- If $p \Leftrightarrow\ m \prec m'$ is a subtype link in $\mathcal{S}$ and w is a world in $\mathcal{W}$ then w assigns p the label **true** just in case m is a subtype of m' in $\mathcal{W}$.
- If n is a variable node of type m and c is an instance of m in a world w then $\mathcal{W}$ contains an assignment $w[n := c]$ of n to c in w.

It is now possible to state the main soundness theorem of this section. The proof of this theorem is long and complex and is given in the next section.

> $\rightarrow_{\mathcal{S}}$ **Soundness Theorem**: Let $\mathcal{W}$ be a satisfactory semantics for a semantic modulation graph $\mathcal{S}$. Let $\mathcal{T}$ be a binding labeling with an empty binding set such that every world in $\mathcal{W}$ satisfies the truth and color labeling of $\mathcal{T}$. Now suppose $\mathcal{T} \rightarrow_{\mathcal{S}}{}^* \mathcal{T}'$ where $\mathcal{T}'$ has binding set β and labeling $\mathcal{L}'$. If p is a formula node that is labeled **true** under $\mathcal{L}'$, and p does not depend on any variable bound under β, then p must be labeled **true** in all worlds in $\mathcal{W}$.

5.2.2 The Proof of the $\rightarrow_{\mathcal{S}}$ Soundness Theorem

The proof of the semantic modulation soundness theorem relies on the construction of a complex property, or induction hypothesis, that is preserved under the relation $\rightarrow_{\mathcal{S}}$. More specifically, given a satisfactory semantics $\mathcal{W}$ for a semantic modulation graph $\mathcal{S}$ we define the notion of a $\mathcal{W}$-valid binding labeling and prove that $\rightarrow_{\mathcal{S}}$ preserves $\mathcal{W}$-validity. A binding labeling is $\mathcal{W}$-valid if its binding set is $\mathcal{W}$-legal and the equations represented by its binding set imply the constraints in its labeling. The notion of a $\mathcal{W}$-legal binding set is quite complex. First of all every $\mathcal{W}$-legal binding set must be *universally satisfiable* in the following sense.

Definition: Let $\mathcal{W}$ be a satisfactory semantics for a semantic modulation graph $\mathcal{S}$ and let β be a binding set over $\mathcal{S}$.

A world w in $\mathcal{W}$ *satisfies* the binding β if for every binding $n \mapsto r$ in β, the world w assigns n and r the same color label.

The binding set β is $\mathcal{W}$*-universally-satisfiable* if for every world w in $\mathcal{W}$ the semantics $\mathcal{W}$ also contains a world $w[\beta]$ such that $w[\beta]$ satisfies β and agrees with w on all nodes that do not depend on any variable bound under β.

It is interesting to note that a binding set can be type respecting but still not be universally satisfiable in the above sense. For example suppose that n is a variable node that ranges over all numbers. The expression $n + 1$ always denotes a number. Thus the binding $n \mapsto n + 1$ is type respecting. However there is no world in which n equals $n + 1$ and so the binding $n \mapsto n + 1$ is not satisfiable.

If one could prove that $\rightarrow_{\mathcal{S}}$ preserves the universal satisfiability of binding sets and preserves the fact that a binding labeling's binding set implies the constraints in its labeling then one could prove the $\rightarrow_{\mathcal{S}}$ soundness theorem. Unfortunately the notion of a universally satisfiable binding set does not provide a strong enough induction hypothesis; to prove that $\rightarrow_{\mathcal{S}}$ preserves the universal satisfiability of binding sets it is necessary to prove that $\rightarrow_{\mathcal{S}}$ preserves a stronger property of binding contexts. This stronger property is called $\mathcal{W}$-legality. Before defining $\mathcal{W}$-legality however we need the notion of a β-assignment. In the presence of a binding set β we are only concerned with those worlds that satisfy β. More specifically if w is a world that satisfies β then we are interested in finding assignments $w[n := c]$ that also satisfy β.

Definition: Let β be a binding set over a semantic modulation graph $\mathcal{S}$ and let w be a world in a satisfactory semantics $\mathcal{W}$ for $\mathcal{S}$. Let n be a variable node in $\mathcal{S}$ and let c be a color in the semantic domain of $\mathcal{W}$. A *β-assignment* of n to c in w is a world $w[\beta, n := c]$ which satisfies β, assigns n the color c, and which agrees with w on all nodes that do not β-depend on n.

Of course the above definition does not guarantee that that β-assignments exist whenever c is an instance of the type of n. It turns out however that $\rightarrow_{\mathcal{S}}$ preserves the property that if n is not bound under β then β-assignments exist for n. Recall that variables which are not bound under β are called β-free.

Definition: Let β be a binding set over a semantic modulation graph $\mathcal{S}$ and let $\mathcal{W}$ be a satisfactory semantics for $\mathcal{S}$. We say that *β-assignments exist* in $\mathcal{W}$ if for every world w in $\mathcal{W}$, every β-free variable node n in $\mathcal{S}$, and every instance c of the type of n in world w under semantics $\mathcal{W}$, the semantics $\mathcal{W}$ also contains a β-assignment $w[\beta, n := c]$ of n to c in w.

There are universally satisfiable binding sets which do not have the property that β-assignments exist. However, the existence of β-assignments is one of the properties preserved under the relation $\rightarrow_{\mathcal{S}}$. The relation $\rightarrow_{\mathcal{S}}$ preserves a property called $\mathcal{W}$-legality. A binding set β is $\mathcal{W}$-legal if it is universally satisfiable, β-assignments exist, and there are not β-dependency loops as defined below.

Definition: Let $\mathcal{W}$ be a satisfactory semantics for a semantic modulation graph $\mathcal{S}$, let β be a binding set over $\mathcal{S}$.

A *β-dependency-loop* is a variable node n such that either n is bound under β with binding $n \mapsto r$ and r β-depends on n or n is β-free and the type node of n β-depends on n.

We say that the binding set β is $\mathcal{W}$*-legal* if there are no β-dependency loops, β is $\mathcal{W}$-universally-satisfiable, and β-assignments exist in $\mathcal{W}$.

The notion of a $\mathcal{W}$-legal binding set leads to the notion of a $\mathcal{W}$-valid binding labeling. A binding labeling is $\mathcal{W}$-valid if its binding set is $\mathcal{W}$-legal and its color and truth labeling is implied by its binding set, i.e. every world which satisfies its binding set also satisfies its labeling.

Definition: Let $\mathcal{W}$ be a satisfactory semantics for a semantic modulation graph $\mathcal{S}$. A binding labeling $\mathcal{T}$ is called $\mathcal{W}$*-valid* if the binding set of $\mathcal{T}$ is $\mathcal{W}$-legal and every world in $\mathcal{W}$ which satisfies the binding set of $\mathcal{T}$ also satisfies the labeling of $\mathcal{T}$.

It is now possible to state the main theorem of this section: the relation $\rightarrow_{\mathcal{S}}$ preserves $\mathcal{W}$-validity.

$\rightarrow_{\mathcal{S}}$ **Preservation Theorem**: Let $\mathcal{W}$ be a satisfactory semantics for a semantic modulation graph $\mathcal{S}$. If $\mathcal{T}$ is a $\mathcal{W}$-valid binding labeling and $\mathcal{T} \rightarrow_{\mathcal{S}} \mathcal{T}'$, then $\mathcal{T}'$ is also $\mathcal{W}$-valid.

Before giving the proof of the $\rightarrow_{\mathcal{S}}$ preservation theorem it is important to note that the $\rightarrow_{\mathcal{S}}$ preservation theorem implies the $\rightarrow_{\mathcal{S}}$ soundness theorem. More specifically consider an initial binding labeling $\mathcal{T}$, i.e. a binding labeling with an empty binding set and such that every world in the satisfactory semantics $\mathcal{W}$ satisfies the labeling of $\mathcal{T}$. It is easy to show that any such initial binding labeling is $\mathcal{W}$-valid. Now suppose $\mathcal{T} \rightarrow_{\mathcal{S}}{}^{*} \mathcal{T}'$ and consider a formula node p which is labeled **true** under the labeling of $\mathcal{T}'$ and such that p does not (directly) depend on any variable bound under the binding set of $\mathcal{T}'$. We must show that the inference relation $\rightarrow_{\mathcal{S}}$ is sound in the sense that under these conditions all worlds in $\mathcal{W}$ label p true. To prove the $\rightarrow_{\mathcal{S}}$ soundness theorem we must show that all worlds in $\mathcal{W}$ label p true. Consider any world w in $\mathcal{W}$. The $\rightarrow_{\mathcal{S}}$ preservation theorem implies that $\mathcal{T}'$ is $\mathcal{W}$-valid and thus the binding set of $\mathcal{T}'$ is $\mathcal{W}$-legal. Let β be the binding set of $\mathcal{T}'$ the binding set β is universally satisfiable and so there exists a world $w[\beta]$ that satisfies β and that agrees with w on all nodes that do not (directly) depend on variables bound under β. Since $\mathcal{T}$ is $\mathcal{W}$-valid, and since $w[\beta]$ satisfies β, $w[\beta]$ satisfies the labeling $\mathcal{L}$ which labels p **true**. Thus $w[\beta]$ labels p **true**. But since p does not depend on any variables bound under β, $w[\beta]$ must agree with w on p. Thus w must label p true. Thus the $\rightarrow_S$ preservation theorem implies the $\rightarrow_S$ soundness theorem.

5.3 Proof of the $\rightarrow_S$ Preservation Theorem

This section can safely be ignored by those readers not interested in correctness proofs.

The proof of the $\rightarrow_S$ preservation theorem is fairly long and complex. Most of the complexity of this theorem results from the definition of β-dependence. The above definition of β-dependence implies that β-dependence is non-monotonic in β; the addition of a binding $n \mapsto r$ can remove as well as add dependencies. In particular, suppose s directly depends on n, i.e. s depends on n relative to the empty binding set. Further suppose that n directly depends on n'. This this case s depends on n' in such a way that the dependency from s to n' passes through the node n. If the dependency from s to n' passes through the node n then the binding $n \mapsto r$ can "erase" this dependency; it is possible that s β-depends on n' when β is empty but s does not β-depend on n' if β consists of the single binding $n \mapsto r$. Thus the β-dependence relation is non-monotonic in β; adding bindings to β can remove dependencies.

There is a simpler, monotonic, notion of β-dependence which I will call weak-β-dependence. A node s weakly-β-depends on a variable n if either s directly depends on n or there is a binding $n' \mapsto r$ in β such that s weakly-β-depends on n' and r weakly-β-depends on n. In the current discussion I will use the term strong-β-dependence to refer to the notion of β-dependence that has been used used in the definition of $\rightarrow_S$ and the definition of a $\mathcal{W}$-legal binding set. Strong-β-dependence implies weak-β-dependence but the converse does not hold; it is possible that s weakly-β-depends on n but that s does not strongly-β-depend on n. Weak-β-dependence is monotonic in β; adding bindings monotonically increases dependencies.

If weak-β-dependence had been used rather than strong-β-dependence the relation $\rightarrow_{\mathcal{S}}$ would still preserve $\mathcal{W}$-validity and the proof of the preservation theorem would be much simpler. Unfortunately the use of weak-β-dependence would not allow as many bindings under the relation $\rightarrow_{\mathcal{S}}$. Furthermore, strong-β-dependence provides a stronger universal generalization inference mechanism. Universal generalization is discussed later.

Under strong-β-dependence the proof of the $\rightarrow_{\mathcal{S}}$ preservation theorem is long and complex. The proof is divided into four parts. The first two parts introduce two concepts needed in the proof: β-dependency-paths and minimalβ-assignments. The third part contains the proof itself. This proof relies on the first minimal assignment lemma which is stated but not proven in the section on minimal assignments. The fourth part of the proof consists of a proof of the first minimal assignment lemma.

5.3.1 β-Dependency-Paths

Before proving the $\rightarrow_{\mathcal{S}}$ preservation theorem it is useful to prove certain lemmas involving the notion of (strong) β-dependence. The following definition and lemma provide an alternative characterization of the notion β-dependence.

Definition: Let β be a binding set over a semantic modulation graph $\mathcal{S}$. A β-dependency-path is a sequence $<n_1, n_2, \ldots n_k>$ each n_i is a variable node and for each pair n_i, n_{i+1} in the path one of the following two conditions hold.

- n_i is β-free and n_{i+1} is a free variable of the type node of n_i.
- n_i is bound under β by virtue of the binding $n_i \mapsto r$ and n_{i+1} is a free variable of the node r.

If s is node in $\mathcal{S}$ such that n_1 is a free variable of s then the β-dependency-path $<n_1, n_2, \ldots n_k>$ is said to be a β-dependency-path from node s to the variable n_k.

Lemma: If β is a binding set over a semantic modulation graph $\mathcal{S}$, s is any node in $\mathcal{S}$, and n is a variable node in $\mathcal{S}$ then s β-depends on a n just in case there exists a β-dependency-path from s to n.

Lemma: There are no β-dependency-loops just in case there is no β-dependency-path of length greater than 1 that begins and ends with the same variable node.

The characterization of β-dependence in terms of β-dependency paths makes it easier to verify certain facts about β-dependency. The following lemma precisely characterizes the non-monotonic nature of β-dependency. This non-monotonicity lemma is will be important in the proof of the $\rightarrow_S$ preservation theorem.

Non-Monotonicity Lemma: Let β be a binding set over a semantic modulation graph $\mathcal{S}$. Let $n \mapsto r$ be a binding such that r does not β-depend on n and let β' be the binding set which results from adding the binding $n \mapsto r$ to β. Now let s be any node and let n' be any variable node. If s β-depends on n' but s does not β'-depend on n' then every β-dependency path form s to n' must include n and r must not β-depend on n'.

Proof: Suppose s β-depends on n' but that s does not β'-depend on n'. It is easy to show that every β-dependency path from s to n' includes n. More specifically if there existed a β-dependency-path from s to n' that does not include n then this path will also be a β'-dependency-path and thus s would β'-depend on n'. Now I will show that r does not β-depend on n'. Suppose r did β-depend on n'. In this case there exists a β-dependency-path from r to n'. The conditions of the lemma state that r does not β-depend on n and thus the β-dependency path from r to n' does not include n. Thus this path is also a β'-dependency path and so r also β'-depends on n'. Furthermore, since s β-depends on n' there must exist a β-dependency-path from s to n' and, by the above comments, any such path must include n. Consider the shortest possible β-dependency path from r to n. This path only involves n as the last node in the path and thus it is also a β'-dependency path. The β'-dependency-paths from s to n and from r to n' can be combined to yield a β'-dependency-path from s to n'. But this violates the assumption that s does not β'-depend on n'. Thus r must not β-depend on n'.

5.3.2 Minimal-β-Assignments

Intuitively one would like an assignment of the form $n := c$ to alter as few nodes as possible. For example suppose that n is a variable node that ranges over numbers and that n' is a variable node that ranges over numbers which are greater than n. Since n is a free variable of the type of n', the variable node n' depends on the variable node n. Now suppose w is a world in which n is 2 and n' is 5 and consider the assignment $n := 4$. Since n' depends on n the assignment $n := 4$ is allowed to change the value of n'. In this case however such a change is not needed; the old value of n', the number 5, is still an instance of the type of n' when n is set to the number 4. A minimal-β-assignment is a β-assignment that changes only those parameters whose values must be changed.

Definition: Let β be any binding context over a semantic modulation graph $\mathcal{S}$ and let n be any variable node in $\mathcal{S}$. A *β-supervariable* of n is defined to be any β-free variable other than n that β-depends on n.

Let β be a binding set over a semantic modulation graph $\mathcal{S}$, let w be a world in a satisfactory semantics $\mathcal{W}$ for $\mathcal{S}$, let n be a β-free variable node in $\mathcal{S}$ and let c be an instance of the type of n in world w under semantics $\mathcal{W}$. A *minimal-β-assignment* $w[\beta, n := c]$ of n to c in world w is a β-assignment $w[\beta, n := c]$

of n to c in w such that if n' is a β-supervariable of n and the color of n' under w is an instance of the type of n' in $w[\beta, n := c]$ then $w[\beta, n := c]$ agrees with w on n'.

Let β be a binding set over a semantic modulation graph $\mathcal{S}$ and let $\mathcal{W}$ be a satisfactory semantics for $\mathcal{S}$. We say that *minimal-β-assignments exist* in $\mathcal{W}$ if for every world w in $\mathcal{W}$, every β-free variable node n in $\mathcal{S}$ and every instance c of the type of n in w under semantics $\mathcal{W}$, the semantics $\mathcal{W}$ contains a minimal-β-assignment of n to c in w.

First Minimal Assignment Lemma: Let β be a binding set over a semantic modulation graph $\mathcal{S}$ and let $\mathcal{W}$ be a satisfactory semantics for $\mathcal{S}$. If β-assignments exist in $\mathcal{W}$ and there are no β-dependency loops then minimal-β-assignments exist in $\mathcal{W}$.

The first minimal assignment lemma is proved by via a conceptual procedure for constructing minimal assignments. A minimal assignment can be found by first making an arbitrary assignment and then "fixing up" the supervariables that were needlessly changed by the assignment. The full proof of the first minimal assignment lemma is fairly long and cumbersome and is relegated to its own section so that it can be easily avoided by the reader.

Second Minimal Assignment Lemma: Let β be a binding set over a semantic modulation graph $\mathcal{S}$. Let w be a world in a satisfactory semantics $\mathcal{W}$ for $\mathcal{S}$ such that w satisfies β. Let n be a variable node in $\mathcal{S}$, let c be a color in the semantic domain of $\mathcal{W}$ and let $w[\beta, n := c]$ be a member of $\mathcal{W}$ that is a minimal-β-assignment of n to c in w. If s is a node in $\mathcal{S}$ such that w and $w[\beta, n := c]$ disagree on s, and if there are no β-dependency loops then there exists a β-dependency-path from s to n such that w and $w[\beta, n := c]$ disagree on every node in that path.

Proof: If there are no β-dependency-loops then no β-dependency path is longer than the number of nodes in the graph $\mathcal{S}$. Thus there is an absolute maximum length for β-dependency-paths.

For any member s of D let the *β-path-distance* from s to n be the maximum length of any β-dependency-path from s to n.

Let D be the set of all nodes s such that w and $w[\beta, n := c]$ disagree on s. Since $w[\beta, n := c]$ is a β-assignment of n to c in w, if w and $w[\beta, n := c]$ disagree on s then s must β-depend on n. Thus if s is in D then there exists a β-dependency-path from s to n. Now consider an arbitrary member s of D. We must show that there exists a β-dependency-path from s to n such that the entire path is contained in D. It suffices to show that there exists a β-dependency-path contained entirely in D from s to some node closer to n; a path in D from s to n can be constructed from smaller paths that always get closer to n. Since $\mathcal{W}$ is a satisfactory semantics for $\mathcal{S}$ the labels of a node are determined by the color labels of the free variables of that node. Thus if s is in D, i.e. if w and $w[\beta, n := c]$ disagree on s, then there must be some free variable n' of s which is also in D. Furthermore the β-path-distance from n' to n must less than *or equal to* the β-path-distance from s to n. If n' equals n then the singleton path $<n'>$ is a β-dependency-path from s to n which is contained entirely in D. So suppose n' is not equal to n. Now there are two cases. First suppose that β contains a binding of the form $n' \mapsto r$. Since both w and $w[\beta, n := c]$ satisfy β both worlds assign the same color to n' and r and since n' is in D, r must be in D. But since r is in D some free variable n'' of r must be in D. But $<n', n''>$ is a β-dependency path contained entirely in D from s to n'' and n'' must be closer to n than s under β-path-distance. Now suppose that n' is β-free. In this case n' is a β-supervariable of n. Furthermore since n' is in D and since $w[\beta, n := c]$ is a *minimal-β*-assignment of n to c in w, the color of n' in $w[\beta, n := c]$ must not be an instance of the type of n in w. This implies that the type of n' in $w[\beta, n := c]$ is different from the type of n' in w. But since $\mathcal{W}$ is a satisfactory semantics the type of a variable is determined by the color of the type node of that variable. Thus the type node of n' must be in D. But this implies that some free variable n'' of the type node of n' is also in D. In this case $<n', n''>$ is the desired β-dependency-path in

D from s to a node which is closer to n under β-path-distance.

5.3.3 The $\rightarrow_S$ Preservation Theorem

Except for the proof of the first minimal assignment lemma, the groundwork has now been laid for the proof of the $\rightarrow_S$ preservation theorem. The theorem uses a simple lemma about $\mathcal{L}$-established-type-nodes.

Lemma: Let $\mathcal{W}$ be a satisfactory semantics for a semantic modulation graph $\mathcal{S}$, let $\mathcal{L}$ be a truth and color labeling of $\mathcal{S}$ and let w be a world in $\mathcal{W}$ such that w satisfies $\mathcal{L}$. If m is an $\mathcal{L}$-established-type-node for a node r of $\mathcal{S}$ then the color of r in the world w is an instance of m in w.

The above lemma follows directly from the definition of a $\mathcal{L}$-established-type-node and the definition of a satisfactory semantics for a semantic modulation graph; the proof is left to the reader. Given this lemma we can now prove the $\rightarrow_S$ preservation theorem.

Proof of the $\rightarrow_S$ Preservation Theorem: Suppose that $\mathcal{T}$ is $\mathcal{W}$-valid and that $\mathcal{T} \rightarrow_S \mathcal{T}'$. We must show that $\mathcal{T}'$ is $\mathcal{W}$-valid. First suppose that the binding set of $\mathcal{T}'$ is the same as the binding set of $\mathcal{T}$. In this case let β be the binding set of $\mathcal{T}$ and let $\mathcal{L}$ and $\mathcal{L}'$ be the labelings of $\mathcal{T}$ and $\mathcal{T}'$ respectively. Since the binding set of $\mathcal{T}'$ also equals β it is clear that the binding set of $\mathcal{T}'$ is $\mathcal{W}$-legal. Now let w be any world in $\mathcal{W}$ that satisfies β. To show that $\mathcal{T}'$ is $\mathcal{W}$-valid it suffices to show that w satisfies $\mathcal{L}'$. Because $\mathcal{T}$ is $\mathcal{W}$-valid, w must satisfy $\mathcal{L}$. Furthermore it follows from the definition of $\rightarrow_S$ that if the binding set of $\mathcal{T}$ equals the binding set of $\mathcal{T}'$ then $\mathcal{L} \rightarrow_C \mathcal{L}'$ where $\mathcal{C}$ is the congruence constraint graph underlying $\mathcal{S}$. But now the soundness of $\rightarrow_C$ implies w satisfies $\mathcal{L}'$.

Now suppose that the binding set of $\mathcal{T}'$ is different from the binding set of $\mathcal{T}$. Let β and β' be the binding set of $\mathcal{T}$ and $\mathcal{T}'$ respectively and let $\mathcal{L}$ and $\mathcal{L}'$ be the labelings of $\mathcal{T}$ and $\mathcal{T}'$ respectively. It follows from the definition of $\rightarrow_S$ that β' equals $\beta \bigcup \{n \mapsto r\}$ where n is a β-free variable of type m, m is an $\mathcal{L}$-established-type-node for r, and r does not β-depend on n.

First consider any world w that satisfies the binding set β'. We must show that w satisfies $\mathcal{L}'$. Since w satisfies β it must also satisfy the labeling $\mathcal{L}$. Since w satisfies the binding $n \mapsto r$ it must assign n and r the same color. Thus w must assign all nodes which are equivalent to n under $\mathcal{L}$ and all nodes which are equivalent to r under $\mathcal{L}$ the same color. The labeling $\mathcal{L}'$ is the labeling derived from $\mathcal{L}$ by merging the equivalence classes of n and r. Thus w satisfies $\mathcal{L}'$.

Next I will show that there are no β'-dependency-loops. The proof is by contradiction. Suppose there were a β'-dependency-loop. In this case there is a β'-dependency-path of length greater than 1 from a variable node to itself, i.e. a loop. This loop must involve the node n because otherwise it would be a β-dependency-loop and by assumption there are no such loops. But β' contains the binding $n \mapsto r$ and thus if there exists a β'-dependency-loop that involves n there must exist a β'-dependency path from r to n. Consider a particular β'-dependency path from r to n. The node n might occur multiple times in this path. Consider the subpath of this path that ends with the first occurance of n. This subpath is a β-dependency path. But by assumption there are no β-dependency-paths from r to n.

Now I will show that β' is $\mathcal{W}$-universally-satisfiable. Let w be any world in $\mathcal{W}$. Since β is universally satisfiable there exists a world $w[\beta]$ which satisfies β and which agrees with w on all nodes that do not depend on any variable bound under β. Because $\mathcal{T}$ is $\mathcal{W}$-valid and $w[\beta]$ satisfies β, $w[\beta]$ must also satisfy $\mathcal{L}$. Because m is an $\mathcal{L}$-established-type-node for r and $w[\beta]$ satisfies $\mathcal{L}$, the color of r in $w[\beta]$ must be an instance of m in $w[\beta]$. Let c be the color assigned to r in the world $w[\beta]$. Because β-assignments exist there exists a β-assignment $w[\beta][\beta, n := c]$ of n to c in $w[\beta]$. Since r does not β-depend on n the world $w[\beta][\beta, n := c]$ must assign r the color c. Thus, in addition to satisfying β, the world $w[\beta][\beta, n := c]$ also satisfies the binding $n \mapsto r$ and thus this world satisfies β'. It remains only to show that $w[\beta][\beta, n := c]$ agrees with w on all nodes that do not directly depend on any variable bound under β'. Let s be such a node. There does not exist any direct dependency path from s to a node bound under β'. Therefore there can not exist any β-dependency path from s to n because any such path would either be a direct path or would include a direct path to some node bound under β'. Thus s does not β-depend on n and thus $w[\beta][\beta, n := c]$ and $w[\beta]$ must agree on s. But by the definition of $w[\beta]$, $w[\beta]$ must agree with w on s.

Finally I will show that β'-assignments exist. Let w be any world in $\mathcal{W}$ that satisfies β', let n' be a β'-free variable and let c be an instance of the type of n' in the world w under the semantics $\mathcal{W}$. We must construct a β'-assignment $w[\beta', n' := c]$ of n' to c in w. Recall that β' differs from β in that β' contains the one additional binding $n \mapsto r$. The world $w[\beta', n' := c]$ is constructed in one of three different ways depending on which, if any, of the nodes n and r β-depend on n'. In all three cases the construction begins by considering a β-assignment $w[\beta, n' := c]$ of n' to c in w. Unfortunately the world $w[\beta, n' := c]$ need not satisfy the binding $n \mapsto r$. Furthermore, and more seriously, in one of the three cases β-dependence is non-monotonic; there may be a node s which β-depends on n' but does not β'-depend on n'. In this case $w[\beta, n' := c]$ may disagree with w on s even though s does not β'-depend on n'.

First consider the case where neither n nor r β-depend on n'. Since $\mathcal{W}$ is a satisfactory semantics for $\mathcal{S}$, $\mathcal{W}$ contains a β-assignment $w[\beta, n' := c]$ of n' to c in w. In this case $w[\beta, n' := c]$ is also a β'-assignment of n' to c in w. To see this first note that $w[\beta, n' := c]$ satisfies the binding $n \mapsto r$. More specifically, by assumption w satisfies $n \mapsto r$ and since neither n nor r β-depend on n', $w[\beta, n' := c]$ also satisfies $n \mapsto r$. Furthermore the non-monotonicity lemma implies that in this case every node which β-depends on n' also β'-depends on n'. Every node on which w and $w[\beta, n' := c]$ disagree must β-depend on n' and therefore every such node must β'-depend on n'.

Now suppose that r β-depends on n'. Since β is $\mathcal{W}$-legal, $\mathcal{W}$ contains a β-assignment $w[\beta, n' := c]$ of n' to c in w. Since $\mathcal{T}$ is $\mathcal{W}$-valid and since $w[\beta, n' := c]$ satisfies β, the world $w[\beta, n' := c]$ satisfies $\mathcal{L}$. However $w[\beta, n' := c]$ need not satisfy the binding $n \mapsto r$; the assignment to n' may change the value of r. In this case we satisfy the binding $n \mapsto r$ by reassigning n. More specifically let c_r be the color assigned to r in the world $w[\beta, n' := c]$. Since the type node for n is an $\mathcal{L}$-established-type-node for r, the color c_r must be an instance of the type node of n in the world $w[\beta, n' := c]$. Thus $\mathcal{W}$ contains a β-assignment $w[\beta, n' := c][\beta, n := c_r]$ of n to c_r in $w[\beta, n' := c]$. I will show that $w[\beta, n' := c][\beta, n := c_r]$ is the desired β'-assignment of n' to c in w. Since r does not β-depend on n the world $w[\beta, n' := c][\beta, n := c_r]$ assigns r the color c_r and thus this world satisfies the binding $n \mapsto r$. Furthermore one can show that n' does not β-depend on n. More specifically, in this case r β-depends on n' so if n' β-depended on n and then r would β-depend on n which is ruled out by the conditions governing the generation of bindings. Since n' does not β-depend on n the world $w[\beta, n' := c][\beta, n := c_r]$ assigns n' the color c. Finally consider some node s such that $w[\beta, n' := c][\beta, n := c_r]$ disagrees with w on s. We must show that s β'-depends on n'. Note that in this case either w and $w[\beta, n' := c]$ disagree on s or $w[\beta, n' := c]$ and $w[\beta, n' := c][\beta, n := c_r]$ must disagree on s. First note that if $w[\beta, n' := c]$ disagrees with w on s then s must β-depend on n'. The non-monotonicity lemma implies that if r β-depends on n' then every node which β-depends on n' also β'-depends on n'. Thus if $w[\beta, n' := c]$ disagrees with w on s then s β'-depends on n'. Now suppose that $w[\beta, n' := c]$ and $w[\beta, n' := c][\beta, n := c_r]$ disagree on s. In this case s must β-depend on n. Furthermore, one can show that s β'-depends on n;

since there are no β-dependency-loops a β-dependency-path from s to n involves n as a the final node and therefore any such path is also a β'-dependency path. Furthermore, since r β-depends on n' but does not β-depend on n there exists a β-dependency path from r to n' that does not involve n. The path from r to n' is also a β'-dependency path. Thus there is a β'-dependency path from s to n'.

Now consider the non-monotonic case where n β-depends on n' but r does not β-depend on n'. Since β-assignments exist in $\mathcal{W}$, minimal β-assignments also exist in $\mathcal{W}$. Thus $\mathcal{W}$ contains a minimal β-assignment $w[\beta, n' := c]$ of n' to c in w. I will show that this minimal β-assignment is the desired β'-assignment of n' to c in w. Since r does not β-depend on n' the worlds w and $w[\beta, n' := c]$ agree on r; let c_r be the color assigned to r in either world. By the argument given above c_r must be an instance of the type of n in the world $w[\beta, n' := c]$. Now by the definition of minimal-β-assignments the world $w[\beta, n' := c]$ must assign n the color c_r. Thus $w[\beta, n' := c]$ satisfies the binding $n \mapsto r$. Now consider a node s such that w and $w[\beta, n' := c]$ disagree on s. By the definition of β-assignments s must β-depend on n'. Now suppose that s does not β'-depend on n'. In this case the non-monotonicity lemma implies that every β-dependency-path from s to n' includes the node n. But the second minimal assignment lemma implies that if w and $w[\beta, n' := c]$ disagree on s then there exists a β-dependency-path from s to n' such that w and $w[\beta, n' := c]$ disagree on every node in the path. But this is impossible because every β-dependency-path from s to n' includes n and it has been shown that w and $w[\beta, n' := c]$ agree on n.

5.3.4 Proof of the First Minimal Assignment Lemma

Intuitively, minimal-β-assignments exist because there exists a conceptual procedure for constructing them. The procedure takes an arbitrary assignment and "fixes up" variables that were unnecessarily changed. Variables are fixed up using a recursive procedure for *targeted assignment.*

> **Definition:** Let β be a binding set over a semantic modulation graph $\mathcal{S}$. Let w and w' be worlds in a satisfactory semantics $\mathcal{W}$ for $\mathcal{S}$ such that both w and w' satisfy β. Let n be a β-free variable node, let c be an instance of the type of n in the world w. A *targeted-β-assignment* of n to c in w with target w' is a β-assignment $w[\beta, n := c]$ of n to c in w such that if n' is a β-supervariables of n and the color of n' under the target world w' is an instance of the type of n' in $w[\beta, n := c]$ then $w[\beta, n := c]$ agrees with the target w' on n'.

A procedure for computing targeted assignments can be used to compute minimal assignments; a minimal assignment is just a targeted assignment where the target equals the world in which the assignment is done. More specifically, to prove the first minimal assignment lemma it suffices to prove that targeted assignments exist.

> **Definition:** Let β be a binding set over a semantic modulation graph $\mathcal{S}$, let $\mathcal{W}$ be a satisfactory semantics for $\mathcal{S}$ and let n be a β-free variable node in $\mathcal{S}$.
>
> We say that *targeted-β-assignments exist for n* in $\mathcal{W}$ if for all worlds w and w' in $\mathcal{W}$ and all colors c which are instances of the type of n in w under the semantics $\mathcal{W}$, the semantics $\mathcal{W}$ contains a targeted-β-assignment of n to c in w with target w'.
>
> We say that *targeted-β-assignments* exist in $\mathcal{W}$ if for every β-free variable node n in $\mathcal{S}$ targeted-β-assignments exist for n in $\mathcal{W}$.

The conceptual procedure for computing a targeted assignment of n to c takes an arbitrary assignment of n to c and recursively "fixes" the immediate-β-supervariables of n. Recall that a β-supervariable of n is a β-free variable node n' other than n which β-depends on n. If there are on β-dependency-loops then the notion of β-dependence determines a partial order on variable nodes. If n' β-depends on n then we can picture n' as being above n. The immediate-β-supervariables of n are the least members (under β-dependence) of the β-supervariables of n.

Definition: Let β be a binding set over a semantic modulation graph $\mathcal{S}$. Let n be a β-free variable node in $\mathcal{S}$.

An *immediate-β-supervariable* of n is a β-supervariable n' of n such that there is no variable in between n' and n, i.e. there is no β-supervariable n'' of n such that n' is a β-supervariable of n''.

Observation: No two immediate-β-supervariables of n β-depend on each other, i.e. if n' and n'' are distinct immediate-β-supervariables of n then n' does not β-depend on n''.

Observation: If there are no β-dependency-loops then every β-supervariable of n is either an immediate-β-supervariable of n or is a β-supervariable of some immediate-β-supervariable of n.

The conceptual procedure for recursively computing targeted assignments always terminates because the recursive calls always involve variables of lower depth and no variable has depth less than 1. The depth of a variable is defined as follows:

Definition: Let β be binding set over a semantic modulation graph $\mathcal{S}$ such that there are no β-dependency-loops. For each variable node n let the *β-depth* of n be the length of longest β-dependency path ending at n.

Observation: If β is a binding set over $\mathcal{S}$ such that there are no β-dependency-loops and n is a β-free variable node in $\mathcal{S}$ then all β-supervariables of n have smaller β-depth than n.

The recursive conceptual procedure for computing targeted assignments can be expressed as an induction proof that targeted assignments exist. The proof is by induction on the β-depth of variable nodes.

Lemma: Let β be a be a binding set over a semantic modulation graph $\mathcal{S}$ such that there are no β-dependency-loops and let $\mathcal{W}$ be a satisfactory semantics for $\mathcal{S}$ such that β-assignments exist in $\mathcal{W}$. Under these conditions targeted β-assignments also exist in $\mathcal{W}$.

Proof: I will show by induction on the depth of variable nodes that for all variable nodes n, if n is β-free then targeted assignments exist for n in $\mathcal{W}$. Every variable node in $\mathcal{S}$ has a β-depth of at least 1 (the singleton path $<n>$ is always a dependency path). Suppose that n has depth 1. In this case there are no β-supervariables of n and thus any assignment of n to c satisfies the definition of a targeted assignment. Thus if n is β-free and has depth 1 then targeted β-assignments exist for n in $\mathcal{W}$. Now suppose that n is a variable of depth k where k is greater than 1 and targeted-β-assignments exist in $\mathcal{W}$ for all β-free variables of depth less than k. Now suppose that n is β-free and let w and w' be worlds in $\mathcal{W}$ that satisfy β. Let c be a color which is an instance of of the type of n in the world w. We must show that $\mathcal{W}$ contains a targeted-β-assignment of n to c in w with target w'. Since β-assignment exist in $\mathcal{W}$ there exists a world $w[\beta,\, n := c]$ in $\mathcal{W}$ which is a β-assignment of n to c in w. Let $n_1, n_2, \ldots n_k$ be the immediate-β-supervariables of n and let $c_1, c_2, \ldots c_k$ be the target colors for $n_1, n_2, \ldots n_k$, i.e. c_i is the color of n_i in the target world w'. Each variable n_i has smaller depth than n so by the induction hypothesis targeted-β-assignments exist in $\mathcal{W}$ for each n_i. Let $w_0, w_1, w_2, \ldots w_n$ be worlds in $\mathcal{W}$ defined as follows: w_0 equals $w[\beta,\, n := c]$. If c_i is an instance of the type of n_i in the world w_{i-1} then w_i is a targeted-β-assignment $w_{i-1}[\beta,\, n_i := c_i]$ of n_i to c_i in w_{i-1} with target w'. If c_i is not an instance of n_i in the world w_{i-1} then w_i is a targeted-β-assignment $w_{i-1}[\beta,\, n_i := b_i]$ with target w' where b_i is the color of n_i in w_{i-1} with target w'

(this targeted-β-assignment fixes the β-supervariables of n_i). I will now show that w_k is the desired targeted-β-assignment of n to c in w with target w'.

Consider an arbitrary β-supervariable n' of n and let c_t be the target color for n', i.e. the color assigned to n' by the target world w'. We must show that if the target color c_t is an instance of the type of n' in the world w_k then w_k in fact assigns n' the target color c_t. So suppose that c_t is an instance of the type of n' in the world w_k. Now there are two cases. The variable n' is either an immediate-β-supervariable of n or n' is a β-supervariable of some immediate-β-supervariable of n.

First consider the case where n' is an immediate-β-supervariable n_i of n and let m_i be the type node of n_i. The type node m_i must not β-depend on any immediate-β-supervariables of n and thus for all $0 \leq j \leq k$ the world w_j must agree with w_k on the type node m_i. In particular w_{i-1} must agree with w_k on m_i. By assumption the target color c_t is a member of the type of n_i in the world w_k and so c_t must also be a member of the type of n_i in the world w_{i-1}. Thus w_i is a target assignment $w_{i-1}[\beta, n_i := c_t]$ of n_i to its target color in w_{i-1} with target w'. Thus n_i is assigned the target color c_t in the world w_i. Furthermore n_i does not β-depend on any other immediate β-supervariables of n and thus w_k must agree with w_i on n_i and thus w_k must assign n_i the target color c_t.

Now suppose that n' is a β-supervariable of one or more of the immediate-β-supervariables n_j. Let n_i be the "last" immediate-β-supervariable such that n' β-depends on n_i, i.e. let n_i be the immediate-β-supervariable such that n' β-depends on n_i and n' does not β-depend on any immediate-β-supervariable n_j of n for $j > i$. Let m be the type node of n'. Since n' does not β-depend on any n_j for $j > i$, the type node m must not β-depend on any n_j for $j > i$. Thus the world w_i defined above must agree with w_k on the type node m. By assumption the target color c_t is an instance of the type of n' in the world w_k. Thus c_t must be an instance of the type of n' in the world w_i. But w_i is always a targeted-β-assignment of n_i with target w'. Furthermore n' β-depends on n_i. Thus, by the definition of a targeted-β-assignment and the fact that the target c_t is an instance of the type of n' in the world w_i, the world w_i must assign n' the target color c_t. But n' does not β-depend on any n_j for $j > i$ and thus the worlds w_i and w_k must agree on n'. Thus w_k assigns n' the target color c_t.

5.4 Focus, Termination, and Order Independence

This section describes a relation $\rightarrow_{S\mathcal{F}}$ which is similar to $\rightarrow_S$ except that binding construction is guided by a set of focus objects. The relation $\rightarrow_{S\mathcal{F}}$ is fully described in the beginning of this section; section 5.4.1 can be safely ignored by readers not interested in correctness proofs.

The semantic modulation inference relation $\rightarrow_S$ generates bindings of the form $n \mapsto r$. Unfortunately, in most applications there is a very large number of potential bindings. To make the semantic modulation inference process effective one must select useful bindings. In the Ontic system binding selection is guided by a set of *focus nodes*. Given a set $\mathcal{F}$ of focus nodes the Ontic system only generates bindings of the form $n \mapsto r$ where r is a member of $\mathcal{F}$.

Focus nodes represent objects that the system is thinking about. Given a set of focus objects the system uses forward chaining to generate facts about those objects. A focus object is often a variable node. For example the user might direct the system to consider an arbitrary lattice. When this is done the system chooses a variable node n whose type node represents the class of all lattices. The variable n is then added to the set of focus objects. While focusing on the arbitrary lattice n the system will generate facts that hold for all lattices. In order to ensure that the facts generated about a focus variable n hold for all instances of the type of n the system must avoid binding n to any particular object. In general the system avoids binding variables that are depended on by focus objects; binding a variable depended on by a focus object can change the meaning of the focus object.

The system also avoids redundant bindings. Suppose that n and n' are two variables that have the same type node m and suppose that m is a $\mathcal{L}$-established-type-node for r. For the graphs generated by the Ontic compiler there is no point in binding both n and n' to r; given the binding $n \mapsto r$ nothing additional will be learned from the binding $n' \mapsto r$.

In summary the Ontic system imposes three constraints on the binding process: variables are only bound to focus nodes, the system does not bind variables depended on by focus nodes, and the system does not generate redundant bindings. These three constraints lead to the following definition of the inference relation $\rightarrow_{\mathcal{S}\mathcal{F}}$ defined relative to a semantic modulation graph $\mathcal{S}$ and a set $\mathcal{F}$ of focus objects.

Definition: Let $\mathcal{F}$ be a subset of the nodes in a semantic modulation graph $\mathcal{S}$.

Definition: Let $\mathcal{T}$ be a binding labeling of a semantic modulation graph $\mathcal{S}$ such that $\mathcal{T}$ has binding set β. Let $\mathcal{T}'$ be a binding labeling of $\mathcal{S}$ with binding set β'.

We write $\mathcal{T} \rightarrow_{\mathcal{S}\mathcal{F}} \mathcal{T}'$ if $\mathcal{T} \rightarrow_{\mathcal{S}} \mathcal{T}'$ and either β' equals β or the difference between β' and β consists of a single binding $n \mapsto r$ where the following conditions hold:

- r is an element of $\mathcal{F}$.
- No member of $\mathcal{F}$ (directly) depends on n.
- β contains no binding $n' \mapsto r$ where n' has the same type node as n.

We say that a variable node n in $\mathcal{S}$ is *$\mathcal{F}$-protected* if some focus node in $\mathcal{F}$ depends on n. We say that an arbitrary node r is $\mathcal{F}$-protected if every free variable of r is $\mathcal{F}$-protected. Clearly the elements of $\mathcal{F}$ are $\mathcal{F}$-protected.

If β is a binding set generated by the relation $\rightarrow_{\mathcal{SF}}$ and if p is a node that is $\mathcal{F}$-protected then no variable depended on by p will be bound under β. One effect of this statement is that if p is $\mathcal{F}$-protected, β is a binding set generated by $\rightarrow_{\mathcal{SF}}$, and n is any variable node then p β-depends on n just in case p directly depends on n. Furthermore all members of the focus set $\mathcal{F}$ are $\mathcal{F}$-protected and thus in the second restriction on bindings in the above definition it doesn't matter whether one uses β-dependence or direct dependence — the two notions of dependence are the same when discussing the dependence of $\mathcal{F}$-protected nodes.

The relation $\rightarrow_{\mathcal{SF}}$ is simply a restriction of the relation $\rightarrow_{\mathcal{S}}$ and thus the soundness theorem holds for $\rightarrow_{\mathcal{SF}}$. Furthermore if p is $\mathcal{F}$-protected then no variable depended on by p will be bound by the inference relation $\rightarrow_{\mathcal{SF}}$. More specifically we have the following special case of the soundness theorem.

> $\rightarrow_{\mathcal{SF}}$ **Soundness Theorem**: Let $\mathcal{W}$ be a satisfactory semantics for a semantic modulation graph $\mathcal{S}$. Let $\mathcal{T}$ be a binding labeling with an empty binding set and with a truth and color labeling $\mathcal{L}$ such that every world in $\mathcal{W}$ satisfies $\mathcal{L}$. Now suppose $\mathcal{T}\rightarrow_{\mathcal{SF}}{}^{*}\mathcal{T}'$ where $\mathcal{T}'$ has binding set β and truth and color labeling $\mathcal{L}'$. If p is a formula node that is $\mathcal{F}$-protected and p is labeled **true** under $\mathcal{L}'$ then p must be labeled **true** in all worlds in $\mathcal{W}$.

5.4.1 Termination and Order Independence

This section proves a certain Church-Rosser property for relation $\rightarrow_{\mathcal{SF}}$. The relation $\rightarrow_{\mathcal{S}}$ is fully specified above and those readers not interested in correctness proofs can safely ignore this section.

The relation $\rightarrow_{\mathcal{SF}}$ operates on binding labelings of a semantic modulation graph $\mathcal{S}$. Since a given variable can only be bound once, and partial truth labelings and color labelings can not be extended indefinitely, there can be no infinite reduction chains of the form

$$\mathcal{T}_1\rightarrow_{\mathcal{SF}}\mathcal{T}_2\rightarrow_{\mathcal{SF}}\mathcal{T}_3\rightarrow_{\mathcal{SF}}\ldots$$

Thus the relation $\rightarrow_{\mathcal{S}\mathcal{F}}$ is well founded.

Let $\mathcal{S}$ be a semantic modulation graph, let $\mathcal{T}$ an initial binding labeling, let $\mathcal{F}$ be a focus set over $\mathcal{S}$, and let p be a formula node which is $\mathcal{F}$-protected, i.e. p represents some statement about the focus objects. The inference relation $\rightarrow_{\mathcal{S}\mathcal{F}}$ can be used in an attempt to prove p by binding variable nodes to focus objects. More specifically the labeling $\mathcal{T}$ can be extended via the relation $\rightarrow_{\mathcal{S}\mathcal{F}}$ until a normal form is found. Let $\mathcal{T}'$ and $\mathcal{T}''$ be two normal forms of $\mathcal{T}$ under the inference relation $\rightarrow_{\mathcal{S}\mathcal{F}}$. Now for the graphs generated by the Ontic compiler either $\mathcal{T}'$ and $\mathcal{T}''$ are both inconsistent or they both agree on p. More specifically, the compilation of individual variables (which compile into generic individual nodes) and closed formulas (such as the formulas in the lemma library) results in a homogeneous graph as described below. For homogeneous graphs it is possible to prove that the normal forms $\mathcal{T}'$ and $\mathcal{T}''$ are equivalent under a certain equivalence relation defined below. This equivalence relation has the property that if $\mathcal{T}'$ and $\mathcal{T}''$ are equivalent then either they both exhibit premature termination of they must agree on p. A binding labeling exhibits premature termination if it is inconsistent or if there is some focus object r and a $\mathcal{L}$-established-type-node m for r but there are no variables of type m that have been bound to r and no variables of type m available for binding to r. In other words a binding labeling exhibits premature termination if it runs out of variables to bind to focus nodes. Because the Ontic compiler generates variables on demand, a binding labeling does not exhibit premature termination in practice unless it is inconsistent. Thus if $\mathcal{T}'$ and $\mathcal{T}''$ are both normals forms of $\mathcal{T}$ under the relation $\rightarrow_{\mathcal{S}\mathcal{F}}$, and if p is $\mathcal{F}$-protected, they either $\mathcal{T}'$ and $\mathcal{T}''$ are both inconsistent or they agree on p.

Definition: Let $\mathcal{T}$ be a binding labeling of a semantic modulation graph $\mathcal{S}$. We say that $\mathcal{T}$ is $\mathcal{S}$-inconsistent if the labeling of $\mathcal{T}$ is $\mathcal{C}$-inconsistent where $\mathcal{C}$ is the congruence constraint graph underlying $\mathcal{S}$.

Let $\mathcal{F}$ be a subset of the nodes of a semantic modulation graph $\mathcal{S}$ and let $\mathcal{T}$ be a binding labeling of $\mathcal{S}$with truth and color labeling $\mathcal{L}$. We say that $\mathcal{T}$ *exhibits premature $\mathcal{F}$-termination* if either $\mathcal{T}$

is $\mathcal{S}$-inconsistent or there exists a focus object r in $\mathcal{F}$ and a $\mathcal{L}$-established-type-node m for r such that there is no binding of the form $n \mapsto r$ in the binding set of $\mathcal{T}$ where n is a variable of type m and every variable of type m is either $\mathcal{F}$-protected or is already bound under the binding set of $\mathcal{T}$.

The equivalence relations defined in previous sections had the property that any two inconsistent labelings were equivalent. The equivalence relation defined below has the property that any two binding labelings which exhibit premature termination are equivalent. In practice the Ontic system generates variables on demand so that there are always enough variables in the graph to avoid premature termination due a lack of variables. Thus, in practice, premature termination always involves an inconsistency. If $\mathcal{T}$ is a normalized binding labeling with truth and color labeling $\mathcal{L}$ such that $\mathcal{T}$ does not exhibit premature termination and if r is a focus object and m is a $\mathcal{L}$-established-type-node for r then some variable of type m is bound to r under the binding set of $\mathcal{T}$.

The graphs generated by the Ontic compiler are *homogeneous* in the sense that if n and n' are two variables with the same type node then n and n' are "identical" as nodes in the graph. More specifically if n and n' are both variables with the same type node then there exists a symmetry of the graph which carries n to n'. A symmetry is a particular way that an object is identical to itself. For example a square is identical to itself when rotated ninety degrees. The formal definition of symmetry is based on the general notion of isomorphism. Two semantic modulation graphs are isomorphic if there is a bijection between there nodes which carries the structure of one onto the structure of the other. A symmetry is an isomorphism of an object with itself, e.g. a rotation of a square is particular way that the square is isomorphic to itself.

To precisely define the notion of isomorphism one needs to define how a map *carries* the structure of a graph. More specifically consider a bijection ι which maps the nodes of a semantic modulation graph $\mathcal{S}$ to some other set of nodes $\mathcal{N}$. The map ι carries the graph $\mathcal{S}$ to the graph $\iota(\mathcal{S})$ such that the nodes of $\iota(\mathcal{S})$ consist of the elements of $\mathcal{N}$ and the classification of nodes and the links of $\iota(\mathcal{S})$ are defined as follows:

Definition: Let $\mathcal{S}$ be a semantic modulation graph and let ι be a bijection mapping the nodes in $\mathcal{S}$ to some set. The map ι carries the graph $\mathcal{S}$ to the graph $\iota(\mathcal{S})$ where the graph $\iota(\mathcal{S})$ is defined as follows:

- The formula nodes of $\iota(\mathcal{S})$ are the objects of the form $\iota(n)$ where n is a formula node of $\mathcal{S}$. The quotation nodes, type nodes, variable nodes and unclassified nodes of $\iota(\mathcal{S})$ are defined similarly.
- If Ψ is a literal over the formula nodes in $\mathcal{S}$ then $\iota(\Psi)$ is defined so that if Ψ is the node n then $\iota(\Psi)$ equals $\iota(n)$ and if Ψ is the literal $\neg n$ then $\iota(\Psi)$ equals $\neg\iota(n)$. The clause links of $\iota(\mathcal{S})$ consist of all clause links of the form

$$\iota(\Psi_1) \vee \iota(\Psi_2) \ldots \vee \iota(\Psi_k)$$

  where $\mathcal{S}$ contains the clause link

$$\Psi_1 \vee \Psi_2 \ldots \vee \Psi_k$$

- The equality links of $\iota(\mathcal{S})$ consist of all links of the form

$$\iota(p) \Leftrightarrow \ \iota(n) = \iota(m)$$

  where $\mathcal{S}$ contains the link

$$p \Leftrightarrow \ n = m$$

- The subexpression links, free variable links, type declaration links, type formula links, and subtype links in $\iota(\mathcal{S})$ are defined similarly.

Now consider a bijection ι that maps the nodes of a graph $\mathcal{S}$ to any set. As discussed above the bijection ι carries the structure of the graph $\mathcal{S}$ over to the structure of a new graph $\iota(\mathcal{S})$. The bijection $\mathcal{S}$ also carries binding labelings of $\mathcal{S}$ over to binding labelings of the graph $\iota(\mathcal{S})$.

Definition: Let ι be a bijection from the nodes of a semantic modulation graph $\mathcal{S}$ to some set.

Let $\mathcal{L}$ be a truth an color labeling of $\mathcal{S}$. The labeling $\iota(\mathcal{L})$ is the truth and color labeling of $\iota(\mathcal{S})$ such that if $\mathcal{L}$ labels p **true** then $\iota(\mathcal{L})$ labels $\iota(p)$ **true** and if $\mathcal{L}$ assigns node r the color c then $\iota(\mathcal{L})$ assigns $\iota(r)$ the color c.

Let β be a binding set over $\mathcal{L}$. The bijection ι carries β to the binding set $\iota(\beta)$ over the graph $\iota(\mathcal{S})$ where $\iota(\beta)$ consists of all bindings of the form $\iota(n) \mapsto \iota(r)$ where $n \mapsto r$ is a binding in β.

Let $\mathcal{T}$ be a binding labeling of $\mathcal{S}$ with binding set β and truth and color labeling $\mathcal{L}$. The mapping ι carries $\mathcal{T}$ to the binding labeling $\iota(\mathcal{T})$ with binding set $\iota(\beta)$ and truth and color labeling $\iota(\mathcal{L})$.

For any bijection ι from the nodes of a semantic modulation graph $\mathcal{S}$to some set, the graph $\iota(\mathcal{S})$ is in some sense identical to the graph $\mathcal{S}$ even though the nodes of $\iota(\mathcal{S})$ may be different from the nodes of $\mathcal{S}$. This observation leads to the notion of isomorphism.

Definition: Two semantic modulation graphs $\mathcal{S}$ and $\mathcal{S}'$ are *isomorphic* just in case $\mathcal{S}'$ can be written as $\iota(\mathcal{S})$ for some bijection ι between the nodes of $\mathcal{S}$ and the nodes of $\mathcal{S}'$. A map ι which carries $\mathcal{S}$ to $\mathcal{S}'$ is called an *isomorphism* between $\mathcal{S}$ and $\mathcal{S}'$.

The notion of isomorphism leads to a notion of symmetry.

Definition: A symmetry of a semantic constraint graph $\mathcal{S}$ is an isomorphism of $\mathcal{S}$ with itself, i.e. a bijection ι from the nodes of $\mathcal{S}$ to themselves such that $\iota(\mathcal{S})$ equals $\mathcal{S}$.

As mentioned above the graphs generated by compiling individual variables and closed formulas are highly symmetrical. More specifically, such graphs are *homogeneous* in the following sense.

> **Definition**: Two variables n and n' in a semantic modulation graph $\mathcal{S}$ will be called *$\mathcal{S}$-identical* if there exists a symmetry ι of $\mathcal{S}$ which exchanges n and n' and which is the identity map for all nodes r which do not depend on either n or n'.
>
> A semantic modulation graph $\mathcal{S}$ is called *homogeneous* if any two variables with the same type node are $\mathcal{S}$-identical.

If variables of the same type are identical then it shouldn't matter which variable is bound to a given focus object; two labelings should be considered to be equivalent if the only difference between them is that they bind different but identical variables to the same focus object. More specifically let $\mathcal{F}$ be a focus set over a semantic modulation graph $\mathcal{S}$ and let ι be a symmetry of $\mathcal{S}$ that is the identity function on all $\mathcal{F}$-protected nodes. The symmetry ι exchanges identical variables but preserves all $\mathcal{F}$-protected nodes. If $\mathcal{T}$ is a binding labeling of $\mathcal{S}$ then the binding labeling $\iota(\mathcal{T})$ should be equivalent to $\mathcal{T}$.

> **Definition**: Let $\mathcal{F}$ be focus set over a semantic modulation graph $\mathcal{S}$.
>
> A symmetry ι of $\mathcal{S}$ is called *$\mathcal{F}$-preserving* if ι is the identity function on all $\mathcal{F}$-protected nodes in $\mathcal{S}$.
>
> Two binding labelings $\mathcal{T}$ and $\mathcal{T}'$ of $\mathcal{S}$ are called *immediately-$\mathcal{S}$-equivalent* if they have the same binding set, they assign the same truth values to formula nodes, and their color labelings define the same equivalence relation on nodes.
>
> Two binding labelings $\mathcal{T}$ and $\mathcal{T}'$ of $\mathcal{S}$ are called *$\mathcal{SF}$-equivalent* if either both $\mathcal{T}$ and $\mathcal{T}'$ exhibit premature termination or there exists a $\mathcal{F}$-preserving symmetry ι of $\mathcal{S}$ such that $\iota(\mathcal{T})$ is immediately-$\mathcal{S}$-equivalent to $\mathcal{T}'$.

It is possible to prove that $\rightarrow_{\mathcal{SF}}$ satisfies the diamond property modulo $\mathcal{SF}$-equivalence and thus $\rightarrow_{\mathcal{SF}}$ is order independent.

$\rightarrow_{\mathcal{SF}}$ **Normalization Theorem**: If $\mathcal{S}$ is a homogeneous semantic modulation graph and $\mathcal{F}$ is a focus set over $\mathcal{S}$ then the relation $\rightarrow_{\mathcal{SF}}$ is a terminating normalizer modulo $\mathcal{SF}$-equivalence.

The above order independence result implies that in certain easily identified cases the answers generated by the the Ontic system do not depend on the order in which inference operations are performed.

Corollary: Let $\mathcal{F}$ be focus set over a homogeneous semantic modulation graph $\mathcal{S}$ let p be a $\mathcal{F}$-protected formula node, and let $\mathcal{T}$ be a binding labeling of $\mathcal{S}$. If $\mathcal{T}'$ and $\mathcal{T}''$ are both normalizations of $\mathcal{T}$ under $\rightarrow_{\mathcal{SF}}$ then either both $\mathcal{T}'$ and $\mathcal{T}''$ exhibit premature termination or $\mathcal{T}'$ and $\mathcal{T}''$ agree on the truth of p.

5.5 Assumptions

This section describes an inference relation $\rightarrow_{\mathcal{SA}}$ which performs inference in the presence of assumptions (suppositions). The inference relation $\rightarrow_{\mathcal{SA}}$ is fully described in the beginning of the section. The relation $\rightarrow_{\mathcal{SFA}}$, that incorporates focus, is described in section 5.5.2. Sections 5.5.1 and 5.5.3 involve soundness and unique normalization respectively and can be safely ignored by readers not interested in correctness proofs.

Recall that a binding labeling $\mathcal{T}$ for $\mathcal{S}$ is $\mathcal{W}$-valid if the binding set of $\mathcal{T}$ is $\mathcal{W}$-legal and the binding set of $\mathcal{T}$ implies the truth and color labeling of $\mathcal{T}$, i.e. every world in $\mathcal{W}$ that satisfies the binding set of $\mathcal{T}$ also satisfies the truth and color labeling of $\mathcal{T}$. If $\mathcal{W}$ is a satisfactory semantics for the graph $\mathcal{S}$ then the relation $\rightarrow_{\mathcal{S}}$ preserves $\mathcal{W}$-validity. Unfortunately the notion of $\mathcal{W}$-validity does not allow for *assumptions*. An assumption is a statement that is true in some worlds but not others. To properly handle assumptions one must deal with labelings that are not $\mathcal{W}$-valid.

Definition: Let $\mathcal{S}$ be a semantic modulation graph and let $\mathcal{W}$

be a satisfactory semantics for $\mathcal{S}$.

An *assumption set* over $\mathcal{S}$ is a subset $\mathcal{A}$ of the formula nodes in $\mathcal{S}$. If w is a world in $\mathcal{W}$ then we say that w *satisfies* $\mathcal{A}$ if w assigns every formula node in $\mathcal{A}$ the label **true**.

Assumptions can be handled by an inference relation $\rightarrow_{\mathcal{SA}}$ where $\mathcal{A}$ is an assumption set over $\mathcal{S}$. A later section will discuss how assumptions can be combined with focus objects to yield an inference relation $\rightarrow_{\mathcal{SFA}}$ which is a controlled restriction of the relation $\rightarrow_{\mathcal{SA}}$ defined here. However, focus objects are ignored in the remainder of this section.

The labelings manipulated by the relation $\rightarrow_{\mathcal{SA}}$ contain information that is deduced from the assumption set $\mathcal{A}$. The assumptions in $\mathcal{A}$ may contain assumptions about the types of objects. Thus a certain binding may be type respecting relative under the assumptions in $\mathcal{A}$ even if that binding can not be proven to be type respecting in general. Furthermore the assumptions in $\mathcal{A}$ place restrictions on the free variables of the assumptions; it may not be possible to assign values to the free variables of assumption without making the assumptions false. Thus the relation $\rightarrow_{\mathcal{SA}}$ avoids binding variables which are depended on by elements of the assumption set $\mathcal{A}$. In fact the only difference between the relations $\rightarrow_{\mathcal{S}}$ and $\rightarrow_{\mathcal{SA}}$ is that $\rightarrow_{\mathcal{SA}}$ avoids binding variables depended on by the assumptions in $\mathcal{A}$.

Definition: Let $\mathcal{A}$ be an assumption set over a semantic modulation graph $\mathcal{S}$.

If β is a binding set over $\mathcal{S}$ then a variable node n in $\mathcal{S}$ is called $\mathcal{A}\beta$-free if n is β-free, i.e. not bound under β, and no assumption in $\mathcal{A}$ β-depends on n.

Let $\mathcal{T}$ and $\mathcal{T}'$ be two binding labelings of $\mathcal{S}$. We write $\mathcal{T} \rightarrow_{\mathcal{SA}} \mathcal{T}'$ if $\mathcal{T} \rightarrow_{\mathcal{S}} \mathcal{T}'$ and either $\mathcal{T}$ and $\mathcal{T}'$ have the same binding set or the binding sets of $\mathcal{T}'$ contains an additional binding $n \mapsto r$ where n is $\mathcal{A}\beta$-free.

The restriction on bindings given in the above definition makes it possible to prove a soundness theorem for the relation $\rightarrow_{\mathcal{SA}}$; this theorem establishes that $\rightarrow_{\mathcal{SA}}$ can be used to find logical consequences of a set of assumptions.

> $\rightarrow_{\mathcal{SA}}$ **Soundness Theorem:** Let $\mathcal{W}$ be a satisfactory semantics for a semantic modulation graph $\mathcal{S}$ and let $\mathcal{A}$ be an assumption set over $\mathcal{S}$. Let $\mathcal{T}$ be a binding labeling with an empty binding set and such that every world in $\mathcal{W}$ that satisfies $\mathcal{A}$ also satisfies the truth and color labeling of $\mathcal{T}$. Now suppose $\mathcal{T} \rightarrow_{\mathcal{SA}}{}^{*} \mathcal{T}'$ where $\mathcal{T}'$ has binding set β. If p is a formula node such that p is labeled **true** under $\mathcal{T}'$ and no variable depended on by p is bound under β then p must be labeled **true** in all worlds in $\mathcal{W}$ that satisfy $\mathcal{A}$.

Intuitively, the assumption soundness theorem holds because assumptions do not constrain variables not depended on by the assumptions; variables not depended on by assumptions are still free to range over their types and such a variable can be assigned to any object that is known to be an instance of its type. These intuitive comments are made more precise below.

5.5.1 Proof of the $\rightarrow_{\mathcal{SA}}$ Soundness Theorem

Like the semantic modulation soundness theorem, the assumption soundness theorem is proven by showing that the relation $\rightarrow_{\mathcal{SA}}$ preserves a certain property of binding labelings. More specifically the relation $\rightarrow_{\mathcal{SA}}$ preserves $\mathcal{AW}$-validity where a binding labeling is $\mathcal{AW}$-valid just in case its binding set is $\mathcal{AW}$-legal and its bindings together with the assumptions in $\mathcal{A}$ imply its truth and color labeling. The notion of an $\mathcal{AW}$-legal binding context is similar to the notion of a $\mathcal{W}$-legal binding context except that the concepts involved are relativized in some way to the assumption set $\mathcal{A}$.

An $\mathcal{AW}$-legal binding set need not be $\mathcal{W}$-legal; the legality of bindings in an $\mathcal{AW}$-legal binding set may depend on assumptions in $\mathcal{A}$. More specifically, an $\mathcal{AW}$-legal binding set need not be $\mathcal{W}$-universally-satisfiable; if β is $\mathcal{AW}$-legal, and w is a world in $\mathcal{W}$ such that w does not satisfy $\mathcal{A}$, then $\mathcal{W}$ need not contain a world $w[\beta]$ that satisfies β and agrees with w on all nodes that do not depend on variables bound under β. In defining the $\mathcal{AW}$-legal binding sets the notion of $\mathcal{W}$-universal-satisfiability is replaced by the notion of $\mathcal{AW}$-universal-satisfiability.

Definition: Let $\mathcal{W}$ be a satisfactory semantics for a semantic modulation graph $\mathcal{S}$, let $\mathcal{A}$ be an assumption set over $\mathcal{S}$, and let β be a binding set over $\mathcal{S}$. The binding set β is $\mathcal{AW}$*-universally-satisfiable* if for every world w in $\mathcal{W}$ em such that w satisfies $\mathcal{A}$ the semantics $\mathcal{W}$ contains a world $w[\beta]$ such that $w[\beta]$ satisfies β and agrees with w on all nodes that do not depend on any variable bound under β.

The following lemma states that if β is $\mathcal{A}$-protecting in the sense defined below then β-assignments to $\mathcal{A}\beta$-free variables always preserve the truth of the assumptions in $\mathcal{A}$. Recall that a variable n is $\mathcal{A}\beta$-free just in case n is β-free and no assumption in $\mathcal{A}$ β-depends on n.

Definition: Let $\mathcal{A}$ be an assumption set over a semantic modulation graph $\mathcal{S}$, let $\mathcal{W}$ be a satisfactory semantics for $\mathcal{S}$, and let β be a binding set over $\mathcal{S}$.

The binding set β is called $\mathcal{A}$-protecting if no variable depended on by an element of $\mathcal{A}$ is bound under β.

Lemma: If β is $\mathcal{A}$-protecting, w is a world in $\mathcal{W}$ that satisfies $\mathcal{A}$, n is an $\mathcal{A}\beta$-free variable node, and c is an instance of the type of n in a world w then any β-assignments of n to c in w also satisfies $\mathcal{A}$.

Proof: Since n is $\mathcal{A}\beta$-free no assumption in $\mathcal{A}$ (directly) depends on n. Furthermore, I will show that no assumption in $\mathcal{A}$ β-depends on n. More specifically, suppose that there existed a β-dependency-path from and assumption p in $\mathcal{A}$ to the variable n. Since p does not directly depend on n this path must involve some variable bound under β. Thus there must be a direct dependency path from p to some variable bound under β. But this is impossible because β is assumed to be $\mathcal{A}$-protecting. Thus no assumption in $\mathcal{A}$ β-depends on n. Thus if $w[\beta, n := c]$ is a β-assignment of n to c in w then w and $w[\beta, n := c]$ must agree on all elements of $\mathcal{A}$. By assumption w satisfies $\mathcal{A}$ so $w[\beta, n := c]$ also satisfies $\mathcal{A}$.

An $\mathcal{AW}$-legal binding set β need not have the property that β-assignments exist in $\mathcal{W}$. More specifically the existence of β-assignments may depend on the assumptions in $\mathcal{A}$ and thus if w is a world that does not satisfy $\mathcal{A}$ there may be a variable node n and an instance c of the type of n such that $\mathcal{W}$ does not contain a β-assignment of n to c in w. When dealing with assumptions the requirement that β-assignments exist must be restricted to those worlds which satisfy the assumption set.

Definition: We say that *β-assignments exist* in $\mathcal{W}$ under $\mathcal{A}$ if for every world w in $\mathcal{W}$ that satisfies both β and $\mathcal{A}$, every $\mathcal{A}\beta$-free variable node n in $\mathcal{S}$, and every instance c of the type of n in world w, the semantics $\mathcal{W}$ contains a β-assignment of n to c in w.

It is now possible to define the $\mathcal{AW}$-legal binding sets.

Definition: Let $\mathcal{W}$ be a satisfactory semantics for a semantic modulation graph $\mathcal{S}$, let $\mathcal{A}$ be an assumption set over $\mathcal{S}$, and let β be a binding set over $\mathcal{S}$. We say that the binding set β is *$\mathcal{AW}$-legal* if there are no β-dependency loops, β is $\mathcal{AW}$-universally-satisfiable, β is $\mathcal{A}$-protecting, and β-assignments exist in $\mathcal{W}$ under $\mathcal{A}$.

If β is the empty binding set then there are no β-dependency-loops; β is clearly $\mathcal{AW}$-universally-satisfiable; and β is $\mathcal{A}$-protecting. Furthermore if

β is empty then β-assignments exist in all worlds in $\mathcal{W}$. Thus the empty binding set is $\mathcal{AW}$-legal.

The notion of an $\mathcal{AW}$-legal binding context leads to the notion of an $\mathcal{AW}$-valid binding labeling. A binding labeling $\mathcal{T}$ is $\mathcal{AW}$-valid if its binding set is $\mathcal{AW}$-legal and its truth and color labeling is implied by its binding set and the assumptions in $\mathcal{A}$.

Definition: Let $\mathcal{W}$ be a satisfactory semantics for a semantic modulation graph $\mathcal{S}$ and let $\mathcal{A}$ be an assumption set over $\mathcal{S}$. A binding labeling $\mathcal{T}$ is called $\mathcal{AW}$*-valid* if the binding set of $\mathcal{T}$ is $\mathcal{AW}$-legal and every world in $\mathcal{W}$ which satisfies both $\mathcal{A}$ and the binding set of $\mathcal{T}$ also satisfies the truth and color labeling of $\mathcal{T}$.

It is now possible to state the main theorem of this section: the relation $\rightarrow_{\mathcal{SA}}$ preserves $\mathcal{AW}$-validity.

$\rightarrow_{\mathcal{SA}}$ **Preservation Theorem**: Let $\mathcal{W}$ be a satisfactory semantics for a semantic modulation graph $\mathcal{S}$ and let $\mathcal{A}$ be an assumption set for $\mathcal{S}$. If $\mathcal{T}$ is an $\mathcal{AW}$-valid binding labeling and $\mathcal{T} \rightarrow_{\mathcal{SA}} \mathcal{T}'$, then $\mathcal{T}'$ is also $\mathcal{AW}$-valid.

The proof of the $\rightarrow_{\mathcal{SA}}$ preservation theorem is essentially the same as the proof of the $\rightarrow_{\mathcal{S}}$ preservation theorem given earlier; the proof will not be given here. It is important to note however that the restriction on bindings stated in the definition of $\rightarrow_{\mathcal{SA}}$ is essential for the $\rightarrow_{\mathcal{SA}}$ preservation theorem. More specifically suppose β contained a binding of the form $n \mapsto r$ where some assumption in $\mathcal{A}$ depends on n. In this case the binding $n \mapsto r$ may violate the assumptions in $\mathcal{A}$; the binding may not be satisfiable by any world that satisfies $\mathcal{A}$.

5.5.2 Combining Assumptions and Focus Objects

Focus objects guide the choice of bindings generated in the Ontic system. It is easy to combine focus and assumptions. More specifically the relation $\rightarrow_{\mathcal{SFA}}$ can be defined as follows:

> **Definition:** If $\mathcal{T}$ and $\mathcal{T}'$ are two binding labelings of a semantic modulation graph $\mathcal{S}$ then we write $\mathcal{T} \rightarrow_{\mathcal{SFA}} \mathcal{T}'$ if $\mathcal{T} \rightarrow_{\mathcal{SA}} \mathcal{T}'$ and $\mathcal{T} \rightarrow_{\mathcal{SF}} \mathcal{T}'$.

The above definition implies that the relation $\rightarrow_{\mathcal{SFA}}$ is a restriction of the relation $\rightarrow_{\mathcal{SA}}$. More specifically $\rightarrow_{\mathcal{SFA}}$ is that restriction of $\rightarrow_{\mathcal{SA}}$ which only generates bindings $n \mapsto r$ where r is a member of the focus set $\mathcal{F}$, no other variable with the same type node as n has already been bound to r, and no member of the focus set depends on n. Since $\rightarrow_{\mathcal{SFA}}$ is a restriction of $\rightarrow_{\mathcal{SA}}$ it preserves $\mathcal{AW}$-validity.

5.5.3 Termination and Order Independence

Since each variable can be bound at most once, and since truth and color labelings can not be extended indefinitely, all of the inference relations discussed so far are well founded; there are no infinite inference chains.

Furthermore it can be shown that the ability of the relation $\rightarrow_{\mathcal{SFA}}$ to prove a given result does not depend on the order in which inferences are performed. More specifically, let $\mathcal{S}$ be a semantic modulation graph; let $\mathcal{F}$ be a focus set over $\mathcal{S}$, and let p be a formula node which is $\mathcal{F}$-protected, i.e. p represents some statement about the focus objects; and let $\mathcal{A}$ be an assumption set over $\mathcal{S}$. The relation $\rightarrow_{\mathcal{SFA}}$ can be used in an attempt to prove that p follows from the assumptions in $\mathcal{A}$. More specifically let $\mathcal{T}$ an initial binding labeling such that the labeling of $\mathcal{T}$ satisfies $\mathcal{A}$ and let $\mathcal{T}'$ and $\mathcal{T}''$ be two normal forms of $\mathcal{T}$ under the inference relation $\rightarrow_{\mathcal{SFA}}$. It turns out that the relation $\rightarrow_{\mathcal{SFA}}$ is order independent in the sense that, for the graphs generated by compiling individual variables and closed formulas, either $\mathcal{T}'$ and $\mathcal{T}''$ are both inconsistent or they both agree on p.

The proof of the order independence result for the relation $\rightarrow_{\mathcal{SFA}}$ is very similar to the proof of the order independence result for $\rightarrow_{\mathcal{SF}}$. In fact the only difference between these two proofs involves the notion of premature termination. It is possible that a binding labeling $\mathcal{T}'$ is normalized under $\rightarrow_{\mathcal{SFA}}$ even though it could be reduced further under $\rightarrow_{\mathcal{SF}}$. More specifically, a variable might be β-free and thus available for binding under $\rightarrow_{\mathcal{SF}}$ but not $\mathcal{A}\beta$-free and thus not available for binding under $\rightarrow_{\mathcal{SFA}}$. In fact it is possible that $\mathcal{T}'$ exhibits premature termination with respect to the relation $\rightarrow_{\mathcal{SFA}}$ even though it does not exhibit premature termination with respect to the relation $\rightarrow_{\mathcal{SF}}$. A binding labeling $\mathcal{T}$ exhibits premature $\mathcal{AF}$-termination just in case the truth and color labeling of $\mathcal{T}$ is inconsistent or there are not enough variables of the appropriate types available for binding to the focus objects (the precise definition should be clear and is not given here).

The $\rightarrow_{\mathcal{SFA}}$ normalization theorem is stated in terms of a certain equivalence relation on labelings. The notion of $\mathcal{AFS}$-equivalence can be defined as follows:

Definition: Let $\mathcal{F}$ be a focus set over a semantic modulation graph $\mathcal{S}$ and let $\mathcal{A}$ be an assumption set over $\mathcal{S}$.

A node r is called *$\mathcal{AF}$-protected* if every variable depended on by r is also depended on by some element of $\mathcal{F}$ or $\mathcal{A}$. (If r is $\mathcal{AF}$-protected then no binding generated by $\rightarrow_{\mathcal{SFA}}$ binds a variable depended on by r.)

A symmetry ι of $\mathcal{S}$ is called *$\mathcal{AF}$-preserving* if ι is the identity function on all $\mathcal{AF}$-protected nodes.

Two binding labelings $\mathcal{T}$ and $\mathcal{T}'$ of $\mathcal{S}$ are called *$\mathcal{ASF}$-equivalent* if either both $\mathcal{T}$ and $\mathcal{T}'$ exhibit premature $\mathcal{AF}$-termination or there exists an $\mathcal{AF}$-preserving symmetry ι of $\mathcal{S}$ such that $\iota(\mathcal{T})$ is immediately-$\mathcal{S}$-equivalent to $\mathcal{T}'$.

Now it is possible to prove that if $\mathcal{S}$ is homogeneous then $\rightarrow_{\mathcal{SFA}}$ satisfies the diamond property modulo $\mathcal{AFS}$-equivalence. Thus $\rightarrow_{\mathcal{SFA}}$ is a terminating normalizer relative to $\mathcal{AFS}$-equivalence. Furthermore if $\mathcal{T}$ and $\mathcal{T}'$ are $\mathcal{AFS}$-equivalent and p is an $\mathcal{AF}$-protected formula node then either $\mathcal{T}$ and $\mathcal{T}'$ both exhibit premature termination or they both agree on the truth of p. Thus the ability of the system to determine the truth of an $\mathcal{AF}$-protected formula does not depend on the order in which reductions are done.

5.6 Automatic Universal Generalization

This section describes an inference relation $\rightarrow_{\mathcal{G}}$ which performs automatic universal generalization. The inference relation $\rightarrow_{\mathcal{G}}$ is fully described in the beginning of the section and sections 5.6.1 can safely be ignored by readers not interested in correctness proofs. Section 5.6.2 describes the relation $\rightarrow_{\mathcal{GA}}$ which is similar to $\rightarrow_{\mathcal{G}}$ except that it handles a set of assumptions (suppositions). Section 5.6.3 discusses semantic soundness and can be safely ignored by readers not interested in correctness proofs. The relations $\rightarrow_{\mathcal{G}}$ and $\rightarrow_{\mathcal{GA}}$ are not guided by focus objects; section 5.6.4 describes a relation that is guided by focus objects.

Universal generalization is a method for deducing formulas of the form

```
(FORALL ((X τ)) Φ)
```

More specifically, suppose that a variable X of type τ appears free in the formula Φ and that Φ has been proven using only the fact that X is an instance of the type τ. In this case Φ must be true no matter how one interprets X as an instance of τ and thus one can infer that the above universal formula is true.

In the Ontic system the formula

```
(FORALL ((X τ)) Φ)
```

abbreviates the formula

```
(NOT
  (EXISTS-SOME
    (LAMBDA ((X τ))
      (NOT Φ))))
```

`LAMBDA` is the only true quantifier in the Ontic system; classical quantification is handled with the quantifier `LAMBDA` and formulas of the form

(EXISTS-SOME σ)

where σ is a type expression. In order to implement universal generalization as a graph labeling inference mechanism two additional kinds of links are needed corresponding to the quantifier `LAMBDA` and the operator `EXISTS-SOME`.

Definition: An *Ontic graph* $\mathcal{G}$ consists of a semantic modulation graph together with

- a set of *existential links* of the form

  $$p \Leftrightarrow \exists m$$

  where p is a formula node and m is a type node. Such a link says that p represents the formula which says that there exist instances of the type m.

- a set of *closure links* of the form

  $$\lambda n.p = m$$

  where n is a variable node, p is a formula node such that no free variable of p other than n depends on n, and m is a type node. Such a link says that m represents the type whose instances are the values of the variable n which satisfy the formula represented by p.

If $\mathcal{S}$ is the semantic modulation graph derived by deleting all existential links and closure links from an Ontic graph $\mathcal{G}$ then $\mathcal{S}$ is called the semantic modulation graph underlying $\mathcal{G}$.

Let $\mathcal{G}$ be an Ontic graph and let $\mathcal{S}$ be the underlying semantic modulation graph. A labeling of $\mathcal{G}$ is simply a labeling of $\mathcal{S}$; a binding set over $\mathcal{G}$ is a binding set over $\mathcal{S}$; and a *binding labeling* of $\mathcal{G}$ is a binding labeling of $\mathcal{S}$.

Universal generalization can be done whenever a fact has been proven about a variable n and no assumptions have been made about n other than that it is an instance of its own type node. The following definitions identify those variable nodes n such that "no assumptions have been made about n". These definitions have been carefully designed to maximize the deductive power of automatic universal generalization while still ensuring the soundness of universal generalization inferences.

> **Definition**: Let $\mathcal{T}$ be a binding labeling of an Ontic graph $\mathcal{G}$, let β be the binding set of $\mathcal{T}$, and let n be a variable node of $\mathcal{G}$.
>
> We say that two type nodes m and m' are *known to be equal* under $\mathcal{T}$ if the labeling of $\mathcal{T}$ assigns m and m' the same color label.
>
> We say that n is *$\mathcal{T}$-free* if either n is β-free or n is bound under β with a binding $n \mapsto n'$ where n' is a β-free variable node such that the type node of n' is known to be equal to the type node of n under $\mathcal{T}$.
>
> If n is $\mathcal{T}$-free then the *$\mathcal{T}$-freedom-source* for n is defined as follows: If n is β-free then the $\mathcal{T}$-freedom-source for n is n itself. If n is $\mathcal{T}$-free and the binding set of $\mathcal{T}$ contains a binding of the form $n \mapsto n'$ then the $\mathcal{T}$-freedom-source for n is the variable node n'.

There are two forms of universal generalization used in the Ontic system: formula generalization and established type generalization. Formula generalization generalizes the truth of a formula node. Consider a formula node p and a variable node n such that n is a free variable of p. Now suppose that p has been proven to be false without using any assumptions about the particular value for n. In this case one can deduce that the type $\lambda n.p$ is empty; there is no interpretation of n that makes p true. If the type $\lambda n.p$ is empty then it may be possible to determine that a certain existential formula node is **false**. A universal formula is always represented as the negation of an existential formulas so formula generalization can result in assigning a universal formal the label **true**.

Established type generalization is a form of universal generalization that involves subtype links. If $\mathcal{G}$ contains a subtype link $p \Leftrightarrow \; m \prec m'$ then the formula node p represents the statement that every instance of the type m is an instance of the type m'. Thus p represents a universally quantified statement: a statement that quantifies over all instances of the type m. Now suppose that n is a variable with type node m and that m' is an established type for n where no assumptions have been made about n. In this case one can deduce that every instance of m is also an instance of m' so the formula p which represents the subtype relation must be true.

In addition to the two kinds of universal generalization Ontic graphs are associated with existential generalization inferences. If an Ontic graph $\mathcal{G}$ contains an existential link $p \Leftrightarrow \; \exists m$ then the node p represents the statement that there exist instance of the type m. Now if there exists a node r such that m is an established type node for r then one can infer that instances of m exist and therefore that p must be true.

Definition: Let $\mathcal{G}$ be an Ontic graph. Let $\mathcal{T}$ be a binding labeling of $\mathcal{G}$ with binding set β and truth and color labeling $\mathcal{L}$.

We say that a formula node q can be proven false by *$\mathcal{TG}$-formula-generalization* over a variable node n just in case $\mathcal{G}$ contains a closure link $\lambda n.p \; = m$ such that $\mathcal{L}$ assigns p the label **false**, n is $\mathcal{T}$-free with freedom source n', no free variable of p other than n β-depends on n', and $\mathcal{G}$ contains the existential link $q \Leftrightarrow \; \exists m$.

We say that a formula node p can be proven true by *$\mathcal{TG}$-type-establishment-generalization* over a variable node n just in case $\mathcal{G}$ contains a subtype link $p \Leftrightarrow \; m \prec m'$ such that m is the type node for n, m' is a $\mathcal{LG}$-established type node for n, n is $\mathcal{T}$-free with freedom source n' and m' does not β-depend on n'.

We say that a formula node p can be proven true by *$\mathcal{TG}$-existential-generalization* if $\mathcal{G}$ contains an existential link $p \Leftrightarrow \; \exists m$ such that there exists a node r in $\mathcal{G}$ such that m is a $\mathcal{L}$-established-type-node for r.

Under certain binding labelings it is possible to prove that a certain formula node is true even though that node has already been assigned the label **false**. Binding labelings with this property are inconsistent.

> **Definition**: Let $\mathcal{G}$ be an Ontic graph and let $\mathcal{T}$ be a binding labeling of $\mathcal{G}$. We say that $\mathcal{T}$ is $\mathcal{G}$-inconsistent if any of the following conditions hold:
>
> - The color and truth labeling of $\mathcal{T}$is $\mathcal{C}$-inconsistent where $\mathcal{C}$ is the congruence constraint graph underlying $\mathcal{G}$.
> - There exists a formula node p which can be proven false via $\mathcal{T}\mathcal{G}$-formula-generalization but p is labeled **true** under $\mathcal{T}$.
> - There exists a formula node p which can be proven true via either $\mathcal{T}\mathcal{G}$-established-type-generalization or $\mathcal{T}\mathcal{G}$-existential-generalization but p is labeled **false** under $\mathcal{T}$.

Given a definition of the kinds of inferences that are associated with Ontic graphs and the notion of $\mathcal{G}$-inconsistency we can now define the relation $\rightarrow_{\mathcal{G}}$.

> **Definition**: Let $\mathcal{G}$ be an Ontic graph and let $\mathcal{T}$ and $\mathcal{T}'$ be binding labelings of $\mathcal{G}$. We write $\mathcal{T} \rightarrow_{\mathcal{G}} \mathcal{T}'$ if either $\mathcal{T} \rightarrow_{\mathcal{S}} \mathcal{T}'$ where $\mathcal{S}$ is the semantic modulation graph underlying $\mathcal{G}$ or else $\mathcal{T}$ is $\mathcal{G}$-consistent, the binding set of $\mathcal{T}'$ equals the binding set of $\mathcal{T}$, and one of the following conditions holds:
>
> - There exists a formula node p that can be proven false via $\mathcal{T}\mathcal{G}$-formula-generalization and the truth and color labeling of $\mathcal{T}'$ is the result of assigning p the label **false** in the truth and color labeling of $\mathcal{T}$.
> - There exists a formula node p that can be proven true via either $\mathcal{T}\mathcal{G}$-established-type-generalization or $\mathcal{G}\mathcal{T}$-existential-generalization and the truth and color labeling of $\mathcal{T}'$ is the result of assigning p the label **true** in the truth and color labeling of $\mathcal{T}$.

5.6.1 Semantic Soundness

The semantics of full Ontic graphs is very similar to that of semantic modulation graphs. However the semantics of full Ontic graphs must properly account for the meaning of closure and existential links. The precise semantic meaning of closure and existential links is captured in the following definition of a satisfactory semantics for an Ontic graph.

Definition: A *satisfactory semantics* for an Ontic graph $\mathcal{G}$ is a satisfactory semantics $\mathcal{W}$ for the semantic modulation graph underlying $\mathcal{G}$ such that the following conditions hold.

- If $p \Leftrightarrow \exists m$ is an existential link in $\mathcal{G}$ and w is a world in $\mathcal{W}$ then w assigns p the label **true** just in case there exists a color c which is an instance of m in the world w.
- If $\lambda n.p = m$ is a closure link in $\mathcal{G}$ and let w be a world in $\mathcal{W}$ then a color c is an instance of m in w just in case c is an instance of the type of n in w such that if $w[n := c]$ is an assignment of n to c in w then $w[n := c]$ assigns p the label **true**.

The formal language Ontic has an intended semantics which can be defined relative to a fixed universe of mathematical objects (a fixed model of ZFC set theory). The meaning, or denotation, of an Ontic expression can be defined relative to a type respecting variable interpretation; a given interpretation of Ontic variables as mathematical objects yields an interpretation for every Ontic expression. In the graph produced by the Ontic compiler each node is associated with an Ontic expression. Since a type-respecting interpretation of Ontic variables assigns a meaning to every expression, such a variable interpretation can be used to assign labels to the nodes in the graph produced by the Ontic compiler. Thus each variable interpretation yields a world and the set of all such variable interpretation yields a set of worlds, i.e. a semantics. The intended semantics for the graphs produced by the Ontic compiler is a satisfactory semantics in the technical sense defined above.

The semantic soundness theorem for Ontic graphs is analogous to the semantic soundness theorem for semantic modulation graphs.

> $\rightarrow_{\mathcal{G}}$ **Soundness Theorem**: Let $\mathcal{W}$ be a satisfactory semantics for an Ontic graph $\mathcal{G}$. Let $\mathcal{T}$ be a binding labeling of $\mathcal{G}$ with an empty binding set and with a labeling $\mathcal{L}$ such that every world in $\mathcal{W}$ satisfies $\mathcal{L}$. Now suppose $\mathcal{T} \rightarrow_{\mathcal{G}}{}^{*}\ \mathcal{T}'$ where $\mathcal{T}'$ has binding set β and labeling $\mathcal{L}'$. If p is a formula node that is labeled **true** under $\mathcal{L}'$ and such that p does not depend on any variable bound under β then p must be labeled **true** in all worlds in $\mathcal{W}$.

The $\rightarrow_{\mathcal{G}}$ soundness theorem implies that universal and existential generalization as allowed under $\rightarrow_{\mathcal{G}}$ are semantically sound inference techniques. As was the case for $\rightarrow_{\mathcal{S}}$, the $\rightarrow_{\mathcal{G}}$ soundness theorem is proven by showing that $\rightarrow_{\mathcal{G}}$ preserves $\mathcal{W}$-validity. Recall that a binding labeling $\mathcal{T}$ is $\mathcal{W}$-valid if its binding set is $\mathcal{W}$-legal and every world in w that satisfies the binding set of $\mathcal{T}$ also satisfies the truth and color labeling of $\mathcal{T}$. Both the notion of a $\mathcal{W}$-legal binding set and the notion of a $\mathcal{W}$-valid binding labeling are defined purely in terms of the semantics $\mathcal{W}$; these notions do not depend on graph structure and do not need to be redefined here. The proof of the $\rightarrow_{\mathcal{G}}$ preservation theorem uses the following lemma:

> **Freedom Source Lemma**: Let $\mathcal{W}$ be a satisfactory semantics for a semantic modulation graph $\mathcal{G}$. Let $\mathcal{T}$ be a $\mathcal{W}$-valid binding labeling of $\mathcal{G}$ with binding set β and truth and color labeling $\mathcal{L}$. Let n be a $\mathcal{T}$-free variable node with freedom source n'. Let w be a world in $\mathcal{W}$ that satisfies β. If c is an instance of the type of n in w then the semantics $\mathcal{W}$ contains a β-assignment $w[\beta, n' := c]$ of n' to c in w and for any such β-assignment assigns n the color c.
>
> **proof**: Since n' is the freedom source for n then either n' is the same node as n or else β contains the binding $n \mapsto n'$ and $\mathcal{L}$ assigns the same color labels to the type nodes of n and n'. In

either case n' is β-free; any world which satisfies β assign n and n' the same color label; and any world which satisfies $\mathcal{L}$ assigns the type nodes for n and n' the same color label.

Since w satisfies β and $\mathcal{T}$ is $\mathcal{W}$-valid, w must satisfy $\mathcal{L}$ and thus w must assign the type nodes for n and n' the same color label. Thus c is an instance of the type of n' in w. Thus, since β is $\mathcal{W}$-legal and n' is β-free, the semantics $\mathcal{W}$ contains a β-assignment $w[\beta, n' := c]$ of n' to c in w. Furthermore $w[\beta, n' := c]$ satisfies β and assigns n' the color c so $w[\beta, n' := c]$ must also assign n the color c.

$\rightarrow_{\mathcal{G}}$ **Preservation Theorem**: Let $\mathcal{W}$ be a satisfactory semantics for an Ontic graph $\mathcal{G}$. Let $\mathcal{T}$ and $\mathcal{T}'$ be binding labelings for $\mathcal{G}$. If $\mathcal{T}$ is $\mathcal{W}$-valid and $\mathcal{T} \rightarrow_{\mathcal{G}} \mathcal{T}'$ then $\mathcal{T}'$ is $\mathcal{W}$-valid.

Proof: Suppose that $\mathcal{T}$ is $\mathcal{W}$-valid and that $\mathcal{T} \rightarrow_{\mathcal{G}} \mathcal{T}'$. Either $\mathcal{T} \rightarrow_{\mathcal{S}} \mathcal{T}'$ where $\mathcal{S}$ is the semantic modulation graph underlying $\mathcal{G}$ or else $\mathcal{T}'$ is derived from $\mathcal{T}$ by universal or existential generalization. If $\mathcal{T} \rightarrow_{\mathcal{S}} \mathcal{T}'$ then the $\rightarrow_{\mathcal{S}}$ preservation theorem implies that $\mathcal{T}'$ is $\mathcal{W}$-valid. Now suppose $\mathcal{T}'$ is derived from $\mathcal{T}$ by either universal or existential generalization. In this case the binding set of $\mathcal{T}'$ equals the binding set of $\mathcal{T}$; let β be this binding set. By assumption $\mathcal{T}$ is $\mathcal{W}$-valid and thus β is $\mathcal{W}$-legal. It remains only to show that every world in $\mathcal{W}$ which satisfies β also satisfies the truth and color labeling of $\mathcal{T}'$. Let $\mathcal{L}$ be the truth and color labeling of $\mathcal{T}$ and let $\mathcal{L}'$ be the truth and color labeling of $\mathcal{T}'$. Consider a world w in $\mathcal{W}$ which satisfies β. Since $\mathcal{T}$ is $\mathcal{W}$-valid, w satisfies $\mathcal{L}$. Now there are three cases.

First suppose that there exists a formula q which can be proven false via $\mathcal{TG}$-formula-generalization over a variable node n and that $\mathcal{L}'$ is derived from $\mathcal{L}$ assigning q the label **false**. In this case there exists a closure link $\lambda n.p = m$ and an existential link $q \Leftrightarrow \exists m$ such that $\mathcal{L}$ labels p false, n is $\mathcal{T}$-free with freedom source n', and no free variables of p other than n β-depend on n'. To show that $\mathcal{T}'$ is $\mathcal{W}$-valid let w be any world in $\mathcal{W}$ that satisfies

β'. We must show that w satisfies $\mathcal{L}'$. Since $\mathcal{T}$ is $\mathcal{W}$-valid, and since β equals β', the world w must satisfy $\mathcal{L}$. Thus to show that w satisfies $\mathcal{L}'$ it suffices to show that w assigns q the label **false**. Given the semantics of existential links it suffices to show that there are no instances of m in w. The semantics of closure links state that a color c is an instance of m in w just in case c is an instance of the type of n such that if $w[n := c]$ is an assignment of n to c in w then $w[n := c]$ assigns p the label **true**. Let c be any instance of the type of n in w and let $w[n := c]$ be an assignment of n to c in w. To show that there are no instances of m it suffices to show that $w[n := c]$ assigns p the label **false**. By the above freedom source lemma the semantics $\mathcal{W}$ contains a β-assignment $w[\beta, n' := c]$ of n' to c in w and any such β-assignment must assign n the color c. Since $w[\beta, n' := c]$ satisfies β, and since $\mathcal{T}$ is $\mathcal{W}$-valid, the world $w[\beta, n' := c]$ must satisfy the labeling $\mathcal{L}$ and thus $w[\beta, n' := c]$ must assign p the label **false**. It now suffices to show that $w[n := c]$ agrees with $w[\beta, n' := c]$ on the formula p. To show that $w[n := c]$ and $w[\beta, n' := c]$ agree on p it suffices to show that these two worlds agree on the free variables of p. Both $w[n := c]$ and $w[\beta, n' := c]$ assign n the color c. Now consider the free variables of p other than n. Since no free variable of p other than n β-depends on n', $w[\beta, n' := c]$ agrees with w on the free variables of p other than n. Furthermore, the definition of an Ontic graph states that no free variable of p other than n directly depends on n. Thus $w[n := c]$ also agrees with w on the free variables of p other than n. Thus $w[n := c]$ and $w[\beta, n' := c]$ agree on all the free variables of p and thus agree on p.

Now suppose that there exists a formula node p such that p can be proven true via $\mathcal{TG}$-established-type-generalization over a variable node n and that $\mathcal{L}'$ is derived from $\mathcal{L}$ by assigning p true. In this case there exists a subtype link $p \Leftrightarrow m \prec m'$ such that m is the type node of n, n is $\mathcal{T}$-free with freedom source n' and m' is a $\mathcal{LG}$-established-type-node for n such that m' does not β-depend on n'. To show that $\mathcal{T}'$ is $\mathcal{W}$-valid consider a world w that satisfies β'. We must show that w satisfies $\mathcal{L}'$. Since $\mathcal{T}$ is $\mathcal{W}$-valid, and since β equals β', the world w must satisfy $\mathcal{L}$. Thus it suffices to show that w assigns p the label **true**. By the definition of a satisfactory semantics it suffices to show that every instances of m in w is also an instances of m' in w. Let c be an instance of m in w. It suffices to show that c is an instance of m' in w. Since the variable n has type node m, the color c is an instance of the type of n. Thus the above freedom source lemma implies that $\mathcal{W}$ contains a β-assignment $w[\beta, n'; = c]$ of n' to c in w and any such β-assignment assigns n the color c. Since $w[\beta, n' := c]$ satisfies β and since $\mathcal{T}$ is $\mathcal{W}$-valid, $w[\beta, n' := c]$ must satisfy $\mathcal{L}$. Now since m' is a $\mathcal{L}$-established-type-node for n the color of n in $w[\beta, n' := c]$ must be an instance of m' in $w[\beta, n' := c]$. Thus c is an instance of m' in the world $w[\beta, n' := c]$. To show that c is an instance of m' in w it now suffices to show that w and $w[\beta, n' := c]$ agree on m'. But this follows immediately from the assumption that m' does not β-depend on n'.

Now consider existential generalization. Suppose that $\mathcal{G}$ contains an existential link $p \Leftrightarrow \exists m$ such that there exists a node r such that m is a $\mathcal{L}$-established-type-node of r and that $\mathcal{L}'$ is derived from $\mathcal{L}$ by assigning p the label **true**. To show that $\mathcal{T}'$ is $\mathcal{W}$-valid let w be a world in $\mathcal{W}$ that satisfies β'. We must show that w satisfies $\mathcal{L}'$. Since β equals β' and since $\mathcal{T}$ is $\mathcal{W}$-valid the world w must satisfy $\mathcal{L}$. To show that w satisfies $\mathcal{L}'$ it suffices to show that w assigns p the label **true**. Since w satisfies $\mathcal{L}$, and since m is a $\mathcal{L}$-established-type-node for r, the color of r in w must an instance of m in w. But this implies that there exists an instance of m in w so by the semantics of existential links w must assign p the label **true**.

5.6.2 Assumptions

Recall that the notion of $\mathcal{W}$-validity does not allow for assumptions; to properly handle assumptions one must deal with labelings that are not $\mathcal{W}$-valid. To deal with relations that not $\mathcal{W}$-valid we need a new inference relation $\rightarrow_{\mathcal{GA}}$. The relation $\rightarrow_{\mathcal{GA}}$ restricts bindings to avoid binding variables depended on by assumptions in $\mathcal{A}$ and also restricts universal generalization so that one does not generalize over variables depended on by assumptions in $\mathcal{A}$.

Definition: An assumption set over an Ontic graph $\mathcal{G}$ is a set $\mathcal{A}$ of the formula nodes in $\mathcal{G}$.

Let $\mathcal{G}$ be an Ontic graph, let $\mathcal{A}$ be an assumption set over $\mathcal{G}$ and let $\mathcal{T}$ be a binding labeling of $\mathcal{G}$ with binding set β.

A variable node n is called $\mathcal{AT}$-free with freedom source n' just in case n is $\mathcal{T}$-free with freedom source n' and no element of $\mathcal{A}$ β-depends on n'.

It is now possible to define the forms of inference associated with an Ontic graph under a set of assumptions.

Definition: Let $\mathcal{G}$ be an Ontic graph and let $\mathcal{A}$ be an assumption set over $\mathcal{G}$. Let $\mathcal{T}$ be a binding labeling of $\mathcal{G}$.

We say that a formula p can be proven false by *$\mathcal{ATG}$-formula-generalization* over a variable node n just in case p can be proven false by $\mathcal{TG}$-formula-generalization over n and n is $\mathcal{AT}$-free.

We say that a formula p can be proven true by *$\mathcal{ATG}$-established-type-generalization* over a variable node n just in case p can be proven true by $\mathcal{TG}$-established-type-generalization over n and n is $\mathcal{AT}$-free.

As the above definition indicates, the inferences that are allowed in the presence of assumptions are slightly different from the inferences that are allowed when no assumptions are present; certain universal generalization inferences may be allowed in the absence of assumptions but not allowed when assumptions are present. This difference in the allowed inferences is reflected in a difference in the notion of consistency.

Definition: Let $\mathcal{G}$ be an Ontic graph, let $\mathcal{T}$ be a binding labeling of $\mathcal{G}$ and let $\mathcal{A}$ an assumption set over $\mathcal{G}$. We say that $\mathcal{T}$ is $\mathcal{AG}$-inconsistent if any of the following conditions hold:

- The color and truth labeling of $\mathcal{T}$is $\mathcal{C}$-inconsistent where $\mathcal{C}$ is the congruence constraint graph underlying $\mathcal{G}$.
- There exists a formula node p which can be proven false via $\mathcal{ATG}$-formula-generalization but p is labeled **true** under $\mathcal{T}$.
- There exists a formula node p which can be proven true via either $\mathcal{ATG}$-established-type-generalization or $\mathcal{TG}$-existential-generalization but p is labeled **false** under $\mathcal{T}$.

Given a definition of the kinds of inferences that are associated with Ontic graphs under assumptions and the notion of $\mathcal{AG}$-inconsistency we can now define the relation $\rightarrow_{\mathcal{GA}}$.

Definition: Let $\mathcal{G}$ be an Ontic graph, let $\mathcal{A}$ be an assumption set over $\mathcal{G}$, and let $\mathcal{T}$ and $\mathcal{T}'$ be binding labelings of $\mathcal{G}$. We write $\mathcal{T} \rightarrow_{\mathcal{GA}} \mathcal{T}'$ if either $\mathcal{T} \rightarrow_{\mathcal{SA}} \mathcal{T}'$ where $\mathcal{S}$ is the semantic modulation graph underlying $\mathcal{G}$ or else $\mathcal{T}$ is $\mathcal{AG}$-consistent, the binding set of $\mathcal{T}'$ equals the binding set of $\mathcal{T}$, and one of the following conditions holds:

- There exists a formula node p that can be proven false via $\mathcal{ATG}$-formula-generalization and the truth and color labeling of $\mathcal{T}'$ is the result of assigning p the label **false** in the truth and color labeling of $\mathcal{T}$.
- There exists a formula node p that can be proven true via either $\mathcal{ATG}$-established-type-generalization or $\mathcal{GT}$-existential-generalization and the truth and color labeling of $\mathcal{T}'$ is the result of assigning p the label **true** in the truth and color labeling of $\mathcal{T}$.

5.6.3 Soundness under Assumptions

The soundness theorem for the relation $\rightarrow_{\mathcal{GA}}$ is analogous to the soundness theorem for $\rightarrow_{\mathcal{SA}}$.

> $\rightarrow_{\mathcal{GA}}$ **Soundness Theorem**: Let $\mathcal{W}$ be a satisfactory semantics for an Ontic graph $\mathcal{G}$ and let $\mathcal{A}$ be an assumption set over $\mathcal{G}$. Let $\mathcal{T}$ be a binding labeling with an empty binding set and such that every world in $\mathcal{W}$ that satisfies $\mathcal{A}$ also satisfies the truth and color labeling of $\mathcal{T}$. Now suppose $\mathcal{T} \rightarrow_{\mathcal{GA}}{}^{*}\ \mathcal{T}'$ where $\mathcal{T}'$ has binding set β. If p is a formula node such that p is labeled **true** under $\mathcal{L}'$ and no variable depended on by p is bound under β then p must be labeled **true** in all worlds in $\mathcal{W}$ that satisfy $\mathcal{A}$.

The soundness theorem for $\rightarrow_{\mathcal{GA}}$ can be proven by showing that $\rightarrow_{\mathcal{GA}}$ preserves $\mathcal{AW}$-validity. Recall that $\mathcal{T}$ is $\mathcal{AW}$-valid if the binding set of $\mathcal{T}$ is $\mathcal{AW}$-legal and every world in $\mathcal{W}$ that satisfies both $\mathcal{A}$ and the binding set of $\mathcal{T}$ also satisfies the truth and color labeling of $\mathcal{T}$. The notion of $\mathcal{AW}$-validity is defined in a purely semantic way; the $\mathcal{AW}$-validity of the binding labeling $\mathcal{T}$ does not depend on any graph structure and need not be redefined here.

> $\rightarrow_{\mathcal{GA}}$ **Preservation Theorem**: Let $\mathcal{W}$ be a satisfactory semantics for an Ontic graph $\mathcal{G}$ and let $\mathcal{A}$ be an assumption set for $\mathcal{G}$. If $\mathcal{T}$ is an $\mathcal{AW}$-valid binding labeling and $\mathcal{T} \rightarrow_{\mathcal{GA}} \mathcal{T}'$, then $\mathcal{T}'$ is also $\mathcal{AW}$-valid.

The proof of the $\rightarrow_{\mathcal{GA}}$ preservation theorem is directly analogous to the proof of the $\rightarrow_{\mathcal{G}}$ preservation theorem and is not given here. The proof relies on the fact that if n is $\mathcal{AT}$-free with freedom source n' then no element of $\mathcal{A}$ β-depends on n' where β is the binding set of $\mathcal{T}$. More specifically, if n' is β-free and no element of $\mathcal{A}$ β-depends on n' then, by definition, n' is $\mathcal{A}\beta$-free. Since n' is $\mathcal{A}\beta$-free, and β is $\mathcal{AW}$-legal, β-assignments exist for n' in all worlds that satisfy both β and $\mathcal{A}$. If n' were β-free but not $\mathcal{A}\beta$-free then the $\mathcal{AW}$-legality of β would not ensure that β-assignments exist for n'.

5.6.4 Focus, Termination and Order Independence

Of course it is possible to control the generation of bindings with focus objects. A *focus set* over an Ontic graph $\mathcal{G}$ is simply a subset of the nodes of $\mathcal{G}$. One can define the relation $\rightarrow_{\mathcal{GFA}}$ as a restriction of the relation $\rightarrow_{\mathcal{GA}}$; the relation $\rightarrow_{\mathcal{GFA}}$ never bindings variables which are $\mathcal{F}$-protected, only binds variables to focus objects and never binds two variables with the same type node to the same focus object. Because the relation $\rightarrow_{\mathcal{GFA}}$ is a restriction of the relation $\rightarrow_{\mathcal{GA}}$ it clearly preserves $\mathcal{AW}$-validity.

Order independence for the relation that $\rightarrow_{\mathcal{GFA}}$ requires a restriction an universal generalization inferences. More specifically the freedom source of the variable being generalized over in a universal generalization inference must be $\mathcal{F}$-protected. This ensures that no binding operation allowed under $\rightarrow_{\mathcal{GFA}}$ binds the freedom source involved in a universal generalization inference. This in turn ensures that all allowed universal generalization inferences commute with all allowed binding operations. This restriction on universal generalization inference has not been a problem in practice.

Chapter 6

The Ontic Language

The formal language Ontic consists of twenty three kinds of expression plus seven macros that provide convenient abbreviations for expressions. The Ontic compiler converts a set Σ of Ontic expressions to an Ontic graph $G(\Sigma)$. The graph $G(\Sigma)$ is simpler than the set Σ; although there can be twenty three different kinds of expressions in Σ there are only nine kinds of links in Ontic graphs. The compiler is described in chapter 7, the current chapter describes the language Ontic and various syntactic properties of that language.

There are several aspects of the syntax of the Ontic language that need explaining. First of all, most of the axioms of Zermelo-Fraenkel set theory are encoded in the notion of a syntactically small type expression; a type expression can be "reified" as a set only if the type expression is syntactically small. This chapter also describes free variables and substitution; the type system used in the Ontic language makes these notions somewhat complex.

6.1 Non-Minimality of the Ontic Language

The Ontic language is not semantically minimal; many of the constructs in the Ontic language could be semantically defined in terms of more basic constructs. There are three reasons for the non-minimality of the Ontic language.

First, the Ontic system encodes the axioms of set theory in the syntax of the Ontic language. Second, the non-minimality of the Ontic language allows the compilation process to generate efficient graph structure. There is an analogy between the non-minimality of the Ontic language and the non-minimality of programming languages — greater efficiency is achieved by allowing the compiler to directly implement certain non-minimal language features. Finally, directly compiling non-minimal language features improves the input-output behavior of the system; there are automatic inferences based on the graph structure generated from the non-minimal language which would not be done automatically if the compilation process was restricted to a minimal language.

The notion of a *syntactically small* type expression encodes many of the axioms of set theory. Rather than have explicit comprehension axioms, the Ontic system allows the construction of sets of the form

`(THE-SET-OF-ALL` τ`)`

where τ is a syntactically small type expression. Not all type expressions are syntactically small; the types `SET`, `GROUP`, `FIELD`, or `TOPOLOGICAL-SPACE` are all large and an error is generated if an attempt is made to construct the set of all sets or the set of all topological spaces. On the other hand if s is a term that denotes a set then the type

`(SUBSET-OF` s`)`

is syntactically small and one can construct the set

`(THE-SET-OF-ALL (SUBSET-OF` s`))`

The smallness of types of the form `(SUBSET-OF` s`)` corresponds to the axiom of power set; for every set s there exists another set $P(s)$ such that $P(s)$ contains all subsets of s. The smallness of types of the form `(EITHER` t_1 t_2`)` corresponds to the set theoretic axiom of pairing. The smallness of lambda types corresponds to the axiom of restricted comprehension and the smallness of types of the form `(RANGE-TYPE` f`)` correspond to the axioms of union, and replacement.

The non-minimality of the Ontic language also allows the graph $G(\Sigma)$ to be smaller than it would be otherwise. For example consider a type expression of the form

$$\texttt{(OR-TYPE } \tau_1 \; \tau_2\texttt{)}$$

An object is an instance of this type just in case it is an instance of either the type τ_1 or the type τ_2. Semantically this type is equivalent to the type

$$\texttt{(LAMBDA ((X THING)) (OR (IS X } \tau_1\texttt{) (IS X } \tau_2\texttt{)))}$$

However the lambda type quantifies over the type `THING` and generates additional graph structure for each variable of type `THING`. By implementing the `OR-TYPE` operator as a primitive one can avoid quantifying over the type `THING` and thus create less graph structure. The primitive implementations of `IF`, `EITHER` and `RANGE-TYPE` lead to similar savings in the amount of graph structure created.

The non-minimality of the Ontic language also leads to greater inferential power. For example consider the reification of functions. Expression in the Ontic language are divided into five syntactic categories: terms, formulas, functions, types and type-generators. Of these five categories terms are the only first class objects; variables can be bound only to terms and only terms can be used to specify focus objects. However certain type expressions (syntactically small type expressions) can be *reified*, i.e. coerced to a term via the operator `THE-SET-OF-ALL`. Furthermore, functions can be reified, or coerced to terms, via the operator `THE-RULE`. If f is a syntactically small function expression which takes one argument then the Ontic expression (`THE-RULE` f) denotes the set of pairs that corresponds to the function f. Unlike the function expression f, the term expression (`THE-RULE` f) is a first class object; variables can be bound to it and it can be used as a focus object in an Ontic context. The operator `THE-RULE` is not semantically minimal; it is possible to define the operator `THE-RULE` using the operator `THE-SET-OF-ALL`. However the primitive implementation of the operator `THE-RULE` allows the system to perform inferences in a single step that would take many steps if the system were forced to reason purely in terms of the operator `THE-SET-OF-ALL`. More specifically the Ontic language includes the operator `APPLY-RULE` such that for any syntactically small function f of one argument the implementation of the operator `THE-RULE` allows the system to derive the following equation in a single step.

```
(IS (APPLY-RULE (THE-RULE f) x)
    (EQUAL-TO (f x)))
```

If THE-RULE were a macro that expanded to an expression involving THE-SET-OF-ALL then the above equation would have to be proved using a several step proof for each reified function f. One can not state the above equation as a lemma about all functions because one can not quantify over functions. However one can quantify over rules and the operator THE-RULE provides a way of reifying syntactically small functions as rules.

6.2 The Ontic Language

The expressions in the Ontic language are divided into four categories: terms, functions, formulas, types and type generators. Terms are expressions that denote mathematical objects such as sets, pairs, graphs, partially ordered sets and lattices. Function expressions denote operators (functions) that map objects to objects. Formulas are expressions that are either true or false in any given interpretation. Type expressions denote one place predicates on objects; if τ is a type expression and the predicate denoted by τ is true of an object x, then we say that x is an *instance* of the type τ. Type generators are operators which take arguments (which are always terms) and return a type. For example the type generator *GREATER-THAN* takes a partially ordered set P and an element x of P and returns a type whose instances are the elements of P which are greater than x under the ordering imposed by P.

Functions, types, and type generators can be λ-expressions. A λ-expression is an expression of the form

(LAMBDA ((X$_1$ τ_1) (X$_2$ τ_2) ... (X$_k$ τ_k)) *body*)

A λ-expression always denotes an operator; the above expression is an operator that takes k arguments where each argument must be an instance of the associated type. If the body of a λ-expression is a formula then the expression is a type expression and is only allowed to take one argument. If the body is a term then the λ-expression is a function; if the body is a type then the λ-expression is a type generator.

There are actually two versions of the Ontic language which differ in the way variables are treated. The first version of the language is the one used in the top level user interface. In this external version of the Ontic language a variable is simply a symbol such as `X` and the same symbol can be used in different ways in different contexts. The external version of the language should be distinguished from the internal version where individual variables have more structure and stronger identity.

There is a one to one correspondence between the nodes in the graph generated by the Ontic compiler and expressions in the internal language. In particular there is a one to one correspondence between variable nodes in the graph structure and variables of the internal language. This one to one correspondence would be impossible for the external language because the external language allows a given symbol to be used as variables of different types in different contexts. In the internal version of the Ontic language each variable has a fixed type that is taken to be a syntactic property of that variable. The following λ-type is an example of an external expression:

```
(LAMBDA ((X SET))
  (IS-EVERY (MEMBER-OF X) SET))
```

This external expression gets mapped to the following internal expression

$$\texttt{(LAMBDA (}x^{\texttt{SET}}\texttt{)}$$
$$\texttt{(IS-EVERY (MEMBER-OF }x^{\texttt{SET}}\texttt{) SET))}$$

Note that in the translation process the external symbol `X` has been replaced by the internal variable $x^{\texttt{SET}}$ of type `SET`.

Only the internal language is formally defined here. Fortunately, the external and internal versions of the Ontic language are very similar and the definition of the external language should be clear from the definition of the internal language. A method of translating external expressions into internal expressions is discussed in a later section.

An internal Ontic expression can be formally defined as one of the twenty three different kinds of expressions listed below.

Definition: An *internal Ontic expression* is one of the following:

- A type expression which is one of the following:
  - One of the type symbols `THING`, `SET`, `RULE` or `SYMBOL`. The type `SYMBOL` is syntactically small while the types `THING`, `SET`, and `RULE` are all large.
  - An application of the form $(g\ t_1\ t_2 \ldots t_k)$ where g is a type generator of k arguments and each t_i is a term. A type expression of this form is syntactically small just in case the type generator g is syntactically small.
  - A λ-type of the form `(LAMBDA` (x^τ) Φ`)` where x^τ is variable of type τ and Φ is a formula. A type of this form is syntactically small just in case the domain type τ is syntactically small. The class of instances of this type is a subclass of the instances of the type τ.
  - An expression of the form `(OR-TYPE` τ_1 τ_2`)` where τ_1 and τ_2 are types. A type expression of this form is syntactically small just in case both the types τ_1 and τ_2 are syntactically small.
  - An expression of the form `(RANGE-TYPE` f`)` where f a function expression of any number of arguments. A type expression of this form is syntactically small just in case the function expression f is syntactically small.
- A term which is one of the following:
  - A variable x^τ where τ is a type expression. Each type τ is associated with an infinite sequence x_1^τ, x_2^τ, $x_3^\tau \ldots$ of variables of type τ.
  - An application of the form $(f\ t_1\ t_2 \ldots t_k)$ where f is a function expression of k arguments and each t_i is a term.
  - An expression of the form `(THE-SET-OF-ALL` τ`)` where τ is a syntactically small type expression.
  - An expression of the form `(THE` τ`)` where τ is a syntactically small type expression.
  - A conditional expression of the form `(IF` Φ t_1 t_2`)` where Φ is a formula and t_1 and t_2 are terms.

  - An expression of the form (THE-RULE f) where f is a syntactically small λ-function of one argument.
  - An expression of the form (QUOTE *symbol*) where *symbol* is an atomic symbol.
- A function expression which is one of the following:
  - A λ-function of k arguments of the form

    (LAMBDA ($x_1^{\tau_1}$ $x_2^{\tau_2}$ ... $x_k^{\tau_k}$) *body*)

    where each $x_i^{\tau_i}$ is a variable of type τ_i and *body* is a term. A λ-function is syntactically small just in case each type expression τ_i is syntactically small.
  - An expression of the form (THE-FUNCTION t) where t is a term. The term t should denote an instance of the type RULE, i.e. something expressible as (THE-RULE f). All functions of this form are functions of one argument and are syntactically small.
  - The primitive function symbol RULE-DOMAIN which is a large function of one argument. This function should only be applied to instances of the type RULE.
- A formula which is one of the following:
  - A type formula of the form (IS t τ) where t is a term and τ is a type expression.
  - An existence formula of the form (EXISTS-SOME τ) where τ is a type expression.
  - An equality of the form (= e_1 e_2) where e_1 and e_2 are any internal Ontic expressions.
  - A Boolean application of formulas constructed with one of the boolean operators NOT, OR, AND, IMPLIES, or IFF.
  - A subtype formula of the form (IS-EVERY σ τ) where σ and τ are type expressions.
- A type generator expression which is one of the following:
  - One of the primitive type generators EQUAL-TO, MEMBER-OF, SUBSET-OF, EITHER or RULE-BETWEEN. The type gener-

ators `EITHER` and `RULE-BETWEEN` both take two arguments, all the others take one. All these type generators are syntactically small.

  – A non-primitive type generator of k arguments of the form

$$\texttt{(LAMBDA (}x_1^{\tau_1}\ x_2^{\tau_2}\ \ldots\ x_k^{\tau_k}\texttt{)}\ \mathit{body}\texttt{)}$$

  where *body* is a type expression. A type generator of this form is syntactically small just in case the type *body* is syntactically small.

- An unclassified combinator expression. Combinator expressions are generated when a λ-type is compiled into graph structure. Combinator expressions are discussed in chapter 7.

The large size of the internal language makes it difficult to define properties of expressions; to define an operation on internal expressions it seems that one must define that operation on each of the twenty three different kinds of expressions. Fortunately this problem can be avoided. More specifically the twenty three different kinds of expressions can be classified into four groups: atomic expressions, variables, lambda expressions, and extensional applications.

Definition: An *atomic expression* is either one of the primitive type symbols, one of the primitive type generator symbols, or a quotation of the form `(QUOTE` *symbol*`)`.

A λ-*expression* is either a λ-type, a λ-function or a non-primitive type generator.

An *extensional application* is an expression other than a variable, an atomic expression or a λ-expression. All extensional applications have the form

$$(op\ arg_1\ arg_2\ \ldots\ arg_k)$$

6.3 Binding and Freedom

There are some subtleties in the internal language concerning the notion of a free variable. The external formula

```
(EXISTS ((X (MEMBER-OF S)))
   (IS X (MEMBER-OF U)))
```

Is an abbreviation for the external formula

```
(EXISTS-SOME
  (LAMBDA ((X (MEMBER-OF S)))
    (IS X (MEMBER-OF U))))
```

Which corresponds to the internal formula

$$\texttt{(EXISTS-SOME}$$
$$\quad\texttt{(LAMBDA (}x^{\texttt{(MEMBER-OF } s^{\texttt{SET}}\texttt{)}}\texttt{)}$$
$$\quad\quad\texttt{(IS } x^{\texttt{(MEMBER-OF } s^{\texttt{SET}}\texttt{)}} \texttt{ (MEMBER-OF } u^{\texttt{SET}}\texttt{))))}$$

This formula says that there exists a member of $s^{\texttt{SET}}$ which is also a member of $u^{\texttt{SET}}$. Thus the variable $s^{\texttt{SET}}$ must be a free variable of this formula. Note however that $s^{\texttt{SET}}$ appears in the type of the bound variable $x^{\texttt{(MEMBER-OF } s^{\texttt{SET}}\texttt{)}}$. More generally consider any λ-type of the form

$$\texttt{(LAMBDA (}x^{\tau}\texttt{) }\Phi\texttt{)}$$

A free variable in the type τ is considered to be free in the λ-type.

In general consider a λ-expression of the form

$$\texttt{(LAMBDA (}x_1^{\tau_1}\ x_2^{\tau_2}\ \ldots\ x_k^{\tau_k}\texttt{) } body\texttt{)}$$

If this λ-expression is a λ-type then it denotes the class of instances of that type. If the λ-expression is a function or type generator then it denotes a certain class of tuples. In either case the meaning of the λ-expression depends on the classes associated with the types τ_i which in turn can depend on the interpretation of free variables in the type expressions. Thus the free variables of a λ-expression include free variables in the types of the bound parameters.

Definition: A variable y^σ *appears free* in an internal expression e if one of the following conditions hold:

- e is the variable y^σ.
- e is an extensional application

  $$(op\ arg_1\ arg_2\ \ldots\ arg_k)$$

  and y^σ either appears free in the operator op or one of the arguments arg_i
- e is a λ-expression of the form

  $$\texttt{(LAMBDA (}x_1^{\tau_1}\ x_2^{\tau_2}\ \ldots\ x_k^{\tau_k}\texttt{)}\ body\texttt{)}$$

  where y^σ is not equal to any $x_i^{\tau_i}$ and y^σ appears free either in *body* or the type τ_i of some formal parameter $x_i^{\tau_i}$.

Note that in λ-functions and type generators of more than one argument a free variable in the type of one argument may be bound as another argument. For example consider the type generator GREATER-OR-EQUAL-TO defined in the external language as follows.

```
(DEFTYPE (GREATER-OR-EQUAL-TO (X (IN-USET P)) (P POSET))
  (LAMBDA ((Y (IN-USET P)))
    (OR (= Y X)
        (IS Y (GREATER-THAN X P)))))
```

The type generator GREATER-OR-EQUAL-TO takes two arguments X and P where P is a partially ordered set and X is a member of P. The above definition introduces the symbol GREATER-OR-EQUAL-TO as an abbreviation for an internal type generator of the form

$$\texttt{(LAMBDA (}x^{\texttt{(IN-USET }p^{\texttt{POSET}}\texttt{)}}\ p^{\texttt{POSET}}\texttt{)}\ body\texttt{)}$$

In this expression the variable $p^{\texttt{POSET}}$ which appears free in the type of the bound variable $x^{\texttt{(IN-USET }p^{\texttt{POSET}}\texttt{)}}$ is bound as the second argument and thus does not appear free in the overall expression.

The definition of the free variables of an expression may seem problematic. In particular consider an external λ-expression of the form

```
(LAMBDA ((X (MEMBER-OF Y)) (Y (MEMBER-OF X))) body)
```

According to the definition given above both occurrences of X and Y in the type expressions are bound as arguments to the λ-expression. But there is a circularity in the typing of the formal parameters; the expression takes two arguments X and Y where X is a member of Y and Y is a member of X. It turns out that no internal λ-expression has circularities of this kind. Any attempt to translate circular external expressions into the internal language produces an error. To see why internal λ-expression are non-circular we need to define the notion of rank for internal expressions.

Definition:

- If e is an atomic expression then the rank of e is 0.
- If e is a variable x^{τ} then the rank of e is one greater than the rank of the type τ.
- If e is an extensional application

  $$(op\ arg_1\ arg_2\ \ldots\ arg_k)$$

  then the rank of e is one greater than the maximum rank of op and the arguments arg_i.
- If e is a λ-expression

  $$(\texttt{LAMBDA}\ (x_1^{\tau_1}\ x_2^{\tau_2}\ \ldots\ x_k^{\tau_k})\ body)$$

  then the rank of e is one greater than the maximum rank of $body$ and variables $x_i^{\tau_i}$.

Lemma: All parameter lists in the internal expression are non-circular, i.e. for any parameter list $(x_1^{\tau_1}\ x_2^{\tau_2}\ \ldots\ x_k^{\tau_k})$ there exists a permutation $(y_1^{\tau_1}\ y_2^{\tau_2}\ \ldots\ y_k^{\tau_k})$ of this list such that if $y_i^{\tau_i}$ appears free in the type expression τ_j then i must be less than j.

Proof: Let

$$(y_1^{\tau_1}\ y_2^{\tau_2}\ \ldots\ y_k^{\tau_k})$$

be a permutation of the list which sorts the parameters by rank, i.e. if i is less than j then the rank of $y_i^{\tau_i}$ is less than or equal to the rank of $y_j^{\tau_j}$. Now suppose that $y_i^{\tau_i}$ appears free in τ_j. We must show that in this case i is strictly less than j. It follows from the definition of rank that if $y_i^{\tau_i}$ appears free in τ_j then the rank of τ_j must be greater than the rank of $y_i^{\tau_i}$. Furthermore the rank of $y_j^{\tau_j}$ is one greater than the rank of τ_j. Thus the rank of $y_i^{\tau_i}$ must be less then the rank of $y_j^{\tau_j}$ so i must be less than j.

6.4 Translating External Expressions

The syntax of the external language is similar to the syntax of the internal language except that external symbols are used rather than variables and the syntax of λ-expressions is slightly different. The definition of when a symbol X *appears free* in an external expression e is directly analogous to the corresponding definition for the internal language.

The translation of an external expression into an internal expression is defined relative to a *symbol translation table* which contains entries of the form

$$\mathtt{X} \mapsto e$$

where X is an external symbol and e is an internal expression. Each context in the Ontic system is associated with a particular symbol translation table; different translation tables are used in different contexts. If σ is a type expression in the external language then the context construction operation

$$(\mathtt{LET\text{-}BE\ X}\ \sigma)$$

constructs a context where the symbol translation table includes the entry

$$\mathtt{X} \mapsto x^{\sigma'}$$

where $x^{\sigma'}$ is an internal variable of type σ' where σ' is the type expression in the internal language that corresponds to the external type expression σ. If

t is a term in the external language then the context constructor

`(LET-BE X` t`)`

yields a context where the symbol translation table contains the entry

$$\mathtt{X} \mapsto t'$$

where t' is the internal term corresponding to the external term t. The same symbol can be used in different ways in different contexts.

Now consider an external λ-expression of the form

`(LAMBDA ((X` τ`))` *body*`)`

To translate this expression relative to a given translation map ρ the system first translates the external type expression τ to an internal expression τ'. If there is some free symbol in τ which is not mapped by ρ then the translation of τ fails. The system then chooses an internal variable $x^{\tau'}$ such that $x^{\tau'}$ does not appear in ρ, i.e. $x^{\tau'}$ does not appear free in any term t which is the right hand side of a mapping $\mathtt{Y} \mapsto t$ in the table ρ. The system then translates *body* relative to the table $\rho[\mathtt{X}\mapsto x^{\tau'}]$ which is the table identical to ρ except that it maps `X` to $x^{\tau'}$. Let *body'* be the result of translating *body* relative to this modified table. The overall translation process then yields the internal λ-expression

`(LAMBDA (`$x^{\tau'}$`)` *body'*`)`

The general translation process can be precisely defined by a simple case analysis on the syntax of external expressions.

Definition: If e is an external expression and ρ is a symbol translation table then the translation *Trans*(e, ρ) of the expression e with respect to the table ρ is defined as follows:

- If e is an atomic expression then *Trans*(e, ρ) equals e.
- If e is an external symbol then *Trans*(e, ρ) equals $\rho(e)$.
- If e is an application

$$(op\ \ arg_1\ \ arg_2\ \ \ldots\ \ arg_k)$$

then *Trans*(e, ρ) equals

(*Trans*(*op*, ρ) *Trans*(arg_1, ρ) *Trans*(arg_2, ρ) ... *Trans*(arg_k, ρ))

- If e is a lambda expression of the form

  (LAMBDA ((X_1 τ_1) ... (X_k τ_k)) *body*)

  then let ρ' be

  NewMap(ρ, ((X_1 τ_1) ... (X_k τ_k)))

  where the function *NewMap* is defined below. The translation *Trans*(e, ρ) is then defined to be

  (LAMBDA ($\rho'(X_1)$ ... $\rho'(X_k)$) *Trans*(*body*, ρ'))

Let *arglist* be an argument list of the form ((X_1 τ_1) ... (X_k τ_k)) and let ρ be a symbol translation table. If *arglist* is empty then the translation table *NewMap*(ρ, *arglist*) equals the table ρ. If *arglist* is not empty then the table *NewMap*(ρ, *arglist*) is defined as follows:

- let (X_i τ_i) be a pair in *arglist* such there is no pair (X_j τ_j) in *arglist* such that X_j appears free in τ_i. If no such pair (X_i τ_i) exists then there is a circularity in the type structure of *arglist* and the attempt to construct a new translation table fails.
- Let τ_i' be *Trans*(τ_i, ρ) and let $x^{\tau_i'}$ be the first variable of type τ_i' which does not appear in ρ, i.e. which does not appear free in any term t which is the right hand side of a mapping $Y \mapsto t$ in ρ.
- Let ρ' be the table $\rho[X_i \mapsto x^{\tau_i'}]$ which is identical to ρ except that it maps X_i to $x^{\tau_i'}$ and let *restargs* be the result of removing the pair (X_i τ_i) from *arglist*.
- *NewMap*(ρ, *arglist*) equals *NewMap*(ρ', *restargs*)

Lemma: If ρ' is a translation table of the form

$$NewMap(\rho, \texttt{((}\texttt{X}_1\ \tau_1\texttt{)}\ \ldots\ \texttt{(}\texttt{X}_k\ \tau_k\texttt{))})$$

then for any pair $(\texttt{X}_i\ \tau_i)$ in the given argument list $\rho'(\texttt{X}_i)$ is an internal variable of type $Trans(\tau_i, \rho')$

When translating λ-expressions the system chooses internal variables which replace external symbols. The internal variables of each type τ are ordered in a linear sequence x_1^τ, x_2^τ, x_3^τ, etc. When the system chooses an internal variable of type τ it always chooses the first acceptable variable in this sequence. In this way the least possible number of distinct variables appear in the internal expression resulting from the translation. Minimizing the number of distinct variables that appear in the output expression reduces the size of the graph generated by the compilation process; the size of the graph is quite sensitive to the number of distinct variables of a given type which appear in the expressions being compiled.

6.5 Substitution

Given the notion of a free variable we can now define the notion of substitution. If e is any internal expression, y^σ is any internal variable, and t is any internal term, the expression $e[t/y^\sigma]$ is the result of replacing all free occurrences of y^σ in e by t with appropriate renaming of bound variables in e. For example suppose e is a λ-expression of the form

$$\texttt{(LAMBDA (}x_1^{\tau_1}\ x_2^{\tau_2}\ \ldots\ x_k^{\tau_k}\texttt{)}\ body\texttt{)}$$

The free variables of this expression may include free variables in the type expressions τ_i and computing $e[t/y^\sigma]$ may involve substituting into a type τ_i of a formal parameter. Thus if e is a lambda expression then the formal parameters of $e[t/y^\sigma]$ may have different types than the formal parameters of e and thus the formal parameters of $e[t/y^\sigma]$ must be different from the formal parameters of e. To properly define substitution for internal Ontic expressions one must use the more general notion of a simultaneous substitution for a set of expressions.

Definition: A *substitution* ω is a finite set of mappings of the form

$$y^\sigma \mapsto t$$

where y^σ is an internal variable and t is an internal term and a given variable has at most one mapping under ω.

The expression $e[t/y^\sigma]$ is defined to be $\omega(e)$ where ω is the substitution containing the single mapping $y^\sigma \mapsto t$.

For any substitution ω and any internal expression e, the expression $\omega(e)$ is defined as follows:

- If ω does not contain a mapping for any free variable in e then $\omega(e)$ equals e.
- If e a variable y^σ and ω contains a mapping of the form $y^\sigma \mapsto t$ then $\omega(e)$ equals t.
- If e is an extensional application of the form

  $$(op\ arg_1\ arg_2\ \ldots\ arg_k)$$

  then $\omega(e)$ equals

  $$(\omega(op)\ \omega(arg_1)\ \omega(arg_2)\ \ldots\ \omega(arg_k))$$

- If e is a λ-expression of the form

  $$(\texttt{LAMBDA}\ (x_1^{\tau_1}\ x_2^{\tau_2}\ \ldots\ x_k^{\tau_k})\ body)$$

  then let *freevars* be the set of free variables of e then let ω' be the substitution

  $$NewSubst(\omega,\ (x_1^{\tau_1}\ x_2^{\tau_2}\ \ldots\ x_k^{\tau_k}),\ freevars)$$

  where then function *NewSubst* is defined below. In this case $\omega(e)$ equals

  $$(\texttt{LAMBDA}\ (\omega'(x_1^{\tau_1})\ \omega'(x_2^{\tau_2})\ \ldots\omega'(x_k^{\tau_k}))\ \omega'(body))$$

Let ω be a substitution, let *arglist* be an argument list of the form $(x_1^{\tau_1}\ x_2^{\tau_2}\ \ldots\ x_k^{\tau_k})$ and let *freevars* be a set variables. If *arglist* is empty then the substitution *NewSubst*(ω, *arglist*, *freevars*) equals the substitution ω. If *arglist* is not empty then

NewSubst(ω, *arglist*, *freevars*)

is defined as follows:

- Let $x_i^{\tau_i}$ be a member of the argument list such that no variable $x_j^{\tau_j}$ in the argument list appears free in τ_i. Such an argument must exist because there must be some argument of least rank.
- Let $z^{\omega(\tau_i)}$ be the first variable of type $\omega(\tau_i)$ such that for every variable y^σ in *freevars* either there exists a mapping $y^\sigma \mapsto t$ in ω and $z^{\omega(\tau_i)}$ does not occur free in t or there is no mapping $y^\sigma \mapsto t$ in ω and $z^{\omega(\tau_i)}$ is distinct from y^σ. [1]
- Let ω' be $\omega[x_i^{\tau_i} \mapsto z^{\omega(\tau_i)}]$ which is identical to ω except that it maps $x_i^{\tau_i}$ to $z^{\omega(\tau_i)}$.
- Let *arglist'* be *arglist* minus the argument $x_i^{\tau_i}$.
- Let *freevars'* be *freevars* plus the variable $x_i^{\tau_i}$.
- *NewSubst*(ω, *arglist*, *freevars*) equals

  NewSubst(ω', *arglist'*, *freevars'*)

Recall that for each type τ the variables of type τ are ordered in a linear sequence x_1^τ, x_2^τ, x_3^τ, etc. The above algorithm specifies that whenever bound variables are renamed, and a variable of type τ must be chosen as a replacement for some other variable, one must take the earliest possible variable of type τ. This minimizes the number of variables which ultimately get translated into graph structure.

[1]The first condition ensures that free variables introduced by ω are not captured by the new bound variables. The second condition ensures that members of *freevars* not mapped by ω are not captured by the new bound variables.

6.6 Macros

The External language includes certain macros that provide convenient abbreviations. The most important macros used in the external language are EXISTS and FORALL. The external expression

```
(EXISTS ((X τ)) Φ)
```

is an abbreviation for the external formula

```
(EXISTS-SOME
  (LAMBDA ((X τ))
    Φ))
```

In general the quantifier EXISTS can involve more than one bound variable. For example consider an external formula of the form

```
(EXISTS ((X (IN-USET P))
         (P POSET))
    Φ)
```

This formula abbreviates the formula

```
(EXISTS ((P POSET))
  (EXISTS ((X (IN-USET P)))
    Φ))
```

Which becomes

```
(EXISTS-SOME
  (LAMBDA ((P POSET))
    (EXISTS-SOME
      (LAMBDA ((X (IN-USET P)))
        Φ))))
```

In general the formula

$$\texttt{(EXISTS ((X}_1\ \tau_1\texttt{)}\ \ldots\ \texttt{(X}_k\ \tau_k\texttt{))}\ \Phi\texttt{)}$$

Abbreviates the formula

```
(EXISTS ((X_i τ_i))
  (EXISTS ((X_1 τ_1)
           ⋮
           (X_{i-1} τ_{i-1})
           (X_{i+1} τ_{i+1})
           ⋮
           (X_k τ_k))
    Φ))
```

Where no $\mathtt{X}_j$ appears free in τ_i. This requirement insures that none of the bound symbols $\mathtt{X}_i$ appear free in the overall expression. If every τ_i has a free occurrences of some $\mathtt{X}_j$ then the macro expansion fails.

The macro `FORALL` is defined in terms of `EXISTS`. More specifically

```
(FORALL ((X_1 τ_1) ... (X_k τ_k)) Φ)
```

abbreviates

```
(NOT (EXISTS ((X_1 τ_1) ... (X_k τ_k)) (NOT Φ)))
```

The following list shows some additional macros where σ and each τ_i are external type expressions, t and u are external terms f is an external function expression of one argument, each $\mathtt{X}_i$ is an external symbol and `Y` and `Z` are external symbols distinct from all $\mathtt{X}_i$ and which do not appear free in t, u, f, σ or any τ_i.

Macro Expression	**Expansion**
`(AND-TYPE τ_1 τ_2)`	`(LAMBDA ((Y τ_1)) (IS Y τ_2))`

(WRITABLE-AS t (X$_1$ τ_1) $\vdots$ (X$_k$ τ_k))	(RANGE-TYPE (LAMBDA ((X$_1$ τ_1) $\vdots$ (X$_k$ τ_k)) t))
(WRITABLE-AS σ (X$_1$ τ_1) $\vdots$ (X$_k$ τ_k))	(WRITABLE-AS Y (X$_1$ τ_1) $\vdots$ (X$_k$ τ_k) (Y σ))
(AT-MOST-ONE σ)	(FORALL ((Y σ) (Z σ)) (= Z Y))
(EXACTLY-ONE σ)	(AND (EXISTS-SOME σ) (AT-MOST-ONE σ))
(APPLY-RULE t u)	((THE-FUNCTION t) u)

In addition to the macros specified above the external language allows some simple syntactic abbreviations involving operators and macros which take a single type as an argument. More specifically the expression

(THE-SET-OF-ALL ((X τ)) Φ)

abbreviates

(THE-SET-OF-ALL (LAMBDA ((X τ)) Φ))

Similarly

(THE ((X τ)) Φ)

abbreviates

(THE (LAMBDA ((X τ)) Φ))

The operators AT-MOST-ONE, EXACTLY-ONE and THE-RULE allow for similar abbreviations.

The macros `EXISTS` and `FORALL` also allow abbreviated type expressions in the list of bound variables. For example the expression

(FORALL ((X τ Φ)) Ψ)

says that Ψ holds for every X of type τ such that Φ. This formula abbreviates (FORALL ((X σ)) Ψ) where σ is the type (LAMBDA ((X τ)) Φ).

6.7 Definitions

Of course the external Ontic language allows for user specified definitions. A definition is an expression of the form

(DEFINE *symbol* *e*)

where *symbol* is an external symbol and *e* is any external expression. A definition of this form alters the base level symbol translation table so that *symbol* gets translated as the expression e' where e' is the internal translation of *e*.

Definitions can be made more concise with the macros `DEFTYPE` and `DEFTERM`. For example the definition

(DEFTYPE *symbol* τ)

is the same as

(DEFINE *symbol* τ)

but the definition

```
(DEFTYPE (symbol (X_1 τ_1) ... (X_k τ_k))
   τ)
```

is an abbreviation for the definition

```
(DEFINE symbol
  (LAMBDA ((X_1 τ_1) ... (X_k τ_k))
    τ))
```

Similarly the definition

```
(DEFTERM symbol u)
```

is the same as

```
(DEFINE symbol u)
```

However, the definition

```
(DEFTERM (symbol (X_1 τ_1) ... (X_k τ_k))
  u)
```

is an abbreviation for the definition

```
(DEFINE symbol
  (LAMBDA ((X_1 τ_1) ... (X_k τ_k))
    u))
```

6.8 Summary

The external Ontic language has now been entirely defined; all of the language constructs that appear as primitives in the proof given in the appendix have been described in this chapter. A procedure has been given for translating expressions in the external language into an internal language where there is a one to one correspondence between the nodes in the graph generated by the Ontic compiler and expressions in the internal language. The structure of the internal language has been discussed in detail, including the notion of free variables and a procedure for performing variable substitutions on internal expressions. The next section shows how a set Σ of internal Ontic expressions can be converted to an Ontic graph $G(\Sigma)$. Ontic graphs are simpler than Ontic expressions; while there are twenty three kinds of Ontic expressions, the Ontic graphs defined in chapter 5 have only five kinds of nodes and nine kinds of links.

Chapter 7

The Ontic Compiler

The Ontic system compiles a set Σ of Ontic expressions into an Ontic graph $G(\Sigma)$. The graph structure is much simpler than the Ontic language. The node and link types of Ontic graphs do not provide the distinguished primitive types `THING`, `SET`, `RULE` or `SYMBOL`. Ontic graphs make no distinction between syntactically small and syntactically large types. The node and link types of Ontic graphs do not provide set construction operations or definite descriptions. Ontic graphs have no explicit provisions for defining new functions or type generators or for reify functions as terms. However, in spite of the relative simplicity of Ontic graphs, it is possible to compile internal Ontic expressions into Ontic graphs in a way that implements all the features of the Ontic language.

7.1 An Overview of Compilation

The Ontic compiler takes a set Σ of internal Ontic expressions and generates an Ontic graph $G(\Sigma)$. Each node in the graph $G(\Sigma)$ corresponds to some particular expression in the internal Ontic language, although the expression represented by a node in $G(\Sigma)$ need not be a member of Σ. The notation $C(\Sigma)$ will be used to denote the set of expressions that correspond to the nodes in $G(\Sigma)$. In order to precisely define the set $C(\Sigma)$ each internal Ontic expression e will be associated with a set $Aux(e)$ of internal Ontic expressions called the *auxiliary expressions* for e. The function Aux is defined on a case by case bases in later sections. The set $C(\Sigma)$ is defined relative to the mapping Aux as follows:

Definition: The *auxiliary closure* $C(\Sigma)$ of a set of expressions Σ is the least set of expressions such that

- If an extensional application (op arg_1 arg_2 $\ldots$ arg_k) is in $C(\Sigma)$ then op and each arg_i are in $C(\Sigma)$.
- If a λ-expression (LAMBDA ($x_1^{\tau_1}$ $x_2^{\tau_2}$ $\ldots$ $x_k^{\tau_k}$) $body$) is in $C(\Sigma)$ then $body$ and each $x_i^{\tau_i}$ is in $C(\Sigma)$.
- If a variable x^τ is in $C(\Sigma)$ then τ is in $C(\Sigma)$.
- If e is in $C(\Sigma)$ then $C(\Sigma)$ contains $Aux(e)$.
- Let σ be a λ-type of the form (LAMBDA (x^τ) $\Phi(x^\tau)$) and let y^τ be a variable of type τ. If both σ and y^τ are in $C(\Sigma)$ then $C(\Sigma)$ also contains the formula

  (IFF (IS y^τ σ) $\Phi(y^\tau)$)

  where $\Phi(y^\tau)$ is the result of replacing all free occurrences of x^τ in Φ with y^τ as discussed in chapter 6.

There is a direct one-to-one correspondence between the expressions in $C(\Sigma)$ and the nodes in the Ontic graph $G(\Sigma)$; If e is in $C(\Sigma)$ then the node represented by e is written as n_e. Recall that the nodes in an Ontic graph come in five types: formula nodes, quotation nodes, variable nodes, type nodes, and unclassified nodes. The nodes in the Ontic graph $G(\Sigma)$ that correspond to Ontic formulas, quotation expressions, Ontic variables, and types expressions, are classified in the obvious way. The nodes corresponding to all other expressions are unclassified. Note that if an extensional application (op arg_1 arg_2 $\ldots$ arg_k) is in $C(\Sigma)$ then $C(\Sigma)$ also contains the operator op. This implies that $C(\Sigma)$ contains "expressions" such as `IMPLIES` and `EXISTS-SOME` which are not technically Ontic expressions. Thus the graph $G(\Sigma)$ contains unclassified nodes that correspond to operators such as `IMPLIES` and `EXISTS-SOME`.

Just as the set $C(\Sigma)$ is defined relative to an auxiliary mapping Aux, the links in the graph $G(\Sigma)$ are defined relative to a meaning postulate mapping M. More specifically each expression e in the internal Ontic language is associated with a set $M(e)$ of *meaning postulates* where each meaning postulate in $M(e)$ is a clause link

$$\Psi_1 \vee \Psi_2 \vee \ldots \Psi_k$$

where each Ψ_i is a literal involving a node n_s where s is either the expression e, a subexpression of e or a member of $Aux(e)$. The mapping M which assigns every expression a set of meaning postulates is defined on a case by case basis in later sections. Recall that Ontic graphs have nine kinds of links: clause links, equality links, subexpression links, free variable links, type declaration links, type formula links, subtype links, existence links, and closure links. The complete Ontic graph $G(\Sigma)$ is defined relative to the meaning postulate map M as follows:

- The nodes of $G(\Sigma)$ consist of all nodes of the form n_e where e is an expression in $C(\Sigma)$.
- The clauses in $G(\Sigma)$ are given as follows:
  - $G(\Sigma)$ includes all clauses in $M(e)$ for e in $C(\Sigma)$.
  - If σ is the λ-type (`LAMBDA` (x^τ) $\Phi(x^\tau)$) and y^τ is a variable of type τ and both σ and y^τ are in $C(\Sigma)$ then $G(\Sigma)$ includes the clause

    $$\neg n_{(\texttt{EXISTS-SOME}\ \tau)} \vee n_{(\texttt{IFF}\ (\texttt{IS}\ y^\tau\ \sigma)\ \Phi(y^\tau))}$$

    where $\Phi(y^\tau)$ is the result of replacing all free occurrences of x^τ in Φ with y^τ as discussed in chapter 6. The significance of such clauses is discussed below.
- The equality links in $G(\Sigma)$ consist of
  - All links of the form

    $$n_{(\text{IS}\ t_1\ (\text{EQUAL-TO}\ t_2))} \Leftrightarrow n_{t_1} = n_{t_2}$$

where the formula (IS t_1 (EQUAL-TO t_2)) is in $C(\Sigma)$.

– All links of the form

$$n_{(\texttt{IFF}\ p\ q)} \Leftrightarrow n_p = n_q$$

where the formula (IFF p q) is in $C(\Sigma)$.

– All links of the form

$$n_{(=\ e_1\ e_2))} \Leftrightarrow n_{e_1} = n_{e_2}$$

where the formula (= e_1 e_2)) is in $C(\Sigma)$.

- The subexpression links in $G(\Sigma)$ consist of all links of the form

$$(n_{op}\, n_{arg_1}\, n_{arg_2} \ldots n_{arg_k}) = n_{(op\ arg_1\ arg_2\ \ldots\ arg_k)}$$

where the extensional application $(op\ arg_1\ arg_2\ \ldots\ arg_k)$ is in $C(\Sigma)$.

- The free variable links in $G(\Sigma)$ consist of all links of the form

$$n_{x^\tau} \ll n_e$$

where e is an expression in $C(\Sigma)$ such that x^τ appears free in e.

- The type declaration links in $G(\Sigma)$ consist of all links of the form

$$n_{x^\tau} : n_\tau$$

where x^τ is in $C(\Sigma)$.

- The type formula links in $G(\Sigma)$ consist of all links of the form

$$n_{(\texttt{IS}\ u\ \tau)} \Leftrightarrow n_u : n_\tau$$

where the formula (IS u τ) is member of $C(\Sigma)$.

- The subtype links in $G(\Sigma)$ consist of all links of the form

$$n_{(\texttt{IS-EVERY}\ \sigma\ \tau)} \Leftrightarrow n_\sigma \prec n_\tau$$

where the formula (IS-EVERY σ τ) is a member of $C(\Sigma)$.

- The existence formula links in $G(\Sigma)$ consist of all links of the form

$$n_{\texttt{(EXISTS-SOME } \tau\texttt{)}} \Leftrightarrow \exists n_\tau$$

  where the formula (EXISTS-SOME τ) is a member of $C(\Sigma)$.

- The closure links in $G(\Sigma)$ consist of all links of the form

$$\lambda n_{y^\tau} . n_{\Phi(y^\tau)} \;=\; n_{\texttt{(LAMBDA (}x^\tau\texttt{) }\Phi(x^\tau)\texttt{)}}$$

  where (LAMBDA (x^τ) $\Phi(x^\tau)$) is a λ-type in $C(\Sigma)$, y^τ is a variable of type τ in $C(\Sigma)$ such that y^τ does not appear free in (LAMBDA (x^τ) $\Phi(x^\tau)$) and $\Phi(y^\tau)$ is the result of replacing all free occurrences of x^τ in Φ by y^τ.

The complete specification of the set $C(\Sigma)$ and the graph $G(\Sigma)$ depends on a specification of the mappings *Aux* and *M* which give the Auxiliary expressions and the meaning postulates respectively that are associated with any given expression. The mappings *Aux* and *M* are defined on a case by case basis in the following sections. The significance of each meaning postulate is also discussed.

7.2 λ-Types and Variables

λ-types and variables are of central importance in the Ontic system; all quantification involves the interaction of λ-types and variables. The graph $G(\Sigma)$ contains meaning postulates for individual λ-types, meaning postulates for individual variables, and clauses which are generated by a combination of a λ-type and a variable.

The meaning postulates for individual λ-types and variables are fairly simple. If σ is the λ-type (LAMBDA (x^τ) Φ) then σ is a subtype of τ; every instance of σ is an instance of τ. Thus σ has the auxiliary expression

(IS-EVERY σ τ)

The meaning postulates for σ include a clause that contains only the node for the above subtype expression. This clause ensures that the node for the subtype expression is true in any consistent normalized labeling. The auxiliary expressions for the λ-type σ also include (EXISTS-SOME σ) and (EXISTS-SOME τ) and the meaning postulates for σ include the clause

$$\neg n_{(\texttt{EXISTS-SOME}\ \sigma)} \vee n_{(\texttt{EXISTS-SOME}\ \tau)}$$

This clause states that if there exists an instance of σ then there exists an instance of τ. While this last clause is semantically redundant it forces certain inferences which would not be performed otherwise.

There are also meaning postulates for λ-types which allow congruence closure to operate on λ-types. In fact every λ-expression in the Ontic language has an auxiliary combinator expression. More specifically there is a function *Comb-Trans* which converts λ-expressions into combinator form. For any λ-expression e the combinator expression *Comb-Trans*(e) is an auxiliary expression of e. The meaning postulates for e include the clause containing the single node

$$n_{(=\ e\ \mathit{Comb\text{-}Trans}(e))}$$

This clause ensures that n_e is equivalent to $n_{\mathit{Comb\text{-}Trans}(e)}$.

Combinator expressions allow congruence closure to act on λ-expressions. For example consider the two lambda types

(LAMBDA (x^τ) (IS u (RELATED-TO x^τ)))

(LAMBDA (x^τ) (IS w (RELATED-TO x^τ)))

where u and w are terms which do not contain x^τ as a free variable. If both of the above expressions are in $C(\Sigma)$ and if a particular labeling $\mathcal{L}$ of $G(\Sigma)$ makes the node for u equivalent to the node for w, then $\mathcal{L}$ will equate the nodes for these two λ-expressions. Note that if x^τ appears free in either u or v then this congruence inference is not valid.

Combinator conversion algorithms are discussed in [Turner 79] and will not be described here. Combinator expressions are used solely for congruence closure on λ-expressions; combinator expressions have no auxiliary expressions or meaning postulates. However combinator expressions are extensional applications and therefore generate subexpression links.

Each individual variable also has some auxiliary expressions and a meaning postulate. If x^τ is a variable of type τ then the auxiliary expressions for x^τ consist of the formulas (EXISTS-SOME τ) and (IS x^τ τ). The meaning postulates for x^τ consists of the the single clause

$$\neg n_{\texttt{(EXISTS-SOME } \tau\texttt{)}} \vee n_{\texttt{(IS } x^\tau\ \tau\texttt{)}}$$

This clause says that if there exists any instance of the type τ then x^τ is an instance of τ. This clause ensures that in any consistent normalized labeling, if the node $n_{\texttt{(EXISTS-SOME } \tau\texttt{)}}$ is labeled true then the type node n_τ is an established type node for the variable node n_{x^τ}.

In addition to the auxiliary expressions and meaning postulates for individual λ-types and variables there are expressions and clauses which are generated by a combination of a λ-type and a variable. Suppose that $C(\Sigma)$ includes both a λ-type (LAMBDA (x^τ) $\Phi(x^\tau)$) a variable y^τ of type τ. Let σ be the lambda type (LAMBDA (x^τ) $\Phi(x^\tau)$). Under these conditions $C(\Sigma)$ includes the formulas

$$\texttt{(EXISTS-SOME } \tau\texttt{)}$$

and

$$\texttt{(IFF (IS } y^\tau\ \sigma\texttt{)}\ \Phi(y^\tau)\texttt{)}$$

where $\Phi(y^\tau)$ is the result of substituting y^τ for all free occurrences of x^τ in Φ as discussed in chapter 6. Furthermore the graph $G(\Sigma)$ includes the clause

$$\neg n_{\texttt{(EXISTS-SOME } \tau\texttt{)}} \vee n_{\texttt{(IFF (IS } y^\tau\ \sigma\texttt{)}\ \Phi(y^\tau)\texttt{)}}$$

This clause says that, as long as there exist instances of the type τ, the formula (IS y^τ σ) is equivalent to $\Phi(y^\tau)$. This equivalence can be viewed as a definition of the type σ.[1] More specifically, suppose that the system is focusing on a term u of type τ and the system is to determine if u is of type σ (which is a more specific type than τ). The above equivalence says that u is of type σ just in case the formula $\Phi(u)$ is true. For simplicity suppose that the formulas (IS u σ) and $\Phi(u)$ have been compiled, i.e. that they are both in $C(\Sigma)$. Since u is of type τ the system can generate the binding $y^\tau \mapsto u$. But if y^τ and u are equivalent then by congruence closure the formula (IS y^τ

[1]Actually the equivalence provides only a partial definition; it does not state the additional condition that σ is a subtype of τ.

σ) is equivalent to the formula (IS u σ) and $\Phi(y^\tau)$ is equivalent to $\Phi(u)$. [2] Thus the binding

$$y^\tau \mapsto u$$

together with the truth of the equivalence

$$\texttt{(IFF (IS } y^\tau\ \sigma\texttt{) } \Phi\texttt{)}$$

causes the formula (IS u σ) to be equivalent to $\Phi(u)$.

In the presence of the binding $y^\tau \mapsto u$ the equivalence

$$\texttt{(IFF (IS } y^\tau\ \sigma\texttt{) } \Phi(y^\tau)\texttt{)}$$

can be used to determine if u is of type σ even when the formulas (IS u σ) and $\Phi(u)$ have not been compiled, i.e. are not in $C(\Sigma)$. In the presence of the binding $y^\tau \mapsto u$ the semantic modulation inference mechanisms ensure that the nodes n_{y^τ} and n_u are virtually indistinguishable and that the formulas (IS y^τ σ) and $\Phi(y^\tau)$ behave exactly as the formulas (IS u σ) and $\Phi(u)$ would behave if they were compiled.

In general there can be more than one variable of type τ. The definition of σ is stated in terms of each variable of type τ. This helps to ensure the homogeneity of the generated graph: different variables nodes with the same type are identical in that they carry exactly the same information.

7.3 Meaning Postulates with Quantifiers

If the lemma library contains a formula of the form (FORALL (x^τ) $\Phi(x^\tau)$) then for each variable y^τ of type τ the compilation process should generate the formula $\Phi(y^\tau)$ which is the result of replacing all free occurrences of x^τ in Φ with y^τ. In this way the compiler should ensure that all information known to hold of the type τ is copied for each variable of type τ and any binding of the form $y^\tau \mapsto u$ causes the term u to inherit information known to hold of

[2]Because combinator expressions ensure that congruence closure is operates on λ-expressions the binding $y^\tau \mapsto u$ causes $\Phi(y^\tau)$ to be equivalent to $\Phi(u)$ even in the case where y^τ appears free inside λ-expressions contained in $\Phi(y^\tau)$.

the type τ. The formula (FORALL (x^τ) $\Phi(x^\tau)$) is actually an abbreviation for

```
(NOT
  (EXISTS-SOME
    (LAMBDA (x^τ)
      (NOT Φ(x^τ)))))
```

If the above formula is true the system should ensure that the formula $\Phi(y^\tau)$ is true. This is done via a meaning postulate for type assertion formulas. More specifically the meaning postulates for a type assertion formula (IS u σ) consist of the single clause

$$\neg n_{(\mathtt{IS}\ u\ \sigma)} \vee n_{(\mathtt{EXISTS\text{-}SOME}\ \sigma)}$$

This clause states that if u is an instance of type τ then there exist instances of type τ. The clause also states the equally important condition that if there are no instances of σ then u is not an instance of σ. In particular, if there are no instances of σ then y^τ is not an instance of σ. Given the above meaning postulate for type assertion formulas and the meaning postulates discussed in the previous section, one can prove an important lemma about quantification in the Ontic system.

Lemma: If the formula (FORALL (x^τ) $\Phi(x^\tau)$) is in $C(\Sigma)$ and y^τ is a variable of type τ in $C(\Sigma)$ then $C(\Sigma)$ also includes $\Phi(y^\tau)$. Furthermore if $\mathcal{L}$ is a consistent normalized labeling of $G(\Sigma)$ such that $\mathcal{L}$ assigns the label **true** to the nodes for (EXISTS-SOME τ) and (FORALL (x^τ) $\Phi(x^\tau)$) then $\mathcal{L}$ also assigns the label **true** to the node for $\Phi(y^\tau)$.

Proof: $C(\Sigma)$ includes the formula

```
(NOT
  (EXISTS-SOME
    (LAMBDA (x^τ)
      (NOT Φ(x^τ)))))
```

Let σ be the λ-type

$$\texttt{(LAMBDA (}x^\tau\texttt{) (NOT }\Phi(x^\tau)\texttt{))}$$

Since both σ and y^τ are in $C(\Sigma)$ the equivalence

$$\texttt{(IFF (IS }y^\tau\ \sigma\texttt{) (NOT }\Phi(y^\tau)\texttt{))}$$

must also be in $C(\Sigma)$ and thus the formula $\Phi(y^\tau)$ is in $C(\Sigma)$. Furthermore the formula (IS y^τ σ) is in $C(\Sigma)$ and so $G(\Sigma)$ includes the clause

$$\neg n_{\texttt{(IS }y^\tau\ \sigma\texttt{)}} \vee n_{\texttt{(EXISTS-SOME }\sigma\texttt{)}}$$

Now if $\mathcal{L}$ assigns the above universal formula the label **true** it must assign the node for (EXISTS-SOME σ) the label **false**. Thus the node for (IS y^τ σ) must also be assigned **false**. Furthermore $G(\Sigma)$ contains the clause

$$\neg n_{\texttt{(EXISTS-SOME }\tau\texttt{)}} \vee n_{\texttt{(IFF (IS }y^\tau\ \sigma\texttt{) (NOT }\Phi(y^\tau)\texttt{))}}$$

Since $\mathcal{L}$ assigns the the node for (EXISTS-SOME τ) the label **true**, $\mathcal{L}$ must also assign the label **true** to the node for

$$\texttt{(IFF (IS }y^\tau\ \sigma\texttt{) (NOT }\Phi(y^\tau)\texttt{))}$$

But since the node for (IS y^τ σ) is assigned **false**, the node for (NOT $\Phi(y^\tau)$) must also be assigned **false**. But this implies that the node for $\Phi(y^\tau)$ is assigned **true**.

The expression

$$\texttt{(FORALL (}x_1^{\tau_1}\ x_2^{\tau_2}\ \ldots\ x_k^{\tau_k}\texttt{) }\Phi\texttt{)}$$

is an abbreviation for nested universal quantification as described in chapter 6. The above lemma for a single universal quantifier immediately generalizes to multiple universal quantification; a universal formula which quantifies over several variables will be instantiated with all variables of the appropriate type.

Several kinds of Ontic expressions have meaning postulates that involve quantification. For example let f be a λ-function or non-primitive type generator of the form

```
(LAMBDA ($x_1^{\tau_1}$ $x_2^{\tau_2}$ ... $x_k^{\tau_k}$) $body$)
```

The λ-expression f has the single auxiliary expression

```
(FORALL ($x_1^{\tau_1}$ $x_2^{\tau_2}$ ... $x_k^{\tau_k}$)
  (= ($f$ $x_1^{\tau_1}$ $x_2^{\tau_2}$ ... $x_k^{\tau_k}$)
     $body$))
```

The meaning postulates for f consist of a single singleton clause which states that the above formula is true. This formula serves as the definition for the operator f. In order for this definition to be invoked on an expression (f u_1 u_2 ... u_k) variables of the appropriate type must be bound to the arguments u_1 u_2 ... u_k. Once this has been done the application (f u_1 u_2 ... u_k) will be equivalent to an appropriate substitution instance of *body*. However in order to get variables of the proper type bound to the arguments one must focus on the arguments. Thus in order to invoke the definition of an operator f in an application (f u_1 u_2 ... u_k) one must focus on all the arguments u_i.

Semantically, the type generator EITHER could be defined as

```
(LAMBDA ($x^{\texttt{THING}}$ $y^{\texttt{THING}}$)
  (LAMBDA ($z^{\texttt{THING}}$)
    (OR (= $z^{\texttt{THING}}$ $x^{\texttt{THING}}$)
        (= $z^{\texttt{THING}}$ $y^{\texttt{THING}}$))))
```

Note however that if EITHER where simply an abbreviation for the above expression then types of the form (EITHER u w) would not be syntactically small. Furthermore, and more seriously, invoking the above definition in a particular application requires focusing on the arguments to the operator EITHER. The usefulness of the operator EITHER is greatly improved by making EITHER a primitive type generator and constructing meaning postulates for every type of the form (EITHER u w).

Let σ be a type expression of the form (EITHER u w). The type σ has the auxiliary expressions

```
(IS $u$ $\sigma$)

(IS $w$ $\sigma$)

(FORALL ($x^\sigma$)
  (OR (= $x^\sigma$ $u$)
      (= $x^\sigma$ $w$)))
```

The meaning postulates for σ consist of three singleton clauses which state that each of the above formulas is true.

Let σ be a type expression of the form (OR-TYPE τ_1 τ_2). The type σ has the auxiliary expressions

```
(IS-EVERY $\tau_1$ $\sigma$)

(IS-EVERY $\tau_2$ $\sigma$)

(FORALL ($x^\sigma$)
  (OR (IS $x^\sigma$ $\tau_1$)
      (IS $x^\sigma$ $\tau_2$)))
```

The meaning postulates for σ consist of three singleton clauses which state that each of the above formulas is true.

Let f be a λ-function of the form

(LAMBDA ($x_1^{\tau_1}$ $x_2^{\tau_2}$ ... $x_k^{\tau_k}$) *body*)

and let σ be the type expression (RANGE-TYPE f). The type expression σ has two auxiliary formulas:

```
(FORALL ($x_1^{\tau_1}$ $x_2^{\tau_2}$ ... $x_k^{\tau_k}$)
  (is body $\sigma$))

(FORALL ($y^\sigma$)
  (EXISTS ($x_1^{\tau_1}$ $x_2^{\tau_2}$ ... $x_k^{\tau_k}$)
    (= $y^\sigma$ body)))
```

The meaning postulates for σ consist of two singleton clauses which assert that the above formulas are true. These formulas constitute a definition of the type σ.

Let u be the term (THE τ) where τ is any type expression. The term u has the auxiliary expressions

$$\texttt{(EXACTLY-ONE } \tau\texttt{)}$$

$$\texttt{(IS } u \ \tau\texttt{)}$$

$$\texttt{(FORALL (}x^\tau\texttt{) (= } x^\tau \ u\texttt{))}$$

where these expressions abbreviate internal Ontic expressions as described in chapter 6. The term u has meaning postulates

$$\neg n_{\texttt{(EXACTLY-ONE } \tau\texttt{)}} \vee n_{\texttt{(IS } u \ \tau\texttt{)}}$$

$$\neg n_{\texttt{(EXACTLY-ONE } \tau\texttt{)}} \vee n_{\texttt{(FORALL (}x^\tau\texttt{) (= } x^\tau \texttt{ u))}}$$

These meaning postulate states that if there is exactly one object of type τ then u is of type τ and everything of type τ is equal to u.

7.4 Reification Expressions

The Ontic system can only focus on terms; in order to focus on types, functions, or type generators the system must first coerce these objects to terms. The process of coercing a higher order object to a first order term is called *reification.* The Ontic language has two reification operators: THE-SET-OF-ALL which coerces a type to a set, and THE-RULE which coerces a function of one argument to a set of pairs. Both of these reification operators can only be applied to syntactically small objects. e.g. one can not construct a set of all sets.

Let s be an expression of the form (THE-SET-OF-ALL τ) where τ is a syntactically small type expression. The auxiliary expressions for s consist of the formulas (IS s SET) and (= τ (MEMBER-OF s)) and the meaning postulates for s consist of two singleton clauses which assert that these two formulas are true.

Now consider the other reification operator, THE-RULE. Let f be the λ-function (LAMBDA (x^τ) u) where τ is a syntactically small type expression and let r be the term (THE-RULE f). The term r has three auxiliary expressions:

(IS r RULE)

(= (THE-FUNCTION r) f)

(= (RULE-DOMAIN r) (THE-SET-OF-ALL τ))

The meaning postulates for r consist of three singleton clauses which state that each of the the auxiliary formulas must be true.

The meaning postulates for expressions of the form (THE-RULE f) do not force this expression to denote a set of pairs; the meaning postulates do not force any particular implementation of a rule in terms of sets. However the meaning postulates are sufficient to recover all of the information in the rule; if r is the expression (THE-RULE f) then one can construct the set of pairs corresponding to r from the function (THE-FUNCTION r) and the set (RULE-DOMAIN r).

7.5 Miscellaneous Meaning Postulates

Let u be the term (IF Φ w_1 w_2). The auxiliary expressions for u consist of the equalities (= u w_1) and (= u w_2). The meaning postulates for u consist of the following two clauses

$$\neg n_\Phi \vee n_{(= \ u \ w_1)}$$

$$n_\Phi \vee n_{(= \ u \ w_2)}$$

These two clauses state that if Φ is true then u equals w_1 and if Φ is false then u equals w_2.

Let u be the quotation (QUOTE *symbol*). The node n_u is a quotation node and any labeling which equates distinct quotation nodes is taken to be explicitly contradictory. The auxiliary expressions for u consist of the single formula (IS u SYMBOL) and the meaning postulates for u consist of a singleton clause which states that this formula is true.

The meaning postulates for expressions of the form (THE-SET-OF-ALL τ) and (THE-RULE f) provide meanings for the types SET and RULE; every reified predicate is a set and every reified function is a rule. Furthermore the type SYMBOL is defined by the meaning postulates for quotations. The type THING is the universal type and the type expression THING has the following auxiliary expressions

```
(IS-EVERY SET THING)

(FORALL (x^SET)
  (IS-EVERY (MEMBER-OF x^SET) THING))

(IS-EVERY RULE THING)

(IS-EVERY SYMBOL THING)
```

The meaning postulates for the type THING consist of three singleton clauses which state that each of the above formulas is true.

The type generator EQUAL-TO has the following auxiliary expression.

```
(= EQUAL-TO
   (LAMBDA (x^THING)
     (EITHER x^THING x^THING)))
```

The meaning postulates for EQUAL-TO consist of a single clause which states that the above formula is true. EQUAL-TO has been listed as a primitive type generator because formulas of the form

(IS u (EQUAL-TO w))

generate equality links; these equality links would not be generated if EQUAL-TO was defined rather than taken as a primitive.

The type generator SUBSET-OF has the following auxiliary expression.

```
(= SUBSET-OF
   (LAMBDA (x^SET)
     (LAMBDA (y^SET)
       (IS-EVERY (MEMBER-OF y^SET)
                 (MEMBER-OF x^SET)))))
```

The meaning postulates for SUBSET-OF consist of a single clause which states that the above equivalence is true. SUBSET-OF has been listed as a primitive type generator because it is syntactically small; the equivalent λ-expression given above is not syntactically small.

The type generator RULE-BETWEEN has the following auxiliary expression.

```
(= RULE-BETWEEN
   (LAMBDA ($x^{\mathtt{SET}}$ $y^{\mathtt{SET}}$)
     (LAMBDA ($z^{\mathtt{RULE}}$)
       (AND (= (RULE-DOMAIN $z^{\mathtt{RULE}}$)
               $x^{\mathtt{SET}}$)
            (FORALL ($w^{\mathtt{(MEMBER\text{-}OF}\ x^{\mathtt{SET}})}$)
              (IS ((THE-FUNCTION $z^{\mathtt{RULE}}$)
                   $w^{\mathtt{(MEMBER\text{-}OF}\ x^{\mathtt{SET}})}$)
                (MEMBER-OF $y^{\mathtt{SET}}$)))))))
```

The meaning postulates for RULE-BETWEEN consist of a single clause which states that the above equivalence is true. RULE-BETWEEN has been listed as a primitive type generator because it is syntactically small; the equivalent λ-expression given above is not syntactically small.

The meaning postulates for Boolean connectives are given in table 4.1 in chapterconst-prop-chap.

7.6 Summary

The Ontic compiler converts a set Σ of expressions in the Ontic Language to an Ontic graph $G(\Sigma)$. There is a one to one correspondence between the nodes in $G(\Sigma)$ and a set $C(\Sigma)$ of Ontic expressions where $C(\Sigma)$ contains Σ as a subset. The compilation process is specified in terms of meaning postulates which are defined on a case by case basis for the various kinds of Ontic expressions.

The compilation process is incremental; if Σ' is an incremental extension of Σ then $G(\Sigma')$ can be constructed as an incremental extension of $G(\Sigma)$.

When a new expression is typed to the top level Ontic interpreter new graph structure is incrementally added to represent that expression. When the system focuses on a term u of type τ it is sometimes necessary to create a new variable of type τ to bind to u. When a new variable is created new graph structure is automatically constructed to represent that variable.

Chapter 8

Some Potential Applications

There are two ways of evaluating the ideas used in the Ontic system. First, one can attempt to evaluate the utility of the ideas in constructing useful systems. Second, one can attempt to evaluate the extent to which Ontic's inference mechanisms provide a plausible model of human mathematical cognition. This chapter addresses the first evaluation technique by presenting a list of potential applications of automated inference systems. The applications on this list represent directions for future research; the limitations of Ontic's object oriented inference techniques in these applications are not currently understood and future research may uncover other inference techniques which make these applications practical.

One potential application for automated inference systems is simply the verification of mathematical arguments; an author could increase his confidence in the correctness of a proof using machine verification. The time required to "debug" the formal representation of proofs in the Ontic system seems to make this application impractical at the current time. However, as the inference power of the system is increased, and the lemma library is made larger, the system may approach the point where machine verification of new mathematics is practical.

Automated inference mechanisms are needed in the construction of interactive knowledge bases. The Ontic system is able to automatically use information from a lemma library. An Ontic system based on a lemma library that contained the contents of a mathematical textbook could answer certain questions about the contents of that book. Such an interactive textbook might be valuable in education. If the system could be made to run with a very large lemma library, a library containing the contents of many textbooks, one could construct an interactive mathematical encyclopedia. An interactive encyclopedia could be used by professional mathematicians to answer questions and verify arguments in domains that were not familiar to the human user.

Automated inference systems might also be useful in constructing interactive documentation systems. A computer operating system, for example, is usually associated with a large amount of documentation. It may be possible to translate this documentation into first order axioms that can serve as a lemma library underlying an inference system. One would then have a device for answering questions about the documented system. The problem of answering questions about engineered devices seems similar to, but possibly more difficult than, the problem of answering questions about the material in a mathematical textbook.

Ontic's object oriented inference mechanism may be useful for program verification. Ontic's type system is similar to the type systems of strongly typed programming languages. With sufficiently expressive types there is no distinction between type checking and verification; any verification problem for a computer program can be phrased as a type-checking problem. Ontic's object-oriented inference mechanisms are organized around types. It would be interesting to explore the application of Ontic's object-oriented inference mechanisms to program verification where verification is viewed as a form of type-checking.

Another possible application for Ontic's object-oriented inference mechanisms is common sense reasoning. In his naive physics manifesto Hayes proposed writing down first order axioms which express common sense knowledge about the physical world [Hayes 85]. One might object to Hayes' proposal on the grounds that first order inference is intractable. It is clear, however, that certain limited inferences can be done quickly. It would be interesting to explore the application of Ontic's inference mechanisms to reasoning about common sense situations. Another objection to Hayes' proposal is that much, if not most, common sense reasoning is heuristic: the conclusions are not strictly implied by the given information. The final section of this chapter suggests a way in which Ontic's object oriented inference mechanisms could be extended to perform certain forms of heuristic reasoning.

8.1 Interactive Knowledge Bases

Ontic's object-oriented inference mechanisms are designed to automatically access a large lemma library. By placing various kinds of information in the knowledge base underlying an Ontic-like system one could construct interactive mathematical textbooks, interactive mathematical encyclopedias, and interactive technical documentation libraries.

Access to information in Ontic's lemma library is controlled via types: the inference mechanism accesses only those portions of the lemma library that concern types which apply to the given focus objects. For example, when reasoning about graphs the system automatically ignores facts about differentiable manifolds. Thus the lemma library could include information about a large number of different subjects and still be used effectively.

There are several ways one could use an interactive mathematical encyclopedia. First, the encyclopedia could be used to answer questions about areas of mathematics that are unfamiliar to the user. Second, the encyclopedia could verify a user's argument. This would be especially useful when the human user is unfamiliar with the subject matter of his own argument. Finally, a mathematician who develops a new concept could ask the system if that concept has already been defined under some other name.

Recognizing user-defined concepts is particularly difficult; there may be a defined concept in the encyclopedia which is "essentially the same" as a user-defined concept but the two definitions are technically different. For example, consider the concept of an equivalence relation. An equivalence relation can be defined as a relation, i.e. a set of pairs, which is symmetric, transitive, and reflexive. Alternatively, an equivalence relation can be defined as a partition of a set into equivalence classes. These two definitions seem to define the same concept and yet the two classes are technically disjoint: a partition is different from a set of pairs. It turns out that one can define a very general notion of *iso-onticity* under which equivalence relations (as sets of pairs) are iso-ontic to partitions [McAllester 83]. There are many other examples of iso-onticities between classes. For example a function f of two arguments defines a Curried function f' such that for for all arguments x and y, the application $f'(x)$ yields a function such that

$$f(x, y) = f'(x)(y)$$

The function f is iso-ontic to its curried version f'. As another example consider a graph. A graph can be defined in two ways: a graph can be defined as a set of nodes together with a set of arcs where each arc is a set of two nodes. Alternatively, a graph could be defined as a set of nodes together with a symmetric anti-reflexive binary relation on those nodes. A relation, i.e. a set of pairs, is different from a set of arcs, i.e. a set of sets. A set of two-elements sets, however, is iso-ontic to a symmetric anti-reflexive binary relation. There are many examples of iso-onticities in mathematics. Ideally an interactive encyclopedia would recognize when a user-defined concept is iso-ontic to a concept that already exists in the encyclopedia.

8.2 Software Verification

Type checking has proved to be a practical way of finding certain errors in computer programs. Currently available type checking systems use a weak vocabulary of types — there is no way to treat an arbitrary predicate as a data type. If the type vocabulary is made richer then stronger "semantic" properties of programs can be expressed as type constraints. In fact, if any predicate on data structures can be expressed as a type then any semantic specification for a computer program can be expressed as type restrictions. For example, if iteration is replaced by recursion then a programmer can provide loop invariants simply by placing type restrictions on the arguments of recursive functions.

If arbitrary predicates on data structures can be expressed as types then type checking requires theorem proving. One might argue that, because theorem proving is intractable, one should not use fully expressive type systems. This criticism carries little weight, however, if one is willing to allow type checking to fail. A failure to type check simply means that the system failed to prove the program correct; it does not mean that the program is wrong. Since Ontic's object-oriented theorem proving mechanisms are guaranteed to terminate quickly, a type checking system based on Ontic's theorem proving mechanisms could also be made to terminate quickly. Programs which fail to type check are classified as "not obviously correct". Since the Ontic's inference mechanisms can automatically use a large lemma library, the power of a type checker based on Ontic could always be increased by adding more lemmas. Such lemmas could either be proved from first principles or simply added as axioms. Adding lemmas should cause more programs to be classified as obviously correct.

Type checking has already been demonstrated to be practical for certain restricted type vocabularies. It seems likely that type checking using more expressive types would be equally practical in the sense that all types which are checked by existing systems could still be checked in the more general setting. A system with fully expressive types could gradually be extended to incorporate more powerful inference techniques under the constraint that type checking terminates quickly.

8.3 Common Sense and Default Reasoning

Hayes has proposed using first order logic as a language for representing common sense knowledge about the physical world [Hayes 85]. One possible objection to first order logic as a representation language is that theorem proving is intractable. It would be interesting to see if Ontic's object oriented theorem proving mechanisms could be used to answer common sense questions about the physical world using a formal fact library.

Another objection to first order logic as a knowledge representation language is that common sense reasoning is often heuristic: heuristic reasoning produces conclusions which are likely, but not necessarily true. This observation has lead to the development of default logics and semantic network formalisms that allow the cancellation of inheritance links [Fahlman 79] [Etherington & Reiter 83]. It seems likely that Ontic's object oriented inference mechanisms could be extended to handle certain kinds of heuristic inference. Ontic's inference mechanisms are organized around types. It seems plausible that heuristic knowledge could also be organized around types. More specifically one could introduce the quantifier FORMOST which is analogous to the quantifier FORALL. One could then write axioms such as the following

```
(FORMOST ((X BIRD)) (IS X FLYING-ANIMAL))
```

One can assign truth values to FORMOST formulas by associating each type with a probability distribution over instances of that type. In general, a formula of the form

(FORMOST ((x τ)) $\Phi(x)$)

is true just in case the fraction of instances of type τ which satisfy $\Phi(x)$ is above some threshold α. If the threshold α is large, say 95%, then a reasoning system might perform heuristic inferences by treating FORMOST the same way it treats FORALL: given that most birds fly, and Tweety is a bird, the system would "deduce" that Tweety flies. The facts that Tweety is a bird and that most birds fly do not imply that Tweety flies, or even that it is likely that tweety flies, whatever that means. People, however, will naturally conclude that Tweety probably flies. Thus heuristic inference is not semantically sound. However, unsound heuristic inference seems to be useful.

The following example indicates that inclusion relationships between types play an important role in human heuristic reasoning. I will use the expression

(ARE-MOST τ σ)

as an abbreviation for

(FORMOST ((x τ)) (IS x σ))

The following "inheritance network" concerning molluscs is adapted from [Etherington & Reiter 83].

```
(ARE-MOST MOLLUSC SHELL-BEARER)
```

```
(IS-EVERY CEPHALOPOD MOLLUSC)
(ARE-MOST CEPHALOPOD (NOT-TYPE SHELL-BEARER))
(IS-EVERY NAUTILUS CEPHALOPOD)
(IS-EVERY NAUTILUS SHELL-BEARER)
```

Given the above information together with the statement that Squirmy is a mollusc one would naturally conclude that Squirmy is probably a shell-bearer. If one is then told that Squirmy is a cephalopod one would conclude that Squirmy is probably not a shell-bearer. Note that in this second case there is a conflict between two FORMOST assertions that apply to Squirmy: most molluscs have shells but most cephalopods do not have shells. In this case the known inclusion relationship between the types CEPHALOPOD and MOLLUSC seems to resolve the conflict. Finally, if one is told that Squirmy is a nautilus one would in fact know, according to the above information, that Squirmy is a shell bearer.

If a reasoning system treats FORMOST assertions in the same way that it treats FORALL assertions it will perform unsound inferences. In particular, each universal instantiation of a FORMOST assertion is unsound. If some unsound FORMOST instantiation produces a conclusion which conflicts with known information then that unsound instantiation inference should be retracted. Furthermore, if two unsound instantiations of FORMOST assertions are mutually contradictory, and there is an inclusion relation between the types quantified over in the two FORMOST assertions, then the FORMOST assertion with the more specific type should dominate and the unsound instantiation of the other FORMOST assertion should be retracted. More research is needed to determine if these guidelines lead to an efficient and useful heuristic reasoning system.

Chapter 9

A Summary of Ontic

The Ontic system has the following features:

- The Ontic formal language is organized around a rich vocabulary of types.
  - There are many different ways of constructing type expressions. Any predicate of one argument is a type. Type generators can be applied to arguments to yield types. There are special constructs such as `WRITABLE-AS` for constructing types from terms. Types can be combined with Boolean combinators to yield other types.
  - There are many different ways of using types. Types are used as predicates in formulas of the form `(IS` x τ`)`. Types restrict the range of quantifiers. A type can be used to construct a term via the operator `THE`. A type can be used to construct a set via the operator `THE-SET-OF-ALL`. Types can be directly related via the combinator `IS-EVERY`.
  - Types play a central role in Ontic's object-oriented inference mechanisms.
- Most of the axioms of Zermelo Fraenkel set theory are incorporated into the syntactic definition of a small type expression and a small function

expression; type and function expressions which are syntactically small can be *reified* via the operators THE-SET-OF-ALL and THE-RULE respectively.

- Many modern theorem provers are based on some kind of backward chaining rewrite mechanism guided by a notion of simplification. Ontic is based on a forward chaining mechanism guided by a notion of focus. Ontic's forward chaining inference process is restricted to formulas which are about a given set of *focus objects.*

- Ontic automatically finds and applies information from a large lemma library. The Ontic system *classifies* each focus object by findings types that are true of that object. If a focus object x is classified as being an instance of type τ then the system automatically applies knowledge about the type τ to the focus object x.

- Ontic's inference mechanisms are implemented as labeling operations on a graph structure. The graph structure represents a compiled version of the lemma library and is analogous to a semantic network. The graph labeling process implements a virtual copy mechanism whereby a focus object becomes a virtual copy of a generic individual.

- Ontic performs automatic universal generalization as part of its forward chaining inference process. In universal generalization the generic individuals in Ontic's graph structure are analogous to the Skolem constants introduced in a resolution theorem prover by a universally quantified goal formula. At other times the same generic individuals are used as universal variables which get instantiated with (bound to) focus objects. At still other times generic individuals act as Skolem constants introduced by existential premises. The types associated with generic individuals are central to the automatic universal generalization mechanism: the types determine the range of applicability of the derived universal statement.

It is not clear which of the above features are most responsible for the power of the Ontic system. Some features are orthogonal to others. For example, the reification operations THE-SET-OF-ALL and THE-RULE could be removed from the system: no other feature of the system depends on the reification operators. Similarly, the universal generalization mechanism could be removed without effecting any other mechanism. Other features are less modular.

It would probably be possible to find some object-oriented forward chaining inference mechanism that does not use graph-labeling. Such a mechanism would be restricted so that variables are only instantiated with focus objects. Implementing congruence closure and automatic universal generalization, however, might be difficult in a system that was based on formula manipulation rather than graph labeling.

On the other hand, one can image a graph-labeling inference mechanism not guided by focus objects. In such a system bindings for generic individuals would be generated in some other way. Early versions of the Ontic system used graph-labeling inference mechanisms, including a virtual copy mechanism based on binding generic individuals, but did not use focus objects to guide the binding process. These early versions of the system did not perform well. User-specified focus objects seem to be central to the operation of Ontic.

All of the features of the Ontic system utilize types. In addition to providing concise and natural formulas, types are central to accessing information in the lemma library, binding generic individuals, automatic universal generalization, and reification. It is difficult to imagine any version of the Ontic system not organized around types.

Knowledge representation and automated inference may ultimately have a profound effect on our society. Interactive encyclopedias may some day be able to answer questions about a large fraction of human knowledge. Such encyclopedias would make all current forms of publication obsolete. Thus, however the future judges the ideas presented here, I hope that research in inference and knowledge representation will continue.

Appendix A

The Stone Representation Theorem

This appendix contains a complete listing of a mathematical development which starts with a foundational system equivalent to ZFC set theory and ends with a proof of the Stone representation theorem. The listing contains three types of information: the definitions of all non-primitive terms used in the development, the lemmas proven, and the machine verified proof of each lemma. Definitions appear centered on the page while lemmas are shown in a left hand column next to their proofs which appear in a right hand column. The "proofs" are actually recorded histories of interactions with the Ontic interpreter.

The listing is cumulative; at each point in the listing the system has access to all definitions and lemmas presented earlier in the listing. At any given point in the listing the definitions and lemmas given prior to that point are stored in a fact library that is accessed automatically by the system. At the end of the listing the accumulated fact library contains 509 facts: 154 definitions and 355 lemmas.

The listing is divided into sections each of which begins with an English description of the contents of that section. The first four sections introduce basic notions from set theory such as singleton and doubleton sets, unions

Section	Number of Facts
Fundamentals	95
Pairs, Rules and Structures	39
Maps	75
Relations, Choice and Relation Structures	45
Partial Orders and Zorn's Lemma	68
Lattices	48
Bounded, Distributive, and Complemented Lattices	40
Sublattices	35
Lattice Morphisms	25
Filters and Ultrafilters	18
The Stone Representation Theorem	21
Total	509

Table A.1: The number of facts in each section

and intersections, pairs, relations, structures, and functions. These first four sections contain 254 facts; roughly half the total. The remaining sections develop facts about partial orders, lattices, filters in lattices, and the Stone representation theorem. Table A.1 shows the number of facts in each section.

A.1 Fundamentals

This section contains basic facts about sets. More specifically this section contains:

- A proof of the existence and uniqueness of the empty set.
- Facts about inserting objects into sets.
- Facts about singleton and doubleton sets.
- A version of Russel's paradox that proves that for every set there exists something not in that set.
- Facts about families of sets.
- Facts about unions and intersections of sets.
- Facts about removing objects from sets.
- Facts about power sets.

We begin with the empty set:

```
(DEFTYPE EMPTY-SET
  (LAMBDA ((S SET))
    (NOT
      (EXISTS-SOME
        (MEMBER-OF S)))))
```

```
(LEMMA (EXISTS-SOME EMPTY-SET))         (IN-CONTEXT
                                            ((PUSH-GOAL (EXISTS-SOME EMPTY-SET))
                                             (LET-BE S SET)
                                             (LET-BE S2
                                               (THE-SET-OF-ALL (X (MEMBER-OF S))
                                                       (NOT (= X X)))))
                                          (NOTE-GOAL))

(LEMMA (AT-MOST-ONE EMPTY-SET))         (IN-CONTEXT
                                            ((LET-BE S1 EMPTY-SET)
                                             (LET-BE S2 EMPTY-SET))
                                          (NOTE (AT-MOST-ONE EMPTY-SET)))
```

```
(DEFTERM THE-EMPTY-SET
  (THE EMPTY-SET))
```

```
(LEMMA
  (NOT
    (EXISTS-SOME
      (MEMBER-OF THE-EMPTY-SET))))
```

```
(IN-CONTEXT
    ((LET-BE S THE-EMPTY-SET))
  (NOTE
    (NOT
     (EXISTS-SOME
       (MEMBER-OF THE-EMPTY-SET)))))
```

```
(DEFTERM (INSERT (X THING) (S SET))
  (THE-SET-OF-ALL
    (OR-TYPE (EQUAL-TO X)
             (MEMBER-OF S))))
```

```
(LEMMA
  (FORALL ((Y THING)
           (S SET))
    (IS (INSERT Y S)
        SET)))

(LEMMA
  (FORALL ((X THING)
           (Y THING)
           (S SET))
    (IS (INSERT X (INSERT Y S))
        SET)))

(LEMMA
  (FORALL ((Y THING)
           (X THING)
           (S SET))
    (= (INSERT X (INSERT Y S))
       (INSERT Y (INSERT X S)))))
```

```
(IN-CONTEXT
    ((LET-BE Y THING)
     (LET-BE X THING)
     (LET-BE S SET)
     (LET-BE IY (INSERT Y S))
     (LET-BE IXY (INSERT X IY)))

  (NOTE (IS IY SET))
  (NOTE (IS IXY SET))

  (IN-CONTEXT
      ((LET-BE IX (INSERT X S))
       (LET-BE IYX (INSERT Y IX))
       (PUSH-GOAL (= IXY IYX)))
    (IN-CONTEXT
        ((PUSH-GOAL (IS IXY (SUBSET-OF IYX))))
      (IN-CONTEXT
          ((LET-BE Z (MEMBER-OF IXY)))
        (IN-CONTEXT
            ((PUSH-GOAL (IS Z (MEMBER-OF IYX))))
          (IN-CONTEXT
              ((SUPPOSE (= Z X)))
            (NOTE-GOAL))
          (IN-CONTEXT
              ((SUPPOSE (= Z Y)))
            (NOTE-GOAL))
          (NOTE-GOAL)))
      (NOTE+GENERALIZE-GOAL))
    (NOTE-GOAL)))
```

```
(LEMMA
   (FORALL ((S SET
               (EXISTS-SOME
                  (MEMBER-OF S)))
            (X (MEMBER-OF S))
            (S2 (SUBSET-OF S)))
     (IS (INSERT X S2)
         (SUBSET-OF S))))
```

```
(IN-CONTEXT
    ((LET-BE S SET
             (EXISTS-SOME (MEMBER-OF S)))
     (LET-BE S2 (SUBSET-OF S))
     (LET-BE X (MEMBER-OF S))
     (LET-BE SX2 (INSERT X S2))
     (PUSH-GOAL (IS SX2 (SUBSET-OF S))))
  (IN-CONTEXT
      ((LET-BE Y (MEMBER-OF SX2)))
    (IN-CONTEXT
        ((PUSH-GOAL (IS Y (MEMBER-OF S))))
      (IN-CONTEXT
          ((SUPPOSE (IS Y (MEMBER-OF S2))))
        (NOTE-GOAL))
      (NOTE-GOAL))
    (NOTE-GOAL)))
```

```
(LEMMA
  (FORALL ((X THING) (S SET))
    (= (INSERT X S)
       (INSERT X
               (INSERT X S)))))
```

```
(IN-CONTEXT
    ((LET-BE X THING)
     (LET-BE S SET)
     (LET-BE S2 (INSERT X S))
     (LET-BE S3 (INSERT X S2))
     (PUSH-GOAL (= S2 S3)))
  (IN-CONTEXT
      ((PUSH-GOAL (IS S3 (SUBSET-OF S2)))
       (LET-BE Y (MEMBER-OF S3)))
    (IN-CONTEXT
        ((PUSH-GOAL (IS Y (MEMBER-OF S2))))
      (IN-CONTEXT
          ((SUPPOSE (= Y X)))
        (NOTE-GOAL))
      (NOTE-GOAL))
    (NOTE-GOAL))
  (NOTE-GOAL))
```

The DEFNOTATION construct allows the user to define macros. The following form defines the operator MAKE-SET so that (MAKE-SET X) abbreviates (INSERT X THE-EMPTY-SET) and (MAKE-SET X1 X2 ...XN) abbreviates (INSERT X1 (MAKE-SET X2...XN)).

```
(DEFNOTATION (MAKE-SET &REST ELEMENTS)
  (IF (NULL ELEMENTS)
      'THE-EMPTY-SET
      `(INSERT ,(CAR ELEMENTS)
               (MAKE-SET ,@(CDR ELEMENTS)))))
```

```
(LEMMA
  (FORALL ((X THING))
    (IS (MAKE-SET X) SET)))


(LEMMA
  (FORALL ((X THING))
    (IS X (MEMBER-OF (MAKE-SET X)))))


(LEMMA
   (FORALL
       ((X THING)
        (Y (MEMBER-OF (MAKE-SET X))))
     (= X Y)))
```

```
(IN-CONTEXT
    ((LET-BE X THING)
     (LET-BE E THE-EMPTY-SET))
  (NOTE (IS (INSERT X E) SET))
  (NOTE (IS X (MEMBER-OF (INSERT X E))))
  (IN-CONTEXT
      ((LET-BE Y (MEMBER-OF (INSERT X E))))
    (NOTE (= X Y))))
```

```
(DEFTYPE SINGLETON-SET
  (WRITABLE-AS (MAKE-SET X)
    (X THING)))
```

```
(LEMMA (FORALL ((S SINGLETON-SET))
         (IS S SET)))


(LEMMA (FORALL ((S1 SINGLETON-SET))
         (EXISTS-SOME (MEMBER-OF S1))))


(LEMMA (FORALL ((S1 SINGLETON-SET))
         (AT-MOST-ONE (MEMBER-OF S1))))
```

```
(IN-CONTEXT
    ((LET-BE S1 SINGLETON-SET)
     (WRITE-AS S1 (MAKE-SET X)
               (X THING)))
  (NOTE (IS S1 SET))
  (NOTE (EXISTS-SOME (MEMBER-OF S1)))
  (IN-CONTEXT
      ((LET-BE Y1 (MEMBER-OF S1))
       (LET-BE Y2 (MEMBER-OF S1)))
    (NOTE (AT-MOST-ONE (MEMBER-OF S1)))))
```

```
(LEMMA
  (FORALL ((S SET))
    (=> (EXACTLY-ONE (MEMBER-OF S))
        (= S
           (MAKE-SET
             (THE (MEMBER-OF S)))))))


(LEMMA
  (FORALL ((S SET))
    (=> (EXACTLY-ONE (MEMBER-OF S))
        (IS S SINGLETON-SET))))
```

```
(IN-CONTEXT
    ((LET-BE S SET)
     (SUPPOSE (EXACTLY-ONE (MEMBER-OF S)))
     (LET-BE THE-MEMBER
             (THE (MEMBER-OF S)))
     (LET-BE S2 (MAKE-SET THE-MEMBER)))
  (NOTE (= S S2))
  (NOTE (IS S SINGLETON-SET)))
```

```
(LEMMA
  (FORALL ((X THING)
           (Y THING))
    (IS (MAKE-SET X Y)
        SET)))
```

```
(IN-CONTEXT
    ((LET-BE X THING)
     (LET-BE Y THING)
     (LET-BE SY (MAKE-SET Y))
     (LET-BE SXY (INSERT X SY)))
  (NOTE (IS SXY SET))
  (NOTE (IS X (MEMBER-OF SXY)))
  (IN-CONTEXT
      ((LET-BE Z (MEMBER-OF SXY)))
    (NOTE (OR (= Z X)
              (= Z Y)))))
```

```
(LEMMA
  (FORALL ((Y THING)
           (X THING))
    (IS X (MEMBER-OF (MAKE-SET X Y)))))
```

```
(LEMMA
  (FORALL ((X THING)
           (Y THING)
           (Z (MEMBER-OF
                (MAKE-SET X Y))))
    (OR (= Z X)
        (= Z Y))))
```

```
(LEMMA
  (FORALL ((Y THING)
           (X THING))
    (= (MAKE-SET X Y)
       (MAKE-SET Y X))))
```

```
(IN-CONTEXT
    ((LET-BE X THING)
     (LET-BE Y THING)
     (LET-BE E THE-EMPTY-SET))
  (NOTE (= (MAKE-SET X Y)
           (MAKE-SET Y X))))
```

```
(LEMMA
  (FORALL ((Y THING)
           (X THING)
           (Z THING))
    (= (MAKE-SET X Y Z)
       (MAKE-SET Y X Z))))
```

```
(IN-CONTEXT
    ((LET-BE X THING)
     (LET-BE Y THING)
     (LET-BE Z THING)
     (PUSH-GOAL
       (= (MAKE-SET X Y Z)
          (MAKE-SET Y X Z))))
  (IN-CONTEXT
      ((LET-BE S (MAKE-SET Z)))
    (NOTE-GOAL)))
```

```
(DEFTYPE (NOT-EQUAL-TO (X THING))
  (LAMBDA ((Y THING))
    (NOT (= X Y))))
```

```
(LEMMA
  (FORALL ((S SET))
    (EXISTS ((X THING))
      (NOT (IS X (MEMBER-OF S))))))
```

```
Russell's Paradox:
(IN-CONTEXT
    ((LET-BE S SET)
     (SUPPOSE
       (FORALL ((X THING))
         (IS X (MEMBER-OF S))))
     (LET-BE S2
             (THE-SET-OF-ALL
               (X (MEMBER-OF S))
               (NOT (IS X (MEMBER-OF X))))))
  (IN-CONTEXT
      ((SUPPOSE (IS S2 (MEMBER-OF S2))))
    (NOTE-CONTRADICTION))
  (NOTE-CONTRADICTION))
```

```
(LEMMA
  (FORALL ((X THING))
    (EXISTS-SOME (NOT-EQUAL-TO X))))
```

```
(IN-CONTEXT
    ((LET-BE X THING)
     (LET-BE SX (MAKE-SET X))
     (LET-BE Y THING
             (NOT (IS Y (MEMBER-OF SX)))))
  (NOTE (EXISTS-SOME (NOT-EQUAL-TO X))))
```

```
(DEFTYPE DOUBLETON-SET
  (WRITABLE-AS (MAKE-SET X Y)
    (X THING)
    (Y (NOT-EQUAL-TO X))))
```

```
(LEMMA (EXISTS-SOME DOUBLETON-SET))
```

```
(IN-CONTEXT
    ((LET-BE X THING)
     (LET-BE Y (NOT-EQUAL-TO X)))
  (NOTE (EXISTS-SOME DOUBLETON-SET)))
```

```
(DEFTYPE (OTHER-MEMBER (S SET) (X (MEMBER-OF S)))
   (AND-TYPE (MEMBER-OF S) (NOT-EQUAL-TO X)))
```

```
(LEMMA
  (FORALL ((S DOUBLETON-SET))
    (IS S SET)))

(LEMMA
  (FORALL ((S DOUBLETON-SET))
    (NOT (IS S SINGLETON-SET))))

(LEMMA
  (FORALL ((S DOUBLETON-SET))
    (EXISTS-SOME (MEMBER-OF S))))

(LEMMA
  (FORALL ((S DOUBLETON-SET)
           (Z (MEMBER-OF S)))
    (EXISTS-SOME (OTHER-MEMBER S Z))))

(LEMMA
  (FORALL ((S DOUBLETON-SET)
           (Z (MEMBER-OF S)))
    (AT-MOST-ONE (OTHER-MEMBER S Z))))

(LEMMA
  (FORALL ((S DOUBLETON-SET)
           (Z (MEMBER-OF S)))
    (= S
       (MAKE-SET
         Z
         (THE (OTHER-MEMBER S Z))))))
```

```
(IN-CONTEXT
    ((LET-BE S DOUBLETON-SET)
     (WRITE-AS S (MAKE-SET X Y)
               (X THING)
               (Y (NOT-EQUAL-TO X))))

  (NOTE (IS S SET))

  (NOTE (NOT (IS S SINGLETON-SET)))

  (NOTE (EXISTS-SOME (MEMBER-OF S)))

  (IN-CONTEXT
      ((LET-BE Z (MEMBER-OF S)))

    (IN-CONTEXT
        ((PUSH-GOAL
           (EXISTS-SOME
             (OTHER-MEMBER S Z))))

      (IN-CONTEXT
          ((SUPPOSE (= Z X)))
        (NOTE-GOAL))
      (NOTE-GOAL))

    (IN-CONTEXT
        ((PUSH-GOAL
           (AT-MOST-ONE (OTHER-MEMBER S Z)))
         (LET-BE W1 (OTHER-MEMBER S Z))
         (LET-BE W2 (OTHER-MEMBER S Z)))

      (IN-CONTEXT
          ((SUPPOSE (= Z X)))
        (NOTE-GOAL))
      (NOTE-GOAL))

    (IN-CONTEXT
      ((PUSH-GOAL
         (= S
            (MAKE-SET
              Z
              (THE (OTHER-MEMBER S Z))))))
      (IN-CONTEXT
          ((SUPPOSE (= X Z)))
        (NOTE-GOAL))
      (NOTE-GOAL))))
```

```
(LEMMA
 (FORALL ((S SINGLETON-SET))
   (NOT (IS S DOUBLETON-SET))))
```

```
(IN-CONTEXT
    ((LET-BE S SINGLETON-SET)
     (LET-BE X (THE (MEMBER-OF S))))
  (NOTE (NOT (IS S DOUBLETON-SET))))
```

```
                (DEFTYPE (SET-CONTAINING (X THING))
                  (LAMBDA ((S SET))
                    (IS X (MEMBER-OF S))))

                (DEFTYPE (SUPERSET-OF (S1 SET))
                  (LAMBDA ((S2 SET))
                    (IS S1 (SUBSET-OF S2))))

                (DEFTYPE (PROPER-SUPERSET-OF (S SET))
                  (AND-TYPE (SUPERSET-OF S) (NOT-EQUAL-TO S)))

                (DEFTYPE (PROPER-SUBSET-OF (S SET))
                  (AND-TYPE (SUBSET-OF S) (NOT-EQUAL-TO S)))

                (DEFTYPE (NOT-MEMBER-OF (S SET))
                  (LAMBDA ((X THING))
                    (NOT (IS X (MEMBER-OF S)))))

                (DEFTYPE NON-EMPTY-SET
                  (LAMBDA ((S SET))
                    (EXISTS-SOME (MEMBER-OF S))))

(LEMMA (EXISTS-SOME NON-EMPTY-SET))      (IN-CONTEXT
                                             ((LET-BE X THING)
                                              (LET-BE SX (MAKE-SET X)))
                                           (NOTE (EXISTS-SOME NON-EMPTY-SET)))

                (DEFTYPE (NON-EMPTY-SUBSET-OF (S NON-EMPTY-SET))
                  (AND-TYPE (SUBSET-OF S) NON-EMPTY-SET))

(LEMMA (FORALL ((S SET)                  (IN-CONTEXT
                (S2 (SUBSET-OF S))           ((LET-BE S SET)
                (S3 (SUBSET-OF S2)))          (LET-BE S2 (SUBSET-OF S))
         (IS S3 (SUBSET-OF S))))              (LET-BE S3 (SUBSET-OF S2))
                                              (PUSH-GOAL (IS S3 (SUBSET-OF S))))
                                           (IN-CONTEXT
                                               ((SUPPOSE
                                                  (EXISTS-SOME (MEMBER-OF S3)))
                                                (LET-BE X (MEMBER-OF S3)))
                                             (NOTE-GOAL))
                                           (NOTE-GOAL))

                (DEFTYPE FAMILY-OF-SETS
                  (LAMBDA ((F NON-EMPTY-SET))
                    (IS-EVERY (MEMBER-OF F) SET)))
```

```
(LEMMA (FORALL ((S1 SET))
         (IS (INSERT S1 THE-EMPTY-SET)
             FAMILY-OF-SETS)))

(LEMMA (EXISTS-SOME FAMILY-OF-SETS))
```

```
(IN-CONTEXT
    ((LET-BE S1 SET))
  (IN-CONTEXT
      ((LET-BE F1 (MAKE-SET S1))
       (LET-BE S (MEMBER-OF F1)))
    (NOTE (IS F1 FAMILY-OF-SETS))
    (NOTE (EXISTS-SOME FAMILY-OF-SETS))))
```

```
(LEMMA
  (FORALL ((S SET)
           (F1 FAMILY-OF-SETS))
    (IS (INSERT S F1)
        FAMILY-OF-SETS)))
```

```
(IN-CONTEXT
    ((LET-BE S SET)
     (LET-BE F1 FAMILY-OF-SETS)
     (LET-BE F2 (INSERT S F1))
     (PUSH-GOAL (IS F2 FAMILY-OF-SETS)))
  (IN-CONTEXT
      ((LET-BE FMEM (MEMBER-OF F2)))
    (IN-CONTEXT
        ((PUSH-GOAL (IS FMEM SET)))
      (IN-CONTEXT
          ((SUPPOSE (= FMEM S)))
        (NOTE-GOAL))
      (NOTE-GOAL))
    (NOTE-GOAL)))
```

```
(LEMMA
  (FORALL ((S2 SET)
           (S3 SET))
    (IS (MAKE-SET S2 S3)
        FAMILY-OF-SETS)))

(LEMMA
  (FORALL ((S1 SET)
           (S2 SET)
           (S3 SET))
    (IS (MAKE-SET S1 S2 S3)
        FAMILY-OF-SETS)))
```

```
(IN-CONTEXT
    ((LET-BE S1 SET)
     (LET-BE S2 SET)
     (LET-BE S3 SET))
  (IN-CONTEXT
      ((LET-BE F1 (MAKE-SET S3))
       (LET-BE F2 (MAKE-SET S2 S3))
       (LET-BE F3 (MAKE-SET S1 S2 S3)))
    (NOTE (IS F2 FAMILY-OF-SETS))
    (NOTE (IS F3 FAMILY-OF-SETS))))
```

```
(LEMMA
 (FORALL ((S NON-EMPTY-SET)
          (X (MEMBER-OF S))
          (Y (MEMBER-OF S)))
   (IS (MAKE-SET X Y)
       (SUBSET-OF S))))
```

```
(IN-CONTEXT
    ((LET-BE S NON-EMPTY-SET)
     (LET-BE X (MEMBER-OF S))
     (LET-BE Y (MEMBER-OF S))
     (LET-BE SXY (MAKE-SET X Y))
     (PUSH-GOAL (IS SXY (SUBSET-OF S))))
  (IN-CONTEXT
      ((LET-BE Z (MEMBER-OF SXY)))
    (IN-CONTEXT
        ((PUSH-GOAL (IS Z (MEMBER-OF S))))
      (IN-CONTEXT
          ((SUPPOSE (= Z X)))
        (NOTE-GOAL))
      (NOTE-GOAL))
    (NOTE-GOAL)))
```

```
(LEMMA
  (FORALL ((S NON-EMPTY-SET)
           (X (MEMBER-OF S))
           (Y (MEMBER-OF S))
           (Z (MEMBER-OF S)))
    (IS (MAKE-SET X Y Z)
        (SUBSET-OF S))))
```

```
(IN-CONTEXT
    ((LET-BE S NON-EMPTY-SET)
     (LET-BE X (MEMBER-OF S))
     (LET-BE Y (MEMBER-OF S))
     (LET-BE Z (MEMBER-OF S))
     (LET-BE S2 (MAKE-SET X Y Z))
     (PUSH-GOAL (IS S2 (SUBSET-OF S))))
  (IN-CONTEXT
      ((LET-BE S3 (MAKE-SET Y Z)))
    (NOTE-GOAL)))
```

```
(DEFTYPE (MEMBER-OF-MEMBER (F FAMILY-OF-SETS))
  (WRITABLE-AS Z
    (Z (MEMBER-OF Y))
    (Y (MEMBER-OF F))))
```

```
(DEFTERM (FAMILY-UNION (F FAMILY-OF-SETS))
  (THE-SET-OF-ALL (MEMBER-OF-MEMBER F)))
```

```
(LEMMA
  (FORALL ((F FAMILY-OF-SETS))
    (IS (FAMILY-UNION F) SET)))

(LEMMA
  (FORALL ((F FAMILY-OF-SETS)
           (S (MEMBER-OF F)))
    (IS S (SUBSET-OF
            (FAMILY-UNION F)))))

(LEMMA
  (FORALL ((F FAMILY-OF-SETS)
           (S SET
              (IS-EVERY
                (MEMBER-OF F)
                (SUBSET-OF S))))
    (IS (FAMILY-UNION F)
        (SUBSET-OF S))))
```

```
(IN-CONTEXT
    ((LET-BE F FAMILY-OF-SETS)
     (LET-BE UNION-F (FAMILY-UNION F)))

  (NOTE (IS UNION-F SET))

  (IN-CONTEXT
    ((LET-BE S (MEMBER-OF F))
     (PUSH-GOAL (IS S (SUBSET-OF UNION-F))))
    (IN-CONTEXT
      ((SUPPOSE (EXISTS-SOME (MEMBER-OF S)))
       (LET-BE X (MEMBER-OF S)))
      (NOTE-GOAL))
    (NOTE-GOAL))

  (IN-CONTEXT
    ((LET-BE S SET
        (IS-EVERY (MEMBER-OF F) (SUBSET-OF S)))
     (PUSH-GOAL (IS UNION-F (SUBSET-OF S))))
    (IN-CONTEXT
      ((SUPPOSE
          (EXISTS-SOME (MEMBER-OF UNION-F)))
       (LET-BE X (MEMBER-OF UNION-F))
       (LET-BE S2 (MEMBER-OF F)
         (IS X (MEMBER-OF S2))))
      (NOTE-GOAL))
    (NOTE-GOAL)))
```

```
(DEFTERM (UNION (S1 SET) (S2 SET))
  (FAMILY-UNION (MAKE-SET S1 S2)))
```

```
(LEMMA
 (FORALL ((S1 SET)
          (S2 SET))
   (IS (UNION S1 S2) SET)))
```

```
(LEMMA
  (FORALL ((S2 SET)
           (S1 SET))
    (IS S1
        (SUBSET-OF (UNION S1 S2)))))
```

```
(LEMMA
  (FORALL ((S1 SET) (S2 SET))
    (= (UNION S1 S2)
       (THE-SET-OF-ALL
         (OR-TYPE (MEMBER-OF S1)
                  (MEMBER-OF S2))))))
```

```
(IN-CONTEXT ((LET-BE S1 SET)
             (LET-BE S2 SET)
             (LET-BE F (MAKE-SET S1 S2))
             (LET-BE USET (UNION S1 S2)))
  (NOTE (IS USET SET))
  (NOTE (IS S1 (SUBSET-OF USET)))

  (IN-CONTEXT
    ((LET-BE USET2
             (THE-SET-OF-ALL
               (OR-TYPE (MEMBER-OF S1)
                        (MEMBER-OF S2)))))
     (PUSH-GOAL (= USET USET2)))

    (IN-CONTEXT
        ((PUSH-GOAL (IS USET (SUBSET-OF USET2))))
      (IN-CONTEXT
           ((SUPPOSE
              (EXISTS-SOME (MEMBER-OF USET)))
            (LET-BE X (MEMBER-OF USET))
            (LET-BE S3 (MEMBER-OF F)
              (IS X (MEMBER-OF S3))))

        (IN-CONTEXT
            ((PUSH-GOAL (IS X (MEMBER-OF USET2))))
          (IN-CONTEXT
              ((SUPPOSE (= S3 S1)))
            (NOTE-GOAL))
          (NOTE-GOAL))
        (NOTE-GOAL))
      (NOTE-GOAL))

    (IN-CONTEXT
        ((PUSH-GOAL (IS USET2 (SUBSET-OF USET))))
      (IN-CONTEXT
           ((SUPPOSE
              (EXISTS-SOME (MEMBER-OF USET2)))
            (LET-BE X (MEMBER-OF USET2)))

        (IN-CONTEXT
             ((PUSH-GOAL (IS X (MEMBER-OF USET))))
          (IN-CONTEXT
              ((SUPPOSE (IS X (MEMBER-OF S1))))
            (NOTE-GOAL))
          (NOTE-GOAL))
        (NOTE-GOAL))
      (NOTE-GOAL))
    (NOTE-GOAL)))
```

```
(LEMMA
  (FORALL ((S1 SET)
           (S2 SET)
           (S3 (AND-TYPE
                 (SUPERSET-OF S1)
                 (SUPERSET-OF S2))))
    (IS S3
        (SUPERSET-OF (UNION S1 S2)))))
```

```
(IN-CONTEXT
    ((LET-BE S1 SET)
     (LET-BE S2 SET)
     (LET-BE F (MAKE-SET S1 S2))
     (LET-BE USET (UNION S1 S2))
     (LET-BE S3 (AND-TYPE (SUPERSET-OF S1)
                          (SUPERSET-OF S2)))
     (PUSH-GOAL
       (IS S3 (SUPERSET-OF (FAMILY-UNION F)))))
    (IN-CONTEXT
       ((LET-BE S4 (MEMBER-OF F)))
      (IN-CONTEXT
         ((PUSH-GOAL (IS S4 (SUBSET-OF S3))))
        (IN-CONTEXT
           ((SUPPOSE (= S4 S1)))
          (NOTE-GOAL))
        (NOTE-GOAL))
      (NOTE-GOAL)))
```

```
(DEFTERM (FAMILY-INTERSECTION (F FAMILY-OF-SETS))
  (THE-SET-OF-ALL (X (MEMBER-OF-MEMBER F))
    (IS-EVERY (MEMBER-OF F) (SET-CONTAINING X))))
```

```
(LEMMA
  (FORALL ((F FAMILY-OF-SETS))
    (IS (FAMILY-INTERSECTION F) SET)))

(LEMMA
  (FORALL ((F FAMILY-OF-SETS)
           (S (MEMBER-OF F)))
    (IS S
        (SUPERSET-OF
          (FAMILY-INTERSECTION F)))))

(LEMMA
  (FORALL ((F FAMILY-OF-SETS)
           (S SET
              (FORALL
                 ((MEM2 (MEMBER-OF F)))
                (IS MEM2
                    (SUPERSET-OF S)))))
    (IS (FAMILY-INTERSECTION F)
        (SUPERSET-OF S))))
```

```
(IN-CONTEXT
    ((LET-BE F FAMILY-OF-SETS)
     (LET-BE INTERSECTION-F
             (FAMILY-INTERSECTION F)))
  (NOTE (IS INTERSECTION-F SET))
  (IN-CONTEXT
      ((LET-BE S (MEMBER-OF F))
       (PUSH-GOAL
         (IS S (SUPERSET-OF INTERSECTION-F))))
    (IN-CONTEXT
        ((SUPPOSE
           (EXISTS-SOME
             (MEMBER-OF INTERSECTION-F)))
         (LET-BE X (MEMBER-OF INTERSECTION-F)))
      (NOTE-GOAL))
    (NOTE-GOAL))
  (IN-CONTEXT
     ((LET-BE S SET
              (IS-EVERY (MEMBER-OF F)
                        (SUPERSET-OF S)))
      (PUSH-GOAL
        (IS INTERSECTION-F (SUPERSET-OF S))))
    (IN-CONTEXT
        ((SUPPOSE (EXISTS-SOME (MEMBER-OF S)))
         (LET-BE X (MEMBER-OF S))
         (LET-BE S2 (MEMBER-OF F)))
      (NOTE-GOAL))
    (NOTE-GOAL)))
```

```
(DEFTERM (INTERSECTION (S1 SET) (S2 SET))
  (FAMILY-INTERSECTION (MAKE-SET S1 S2)))
```

```
(LEMMA
  (FORALL ((S1 SET)
           (S2 SET))
    (IS (INTERSECTION S1 S2) SET)))

(LEMMA
  (FORALL ((S2 SET)
           (S1 SET))
    (IS S1
        (SUPERSET-OF
          (INTERSECTION S1 S2)))))

(LEMMA
  (FORALL ((S1 SET) (S2 SET))
    (= (INTERSECTION S1 S2)
       (THE-SET-OF-ALL
         (AND-TYPE (MEMBER-OF S1)
                   (MEMBER-OF S2))))))
```

```
(IN-CONTEXT
    ((LET-BE S1 SET)
     (LET-BE S2 SET)
     (LET-BE F (MAKE-SET S1 S2))
     (LET-BE ISET (INTERSECTION S1 S2)))

  (NOTE (IS ISET SET))

  (NOTE (IS S1 (SUPERSET-OF ISET)))

  (IN-CONTEXT
      ((LET-BE ISET2
               (THE-SET-OF-ALL
                 (AND-TYPE (MEMBER-OF S1)
                           (MEMBER-OF S2))))
       (PUSH-GOAL (= ISET ISET2)))

    (IN-CONTEXT
        ((PUSH-GOAL (IS ISET (SUBSET-OF ISET2))))
      (IN-CONTEXT
          ((SUPPOSE
             (EXISTS-SOME (MEMBER-OF ISET)))
           (LET-BE X (MEMBER-OF ISET)))
        (NOTE-GOAL))
      (NOTE-GOAL))

    (IN-CONTEXT
        ((PUSH-GOAL (IS ISET2 (SUBSET-OF ISET))))
      (IN-CONTEXT
          ((SUPPOSE
             (EXISTS-SOME (MEMBER-OF ISET2)))
           (LET-BE X (MEMBER-OF ISET2))
           (LET-BE S3 (MEMBER-OF F)))
        (IN-CONTEXT
            ((PUSH-GOAL (IS X (MEMBER-OF S3))))
          (IN-CONTEXT
              ((SUPPOSE (= S3 S1)))
            (NOTE-GOAL))
          (NOTE-GOAL))
        (NOTE-GOAL))
      (NOTE-GOAL))

    (NOTE-GOAL)))
```

```
(LEMMA
  (FORALL ((S1 SET)
           (S2 SET)
           (S3 (AND-TYPE
                 (SUBSET-OF S1)
                 (SUBSET-OF S2))))
    (IS S3
        (SUBSET-OF
          (INTERSECTION S1 S2)))))
```

```
(IN-CONTEXT
    ((LET-BE S1 SET)
     (LET-BE S2 SET)
     (LET-BE F (MAKE-SET S1 S2))
     (LET-BE ISET (INTERSECTION S1 S2))
     (LET-BE S3 (AND-TYPE (SUBSET-OF S1)
                          (SUBSET-OF S2)))
     (PUSH-GOAL (IS S3 (SUBSET-OF ISET))))

    (IN-CONTEXT
        ((LET-BE S4 (MEMBER-OF F)))
      (IN-CONTEXT
          ((PUSH-GOAL
             (IS S4 (SUPERSET-OF S3))))
        (IN-CONTEXT
            ((SUPPOSE (= S4 S1)))
          (NOTE-GOAL))
        (NOTE-GOAL))
      (NOTE-GOAL)))
```

```
(LEMMA
  (FORALL ((S2 SET)
           (S1 SET)
           (S3 SET))
    (= (INTERSECTION S1
                     (UNION S2 S3))
       (UNION (INTERSECTION S1 S2)
              (INTERSECTION S1 S3)))))
```

```
(IN-CONTEXT
    ((LET-BE S1 SET)
     (LET-BE S2 SET)
     (LET-BE S3 SET)
     (LET-BE U-S2-S3 (UNION S2 S3))
     (LET-BE I-S1-S2 (INTERSECTION S1 S2))
     (LET-BE I-S1-S3 (INTERSECTION S1 S3))
     (LET-BE ISET (INTERSECTION S1 U-S2-S3))
     (LET-BE USET (UNION I-S1-S2 I-S1-S3))
     (PUSH-GOAL (= ISET USET)))

  (IN-CONTEXT
      ((PUSH-GOAL (IS ISET (SUBSET-OF USET))))
    (IN-CONTEXT
        ((SUPPOSE
           (EXISTS-SOME (MEMBER-OF ISET)))
         (LET-BE X (MEMBER-OF ISET)))

      (IN-CONTEXT
          ((PUSH-GOAL (IS X (MEMBER-OF USET))))
        (IN-CONTEXT
            ((SUPPOSE (IS X (MEMBER-OF S2))))
          (NOTE-GOAL))
        (NOTE-GOAL))
      (NOTE-GOAL))
    (NOTE-GOAL))

  (IN-CONTEXT
      ((PUSH-GOAL (IS USET (SUBSET-OF ISET))))
    (IF-CONTEXT
        ((SUPPOSE
           (EXISTS-SOME (MEMBER-OF USET)))
         (LET-BE X (MEMBER-OF USET)))
      (IN-CONTEXT
          ((PUSH-GOAL (IS X (MEMBER-OF ISET))))
        (IN-CONTEXT
            ((SUPPOSE
               (IS X (MEMBER-OF I-S1-S2))))
          (NOTE-GOAL))
        (NOTE-GOAL))
      (NOTE-GOAL))
    (NOTE-GOAL))

  (NOTE-GOAL))
```

```
(LEMMA
  (FORALL ((S2 SET)
           (S1 SET)
           (S3 SET))
    (= (UNION S1
              (INTERSECTION S2 S3))
       (INTERSECTION (UNION S1 S2)
                     (UNION S1 S3)))))
```

```
(IN-CONTEXT
    ((LET-BE S1 SET)
     (LET-BE S2 SET)
     (LET-BE S3 SET)
     (LET-BE I-S2-S3 (INTERSECTION S2 S3))
     (LET-BE U-S1-S2 (UNION S1 S2))
     (LET-BE U-S1-S3 (UNION S1 S3))
     (LET-BE USET (UNION S1 I-S2-S3))
     (LET-BE ISET (INTERSECTION U-S1-S2 U-S1-S3))
     (PUSH-GOAL (= USET ISET)))

  (IN-CONTEXT
      ((PUSH-GOAL (IS USET (SUBSET-OF ISET))))
    (IN-CONTEXT
        ((SUPPOSE
           (EXISTS-SOME (MEMBER-OF USET)))
         (LET-BE X (MEMBER-OF USET)))
      (IN-CONTEXT
          ((PUSH-GOAL (IS X (MEMBER-OF ISET))))
        (IN-CONTEXT
            ((SUPPOSE (IS X (MEMBER-OF S1))))
          (NOTE-GOAL))
        (NOTE-GOAL))
      (NOTE-GOAL))
    (NOTE-GOAL))

  (IN-CONTEXT
      ((PUSH-GOAL (IS ISET (SUBSET-OF USET))))
    (IN-CONTEXT
        ((SUPPOSE
           (EXISTS-SOME (MEMBER-OF ISET)))
         (LET-BE X (MEMBER-OF ISET)))

      (IN-CONTEXT
          ((PUSH-GOAL (IS X (MEMBER-OF USET))))
        (IN-CONTEXT
            ((SUPPOSE (IS X (MEMBER-OF S1))))
          (NOTE-GOAL))
        (NOTE-GOAL))
      (NOTE-GOAL))
    (NOTE-GOAL))

  (NOTE-GOAL))
```

```
(LEMMA
 (FORALL ((S1 SET)
          (S3 (SUBSET-OF S1))
          (S2 SET))
   (IS (UNION S3 S2)
       (SUBSET-OF (UNION S1 S2)))))
```

```
(IN-CONTEXT
    ((LET-BE S1 SET)
     (LET-BE S2 SET)
     (LET-BE S3 (SUBSET-OF S1))
     (LET-BE USET1
             (UNION S1 S2))
     (LET-BE USET2
             (UNION S3 S2))
     (PUSH-GOAL (IS USET2 (SUBSET-OF USET1))))
  (IN-CONTEXT
      ((SUPPOSE
         (EXISTS-SOME (MEMBER-OF USET2)))
       (LET-BE X (MEMBER-OF USET2)))
    (IN-CONTEXT
        ((PUSH-GOAL (IS X (MEMBER-OF USET1))))
      (IN-CONTEXT
          ((SUPPOSE (IS X (MEMBER-OF S3))))
        (NOTE-GOAL))
      (NOTE-GOAL))
    (NOTE-GOAL))
  (NOTE-GOAL))
```

```
(LEMMA
  (FORALL ((S1 SET)
           (S3 (SUBSET-OF S1))
           (S2 SET))
    (IS (INTERSECTION S3 S2)
      (SUBSET-OF
        (INTERSECTION S1 S2)))))
```

```
(IN-CONTEXT
    ((LET-BE S1 SET)
     (LET-BE S2 SET)
     (LET-BE S3 (SUBSET-OF S1))
     (LET-BE ISET1
             (INTERSECTION S1 S2))
     (LET-BE ISET2
             (INTERSECTION S3 S2))
     (PUSH-GOAL (IS ISET2 (SUBSET-OF ISET1))))
  (IN-CONTEXT
      ((SUPPOSE
         (EXISTS-SOME (MEMBER-OF ISET2)))
       (LET-BE X (MEMBER-OF ISET2)))
    (NOTE-GOAL))
  (NOTE-GOAL))
```

```
(LEMMA
  (FORALL ((S1 SET)
           (S2 (SUBSET-OF S1)))
    (= S1
       (UNION S1 S2))))

(LEMMA
  (FORALL ((S1 SET)
           (S2 (SUBSET-OF S1)))
    (= S2
       (INTERSECTION S1 S2))))
```

```
(IN-CONTEXT
    ((LET-BE S1 SET)
     (LET-BE S2 (SUBSET-OF S1)))
  (IN-CONTEXT
      ((LET-BE USET
               (UNION S1 S2)))
    (NOTE (= S1
             (UNION S1 S2))))
  (IN-CONTEXT
      ((LET-BE ISET
               (INTERSECTION S1 S2)))
    (NOTE (= S2
             (INTERSECTION S1 S2)))))
```

```
(DEFTYPE (DISJOINT-FROM (S1 SET))
  (LAMBDA ((S2 SET))
    (= (INTERSECTION S1 S2)
       THE-EMPTY-SET)))
```

```
(LEMMA                                   (IN-CONTEXT ((LET-BE S1 SET)
  (FORALL ((S1 SET))                                  (LET-BE ESET THE-EMPTY-SET))
    (EXISTS-SOME (DISJOINT-FROM S1))))     (NOTE (EXISTS-SOME (DISJOINT-FROM S1))))
```

```
(LEMMA                                   (IN-CONTEXT
  (FORALL ((S1 SET) (S2 SET))                ((LET-BE S1 SET)
    (IFF (IS S1 (DISJOINT-FROM S2))           (LET-BE S2 SET)
         (IS-EVERY                            (LET-BE INT (INTERSECTION S1 S2))
           (MEMBER-OF S1)                     (PUSH-GOAL
           (NOT-MEMBER-OF S2)))))               (IFF (IS S1 (DISJOINT-FROM S2))
                                                     (IS-EVERY (MEMBER-OF S1)
                                                               (NOT-MEMBER-OF S2)))))
                                            (IN-CONTEXT
                                                ((SUPPOSE (IS-EVERY (MEMBER-OF S1)
                                                                    (NOT-MEMBER-OF S2))))
                                              (IN-CONTEXT
                                                  ((SUPPOSE
                                                     (EXISTS-SOME (MEMBER-OF INT)))
                                                   (LET-BE X (MEMBER-OF INT)))
                                                (NOTE-CONTRADICTION))
                                              (NOTE-GOAL))
                                            (IN-CONTEXT
                                                ((SUPPOSE (IS S1 (DISJOINT-FROM S2))))
                                              (IN-CONTEXT
                                                  ((PUSH-GOAL
                                                     (IS-EVERY (MEMBER-OF S1)
                                                               (NOT-MEMBER-OF S2))))
                                                (IN-CONTEXT
                                                    ((SUPPOSE
                                                       (EXISTS-SOME (MEMBER-OF S1)))
                                                     (LET-BE X (MEMBER-OF S1)))
                                                  (NOTE-GOAL))
                                                (NOTE-GOAL))
                                              (NOTE-GOAL))
                                            (NOTE-GOAL))
```

```
(DEFTERM (SET-DIFFERENCE (S1 SET) (S2 SET))
  (THE-SET-OF-ALL
    (AND-TYPE (MEMBER-OF S1) (NOT-MEMBER-OF S2))))
```

```
(LEMMA
  (FORALL ((S1 SET) (S2 SET))
    (IS (SET-DIFFERENCE S1 S2)
        (SUBSET-OF S1))))

(LEMMA (FORALL ((S1 SET) (S2 SET))
         (IS (SET-DIFFERENCE S1 S2)
             (DISJOINT-FROM S2))))

(LEMMA (FORALL ((S1 SET) (S2 SET))
         (= (UNION
              S2
              (SET-DIFFERENCE S1 S2))
            (UNION S1 S2))))
```

```
(IN-CONTEXT
    ((LET-BE S1 SET)
     (LET-BE S2 SET)
     (LET-BE SD (SET-DIFFERENCE S1 S2)))

    (IN-CONTEXT
        ((PUSH-GOAL (IS SD (SUBSET-OF S1))))
      (IN-CONTEXT
          ((SUPPOSE
             (EXISTS-SOME (MEMBER-OF SD)))
           (LET-BE X (MEMBER-OF SD)))
        (NOTE-GOAL))
      (NOTE-GOAL))

    (IN-CONTEXT
        ((PUSH-GOAL
           (IS SD (DISJOINT-FROM S2))))
      (IN-CONTEXT
          ((SUPPOSE
             (EXISTS-SOME (MEMBER-OF SD)))
           (LET-BE X (MEMBER-OF SD)))
        (NOTE-GOAL))
      (NOTE-GOAL))

    (IN-CONTEXT
        ((LET-BE USET1 (UNION S2 SD))
         (LET-BE USET2 (UNION S1 S2))
         (PUSH-GOAL (= USET1 USET2)))

      (IN-CONTEXT
          ((PUSH-GOAL
             (IS USET2 (SUBSET-OF USET1))))
        (IN-CONTEXT
            ((SUPPOSE
               (EXISTS-SOME
                 (MEMBER-OF USET2)))
             (LET-BE X (MEMBER-OF USET2)))
          (IN-CONTEXT
              ((PUSH-GOAL
                 (IS X (MEMBER-OF USET1))))
            (IN-CONTEXT
                ((SUPPOSE
                   (IS X (MEMBER-OF S2))))
              (NOTE-GOAL))
            (NOTE-GOAL))
          (NOTE-GOAL))
        (NOTE-GOAL))
      (NOTE-GOAL)))
```

```
(DEFTERM (REMOVE (X THING) (S SET))
  (SET-DIFFERENCE S (MAKE-SET X)))
```

```
(LEMMA
  (FORALL ((S SET) (X THING))
    (= (REMOVE X S)
       (THE-SET-OF-ALL
         (AND-TYPE (MEMBER-OF S)
                   (NOT-EQUAL-TO X))))))
```

```
(IN-CONTEXT
    ((LET-BE X THING)
     (LET-BE S SET)
     (LET-BE REM
             (REMOVE X S))
     (LET-BE S2 (MAKE-SET X))
     (LET-BE S3
        (THE-SET-OF-ALL
          (AND-TYPE (MEMBER-OF S)
                    (NOT-EQUAL-TO X))))
     (PUSH-GOAL (= REM S3)))

 (IN-CONTEXT
     ((PUSH-GOAL (IS REM (SUBSET-OF S3))))
   (IN-CONTEXT
       ((SUPPOSE
          (EXISTS-SOME (MEMBER-OF REM)))
        (LET-BE Y (MEMBER-OF REM)))
      (NOTE (IS Y (NOT-EQUAL-TO X)))
      (NOTE-GOAL))
   (NOTE-GOAL))
 (IN-CONTEXT
       ((PUSH-GOAL (IS S3 (SUBSET-OF REM))))
   (IN-CONTEXT
       ((SUPPOSE (EXISTS-SOME (MEMBER-OF S3)))
        (LET-BE Y (MEMBER-OF S3)))
     (NOTE (IS Y (NOT-MEMBER-OF
                   (INSERT X THE-EMPTY-SET))))
     (NOTE-GOAL))
   (NOTE-GOAL))

 (NOTE-GOAL))
```

```
(LEMMA
  (FORALL ((S SET)
           (X THING)
           (Y THING))
    (= (REMOVE Y (REMOVE X S))
       (THE-SET-OF-ALL
         (AND-TYPE (MEMBER-OF S)
                   (NOT-EQUAL-TO X)
                   (NOT-EQUAL-TO Y))))))
```

```
(IN-CONTEXT
    ((LET-BE X THING)
     (LET-BE Y THING)
     (LET-BE S SET)
     (LET-BE SX (REMOVE X S))
     (LET-BE SYX (REMOVE Y SX))
     (LET-BE SYX2
             (THE-SET-OF-ALL
               (AND-TYPE (MEMBER-OF S)
                         (NOT-EQUAL-TO X)
                         (NOT-EQUAL-TO Y))))
     (PUSH-GOAL (= SYX SYX2)))
   (IN-CONTEXT
      ((PUSH-GOAL (IS SYX (SUBSET-OF SYX2))))
     (IN-CONTEXT
         ((SUPPOSE
            (EXISTS-SOME (MEMBER-OF SYX)))
          (LET-BE Z (MEMBER-OF SYX)))
      (NOTE-GOAL))
     (NOTE-GOAL))
   (IN-CONTEXT
      ((PUSH-GOAL (IS SYX2 (SUBSET-OF SYX))))
     (IN-CONTEXT
         ((SUPPOSE
            (EXISTS-SOME (MEMBER-OF SYX2)))
          (LET-BE Z (MEMBER-OF SYX2)))
      (NOTE-GOAL))
     (NOTE-GOAL))
   (NOTE-GOAL))
```

```
(LEMMA
  (FORALL ((Y THING)
           (X THING)
           (S SET))
    (= (REMOVE X (REMOVE Y S))
       (REMOVE Y (REMOVE X S)))))
```

```
(IN-CONTEXT
    ((LET-BE X THING)
     (LET-BE Y THING)
     (LET-BE S SET)
     (LET-BE SXY (REMOVE X (REMOVE Y S)))
     (LET-BE SYX (REMOVE Y (REMOVE X S)))
     (PUSH-GOAL (= SXY SYX)))
    (IN-CONTEXT
        ((PUSH-GOAL (IS SXY (SUBSET-OF SYX))))
      (IN-CONTEXT
          ((SUPPOSE
             (EXISTS-SOME (MEMBER-OF SXY)))
           (LET-BE Z (MEMBER-OF SXY)))
        (NOTE-GOAL))
      (NOTE+GENERALIZE-GOAL))
    (NOTE-GOAL))
```

```
(DEFTERM (POWER-SET (S SET))
  (THE-SET-OF-ALL (SUBSET-OF S)))
```

```
(LEMMA
  (FORALL ((S SET))
    (IS (POWER-SET S)
        FAMILY-OF-SETS)))

(LEMMA
  (FORALL ((S SET))
    (= S
       (FAMILY-UNION (POWER-SET S)))))
```

```
(IN-CONTEXT
    ((LET-BE S SET)
     (LET-BE P (POWER-SET S)))

  (IN-CONTEXT
      ((LET-BE S2 (MEMBER-OF P)))
    (NOTE (IS P FAMILY-OF-SETS)))

  (IN-CONTEXT
      ((LET-BE S2
          (FAMILY-UNION (POWER-SET S))))
    (NOTE (= S (FAMILY-UNION (POWER-SET S))))))
```

A.2 Pairs, Rules and Structures

This section contains facts about pairs rules and structures. For any two things x and y the pair $< x,\ y >$ is implemented as the set $\{x,\ \{x,\ y\}\}$. A rule is a set of pairs. An objects which appears on the right side a pair in a rule r is called a *domain element* of r. The set of all domain elements of r is called the *rule domain* of the rule r (rule domains are different from map domains; map domains are discussed below).

A structure is a rule whose domain is a set of symbols. Ontic structures are similar to the "structures" or "records" used in computer programming langauges (e.g. structures defined via `DEFSTRUCT` in Common Lisp). The symbols in the domain of a structure rule are somtimes called the "slots" of the structure. From a mathematical point of view the most interesting structures have a `U-SET` slot which contains the "domain" or "underlying set" of the structure. A structure with a `U-SET` slot that contains a set is called a *set structure.* Many different kinds of mathematical objects can be modeled as set structures; partial orders, algebras, topologies, graphs, and differentiable manifolds can all be implemented as set structures.

```
(DEFTERM (MAKE-PAIR (X THING) (Y THING))
  (MAKE-SET (MAKE-SET X Y) (MAKE-SET X)))
```

```
(LEMMA
  (FORALL ((X THING) (Y THING))
    (= (FAMILY-UNION (MAKE-PAIR X Y))
       (MAKE-SET X Y))))

(LEMMA
  (FORALL ((Y THING) (X THING))
    (= (FAMILY-INTERSECTION
         (MAKE-PAIR X Y))
       (MAKE-SET X))))
```

```
(IN-CONTEXT
    ((LET-BE X THING)
     (LET-BE Y THING)
     (LET-BE SX (MAKE-SET X))
     (LET-BE SXY (MAKE-SET X Y))
     (LET-BE SPAIR (MAKE-PAIR X Y)))
  (NOTE (IS (FAMILY-UNION SPAIR) SXY))
  (NOTE (IS (FAMILY-INTERSECTION SPAIR) SX)))
```

```
(DEFTYPE PAIR
  (WRITABLE-AS (MAKE-PAIR X Y)
    (X THING)
    (Y THING)))

(DEFTERM (LEFT (P PAIR))
  (THE (MEMBER-OF (FAMILY-INTERSECTION P))))
```

```
(LEMMA
  (FORALL ((X THING) (Y THING))
    (= X
       (LEFT (MAKE-PAIR X Y)))))
```

```
(IN-CONTEXT
    ((LET-BE X THING)
     (LET-BE Y THING)
     (LET-BE P (MAKE-PAIR X Y))
     (LET-BE SX (FAMILY-INTERSECTION P)))
  (NOTE (= X (LEFT P))))
```

```
(DEFTERM (RIGHT (P PAIR))
  (IF (SINGLETON-SET P)
      (LEFT P)
      (THE (OTHER-MEMBER
             (FAMILY-UNION P)
             (LEFT P)))))
```

```
(LEMMA
  (FORALL ((X THING) (Y THING))
    (= Y
       (RIGHT (MAKE-PAIR X Y)))))
```

```
(IN-CONTEXT
    ((LET-BE X THING)
     (LET-BE Y THING)
     (LET-BE P (MAKE-PAIR X Y))
     (PUSH-GOAL (= Y (RIGHT P)))
     (LET-BE MX (MAKE-SET X))
     (LET-BE MY (MAKE-SET X Y)))
  (IN-CONTEXT
      ((SUPPOSE (= X Y)))
    (NOTE-GOAL))
  (IN-CONTEXT
      ((SUPPOSE (NOT (= X Y))))
    (NOTE (NOT (= MX MY)))
    (NOTE-GOAL))
  (NOTE-GOAL))
```

For efficiency the type RULE, the operators THE-RULE and THE-FUNCTION and the type generators DOMAIN-TYPE, and the type generator RULE-BETWEEN are all implemented primitively. If f is a syntactically small function expression of one argument then the term (THE-RULE f) denotes a set theoretic object, such as a set of pairs, that corrosponds to the function f. Instances of the the type RULE are objects which can be written as (THE-RULE f) where is a syntactically small function expression of one argument. If R denotes a rule then the type (DOMAIN-TYPE R) is the type corrosponding to the domain of the rule (function) f and (THE-FUNCTION R) is the function corrosponding to R. If S1 and S2 denote sets then instances the type (RULE-BETWEEN S1 S2) are rules that give mappings from S1 into S2.

```
(DEFTYPE (DOMAIN-TYPE (R RULE))
  (MEMBER-OF (RULE-DOMAIN R)))
```

```
(LEMMA                                  (IN-CONTEXT
  (FORALL ((S1 NON-EMPTY-SET)               ((LET-BE S1 NON-EMPTY-SET)
           (S2 NON-EMPTY-SET))               (LET-BE S2 NON-EMPTY-SET)
    (EXISTS-SOME                             (LET-BE Y (MEMBER-OF S2))
      (RULE-BETWEEN S1 S2))))                (LET-BE R
                                               (THE-RULE ((X (MEMBER-OF S1))) Y)))
                                          (NOTE
                                           (EXISTS-SOME (RULE-BETWEEN S1 S2))))
```

```
(DEFTERM (RESTRICT-RULE (R RULE)
                        (S (SUBSET-OF
                             (RULE-DOMAIN R))))
  (THE-RULE ((X (MEMBER-OF S)))
    (APPLY-RULE R X)))
```

```
(DEFTERM (RESTRICT-RELATION (R RELATION)
                            (S (SUBSET-OF
                                 (RULE-DOMAIN R))))
  (THE-RULE ((X (MEMBER-OF S)))
    (INTERSECTION S (APPLY-RULE R X))))
```

```
(DEFTYPE (INJECTIVE-RULE-BETWEEN (S1 SET) (S2 SET))
  (LAMBDA ((R (RULE-BETWEEN S1 S2)))
    (FORALL ((Y (MEMBER-OF S1)))
      (EXACTLY-ONE (X (MEMBER-OF (RULE-DOMAIN R)))
        (= (APPLY-RULE R X) (APPLY-RULE R Y))))))

(DEFTYPE INJECTIVE-RULE
  (WRITABLE-AS R
    (R (INJECTIVE-RULE-BETWEEN S1 S2))
    (S1 SET)
    (S2 SET)))

(DEFTERM (RULE-RANGE (R RULE))
  (THE-SET-OF-ALL
    (WRITABLE-AS (APPLY-RULE R X)
      (X (MEMBER-OF (RULE-DOMAIN R))))))
```

The type SYMBOL and the macro QUOTE are implemented primitively. All atomic quotations are symbols. A structure is a rule whose domain is a set of symbols.

```
(DEFTYPE STRUCTURE
  (LAMBDA ((R RULE))
    (AND (EXISTS-SOME
           (MEMBER-OF (RULE-DOMAIN R)))
         (IS-EVERY (MEMBER-OF (RULE-DOMAIN R))
                   SYMBOL))))

(DEFTYPE (SIGNATURE-SYMBOL (W STRUCTURE))
  (MEMBER-OF (RULE-DOMAIN W)))

(DEFTERM (STRUCTURE-COMPONENT
           (STRUCT STRUCTURE)
           (SYM (SIGNATURE-SYMBOL STRUCT)))
  (APPLY-RULE STRUCT SYM))

(DEFTERM (ASSIGN (ARG THING) (VALUE THING) (OLD-RULE RULE))
  (THE-RULE ((X (OR-TYPE
                  (EQUAL-TO ARG)
                  (MEMBER-OF (RULE-DOMAIN OLD-RULE)))))
    (IF (= X ARG)
        VALUE
        (APPLY-RULE OLD-RULE X))))
```

```
(LEMMA                                   (IN-CONTEXT
  (FORALL ((S SYMBOL)                        ((LET-BE W STRUCTURE)
           (VAL THING)                        (LET-BE S SYMBOL)
           (W STRUCTURE))                     (LET-BE VAL THING)
    (IS (ASSIGN S VAL W)                      (LET-BE W2 (ASSIGN S VAL W))
        STRUCTURE))                           (PUSH-GOAL (IS W2 STRUCTURE)))
                                           (IN-CONTEXT
                                               ((LET-BE SYM
                                                  (MEMBER-OF (RULE-DOMAIN W2))))
                                             (IN-CONTEXT
                                                 ((PUSH-GOAL (IS SYM SYMBOL)))
                                               (IN-CONTEXT ((SUPPOSE (= SYM S)))
                                                 (NOTE-GOAL))
                                               (NOTE-GOAL))
                                             (NOTE-GOAL)))
```

```
(DEFTERM (BASE-STRUCTURE (S SYMBOL) (X THING))
  (THE-RULE ((Z (EQUAL-TO S))) X))
```

```
(LEMMA                                   (IN-CONTEXT
  (FORALL ((S SYMBOL)                        ((LET-BE S SYMBOL)
           (X THING))                         (LET-BE X THING)
    (IS (BASE-STRUCTURE S X)                  (LET-BE W (BASE-STRUCTURE S X))
        STRUCTURE))                           (PUSH-GOAL (IS W STRUCTURE)))
                                           (NOTE (IS W STRUCTURE)))
```

```
(DEFTERM (MAKE-SET-STRUCTURE (S NON-EMPTY-SET))
  (BASE-STRUCTURE 'U-SET S))

(DEFTERM (U-SET (W STRUCTURE))
  (STRUCTURE-COMPONENT W 'U-SET))

(DEFTYPE SET-STRUCTURE
  (LAMBDA ((S STRUCTURE))
    (AND (IS 'U-SET (SIGNATURE-SYMBOL S))
         (IS (U-SET S) NON-EMPTY-SET))))
```

```
(LEMMA
  (FORALL ((S NON-EMPTY-SET))
    (IS (MAKE-SET-STRUCTURE S)
        SET-STRUCTURE)))

(LEMMA
  (FORALL ((S NON-EMPTY-SET))
    (= (U-SET (MAKE-SET-STRUCTURE S))
       S)))
```

```
(IN-CONTEXT
    ((LET-BE S NON-EMPTY-SET)
     (LET-BE M (MAKE-SET-STRUCTURE S))
     (LET-BE SYM 'U-SET))
  (NOTE (IS M SET-STRUCTURE))
  (NOTE (= (U-SET M) S)))
```

```
(DEFTYPE (IN-U-SET (W SET-STRUCTURE))
  (MEMBER-OF (U-SET W)))
```

```
(LEMMA
  (FORALL ((W SET-STRUCTURE))
    (EXISTS-SOME (IN-U-SET W))))

(LEMMA
  (FORALL ((W SET-STRUCTURE)
           (X (IN-U-SET W)))
    (IS X THING)))

(LEMMA
  (FORALL ((W SET-STRUCTURE)
           (X (IN-U-SET W)))
    (IS (MAKE-SET X)
        (NON-EMPTY-SUBSET-OF
          (U-SET W)))))

(LEMMA
  (FORALL ((W SET-STRUCTURE)
           (X (IN-U-SET W))
           (Y (IN-U-SET W)))
    (IS (MAKE-SET X Y)
        (SUBSET-OF (U-SET W)))))

(LEMMA
  (FORALL ((W SET-STRUCTURE)
           (X (IN-U-SET W))
           (S2 (SUBSET-OF (U-SET W))))
    (IS (INSERT X S2)
        (SUBSET-OF (U-SET W)))))
```

```
(IN-CONTEXT ((LET-BE W SET-STRUCTURE)
             (LET-BE S (U-SET W)))
  (NOTE (EXISTS-SOME (IN-U-SET W)))
  (IN-CONTEXT ((LET-BE X (IN-U-SET W)))
    (NOTE (IS X THING))
    (IN-CONTEXT ((LET-BE SX (MAKE-SET X)))
      (NOTE (IS SX (NON-EMPTY-SUBSET-OF S))))
    (IN-CONTEXT ((LET-BE Y (IN-U-SET W))
                 (LET-BE SXY (MAKE-SET X Y)))
      (NOTE (IS SXY (SUBSET-OF S))))
    (IN-CONTEXT ((LET-BE S2 (SUBSET-OF S))
                 (LET-BE SX2 (INSERT X S2)))
      (NOTE (IS SX2 (SUBSET-OF S))))))
```

```
(LEMMA
  (FORALL ((W SET-STRUCTURE)
           (S2 (SUBSET-OF (U-SET W))))
    (IS S2 SET)))

(LEMMA
  (FORALL ((W SET-STRUCTURE)
           (S2 (SUBSET-OF (U-SET W))))
    (IS-EVERY (MEMBER-OF S2)
              (IN-U-SET W))))

(LEMMA
  (FORALL ((W SET-STRUCTURE)
           (S2 (SUBSET-OF (U-SET W))))
    (=> (EXISTS-SOME (MEMBER-OF S2))
        (IS S2
            (NON-EMPTY-SUBSET-OF
              (U-SET W))))))
```

```
(IN-CONTEXT
    ((LET-BE W SET-STRUCTURE)
     (LET-BE S (U-SET W))
     (LET-BE S2 (SUBSET-OF (U-SET W))))
  (NOTE (IS S2 SET))
  (NOTE (IS-EVERY (MEMBER-OF S2)
                  (IN-U-SET W)))
  (IN-CONTEXT
      ((SUPPOSE
         (EXISTS-SOME (MEMBER-OF S2))))
    (NOTE (IS S2 (NON-EMPTY-SUBSET-OF S)))))
```

A.3 Maps

The terminology used in the proof of Stone's theorem makes a distinction between rules and maps; a rule is a just a set of pairs while a map consists of a domain set structure, a range set structure, and a rule between the underlying sets of the domain and range structures. The significance of the distinction between rules and maps can be seen in the following formula:

```
(IS (DOMAIN F) LATTICE)
```

If `F` denoted a rule (a set of pairs) there would be no well defined domain structure for `F`, at best the domain of `F` would be an unstructured set. On the other hand maps, as opposed to rules, have specified domain and range *structures* and it is possible that the domain of `F` is in fact a lattice.

Category theory generalizes the notion of a map to the notion of a "morphism". A morphism is like a map in that it has a domain and a range but the domain and range of a morphism need not be set structures. In anticipation of category theory we define a "mapoid" to be a structure with domain and range slots. A map is a mapoid in which the domain and range slots are filled with set structures and where the rule slot is filled with a rule between the underlying sets of the domain and range.

```
(DEFTYPE MAPOID
  (LAMBDA ((W STRUCTURE))
    (AND (IS 'DOMAIN (SIGNATURE-SYMBOL W))
         (IS 'RANGE (SIGNATURE-SYMBOL W)))))

(DEFTERM (MAKE-MAPOID (D THING) (R THING) (W STRUCTURE))
  (ASSIGN 'DOMAIN D
          (ASSIGN 'RANGE R W)))

(DEFTERM (DOMAIN (W STRUCTURE))
  (STRUCTURE-COMPONENT W 'DOMAIN))

(DEFTERM (RANGE (W STRUCTURE))
  (STRUCTURE-COMPONENT W 'RANGE))
```

```
(LEMMA
  (FORALL ((D THING)
           (R THING)
           (W STRUCTURE))
    (IS (MAKE-MAPOID D R W)
        MAPOID)))

(LEMMA
  (FORALL ((D THING)
           (R THING)
           (W STRUCTURE))
    (= D
       (DOMAIN
         (MAKE-MAPOID D R W)))))

(LEMMA
  (FORALL ((D THING)
           (R THING)
           (W STRUCTURE))
    (= R
       (RANGE
         (MAKE-MAPOID D R W)))))
```

```
(IN-CONTEXT
    ((LET-BE D THING)
     (LET-BE R THING)
     (LET-BE W STRUCTURE)
     (LET-BE M (MAKE-MAPOID D R W))
     (LET-BE W2 (ASSIGN 'RANGE R W))
     (LET-BE SYM1 'DOMAIN)
     (LET-BE SYM2 'RANGE))
  (NOTE (IS M MAPOID))
  (NOTE (= D (DOMAIN M)))
  (NOTE (= R (RANGE M))))
```

```
(DEFTERM (MAKE-MAP (G SET-STRUCTURE)
                   (H SET-STRUCTURE)
                   (R (RULE-BETWEEN
                        (U-SET G)
                        (U-SET H))))
  (MAKE-MAPOID
    G
    H
    (BASE-STRUCTURE 'RULE R)))

(DEFTYPE (MAP-BETWEEN (G SET-STRUCTURE)
                      (H SET-STRUCTURE))
  (WRITABLE-AS (MAKE-MAP G H R)
    (R (RULE-BETWEEN (U-SET G)
                     (U-SET H)))))
```

```
(LEMMA
  (FORALL ((G SET-STRUCTURE)
           (H SET-STRUCTURE))
    (EXISTS-SOME
      (RULE-BETWEEN (U-SET G)
                    (U-SET H)))))

(LEMMA
  (FORALL ((G SET-STRUCTURE)
           (H SET-STRUCTURE)
           (R (RULE-BETWEEN (U-SET G)
                            (U-SET H))))
    (IS R RULE)))

(LEMMA
  (FORALL ((H SET-STRUCTURE)
           (G SET-STRUCTURE)
           (R (RULE-BETWEEN (U-SET G)
                            (U-SET H))))
    (= (RULE-DOMAIN R)
       (U-SET G))))

(LEMMA
  (FORALL
      ((G SET-STRUCTURE)
       (H SET-STRUCTURE)
       (R (RULE-BETWEEN (U-SET G)
                        (U-SET H)))
       (X (MEMBER-OF (RULE-DOMAIN R))))
    (IS (APPLY-MAP R X)
        (MEMBER-OF (U-SET H)))))
```

```
(IN-CONTEXT
    ((LET-BE G SET-STRUCTURE)
     (LET-BE H SET-STRUCTURE))
  (IN-CONTEXT
      ((LET-BE S1 (U-SET G))
       (LET-BE S2 (U-SET H)))
    (NOTE
      (EXISTS-SOME (RULE-BETWEEN S1 S2)))
    (IN-CONTEXT
        ((LET-BE R (RULE-BETWEEN S1 S2)))
      (NOTE (IS R RULE))
      (NOTE (= (RULE-DOMAIN R) (U-SET G)))
      (NOTE
        (FORALL ((X (MEMBER-OF
                      (RULE-DOMAIN R))))
          (IS (APPLY-RULE R X)
              (MEMBER-OF (U-SET H))))))))
```

```
(DEFTYPE (MAP-ON (G SET-STRUCTURE))
  (WRITABLE-AS F
    (F (MAP-BETWEEN G H))
    (H SET-STRUCTURE)))

(DEFTYPE (MAP-INTO (H SET-STRUCTURE))
  (WRITABLE-AS F
    (F (MAP-BETWEEN G H))
    (G SET-STRUCTURE)))

(DEFTYPE MAP
  (WRITABLE-AS (MAP-BETWEEN G H)
    (G SET-STRUCTURE)
    (H SET-STRUCTURE)))

(DEFTERM (MAP-RULE (M MAP))
  (STRUCTURE-COMPONENT M 'RULE))
```

```
(LEMMA
  (FORALL ((H SET-STRUCTURE)
           (G SET-STRUCTURE)
           (R (RULE-BETWEEN
                (U-SET G)
                (U-SET H))))
    (= (DOMAIN (MAKE-MAP G H R))
       G)))

(LEMMA
  (FORALL ((G SET-STRUCTURE)
           (H SET-STRUCTURE)
           (R (RULE-BETWEEN (U-SET G)
                            (U-SET H))))
    (= (RANGE (MAKE-MAP G H R))
       H)))

(LEMMA
  (FORALL ((G SET-STRUCTURE)
           (H SET-STRUCTURE)
           (R (RULE-BETWEEN
                (U-SET G)
                (U-SET H))))
    (= (MAP-RULE (MAKE-MAP G H R))
       R)))
```

```
(IN-CONTEXT
   ((LET-BE G SET-STRUCTURE)
    (LET-BE H SET-STRUCTURE)
    (LET-BE R (RULE-BETWEEN
                (U-SET G)
                (U-SET H)))
    (LET-BE M (MAKE-MAP G H R))
    (LET-BE B (BASE-STRUCTURE
                'RULE
                R))
    (LET-BE W (ASSIGN 'RANGE H B))
    (LET-BE SYM1 'DOMAIN)
    (LET-BE SYM2 'RANGE)
    (LET-BE SYM3 'RULE))
  (NOTE (= (DOMAIN M) G))
  (NOTE (= (RANGE M) H))
  (NOTE (= (MAP-RULE M) R)))
```

```
(LEMMA
  (FORALL ((H SET-STRUCTURE)
           (G SET-STRUCTURE)
           (M (MAP-BETWEEN G H)))
    (= G (DOMAIN M))))

(LEMMA
  (FORALL ((G SET-STRUCTURE)
           (H SET-STRUCTURE)
           (M (MAP-BETWEEN G H)))
    (= H (RANGE M))))
```

```
(IN-CONTEXT
   ((LET-BE G SET-STRUCTURE)
    (LET-BE H SET-STRUCTURE)
    (LET-BE M (MAP-BETWEEN G H))
    (WRITE-AS M (MAKE-MAP G H R)
      (R (RULE-BETWEEN
           (U-SET G)
           (U-SET H)))))
  (NOTE (= (DOMAIN M) G))
  (NOTE (= (RANGE M) H)))
```

```
(DEFTERM (APPLY-MAP (F MAP)
                    (X (IN-U-SET (DOMAIN F))))
  (APPLY-RULE (MAP-RULE F) X))
```

```
(LEMMA
  (FORALL ((M MAP))
    (IS (DOMAIN M)
        SET-STRUCTURE)))

(LEMMA
  (FORALL ((M MAP))
    (= (RULE-DOMAIN (MAP-RULE M))
       (U-SET (DOMAIN M)))))

(LEMMA
  (FORALL ((M MAP))
    (IS (RANGE M) SET-STRUCTURE)))

(LEMMA
  (FORALL ((M MAP))
    (IS (MAP-RULE M)
        (RULE-BETWEEN
          (U-SET (DOMAIN M))
          (U-SET (RANGE M))))))

(LEMMA
  (FORALL ((M MAP)
           (X (IN-U-SET (DOMAIN M))))
    (IS (APPLY-MAP M X)
        (IN-U-SET (RANGE M)))))
```

```
(IN-CONTEXT
    ((LET-BE M MAP)
     (WRITE-AS M (MAP-BETWEEN G H)
       (G SET-STRUCTURE)
       (H SET-STRUCTURE))
     (WRITE-AS M (MAKE-MAP G H R)
       (R (RULE-BETWEEN (U-SET G)
                        (U-SET H))))
     (LET-BE X (IN-U-SET (DOMAIN M))))
  (NOTE (IS (DOMAIN M) SET-STRUCTURE))
  (NOTE (= (RULE-DOMAIN (MAP-RULE M))
           (U-SET (DOMAIN M))))
  (NOTE (IS (RANGE M) SET-STRUCTURE))
  (NOTE (IS (MAP-RULE M)
            (RULE-BETWEEN
              (U-SET (DOMAIN M))
              (U-SET (RANGE M)))))
  (NOTE (IS (APPLY-MAP M X)
            (IN-U-SET (RANGE M)))))
```

```
(DEFTYPE (IN-MAP-DOMAIN (F MAP))
  (IN-U-SET (DOMAIN F)))

(DEFTYPE (IN-MAP-RANGE (F MAP))
  (IN-U-SET (RANGE F)))
```

```
(LEMMA
  (FORALL ((M MAP))
    (IS (U-SET (DOMAIN M))
        SET)))

(LEMMA
  (FORALL ((M MAP))
    (= (IN-U-SET (DOMAIN M))
       (MEMBER-OF
         (U-SET (DOMAIN M))))))

(LEMMA
  (FORALL ((M MAP))
    (EXISTS-SOME
      (MEMBER-OF
        (U-SET (DOMAIN M))))))

(LEMMA
  (FORALL ((M MAP))
    (IS (U-SET (RANGE M))
        SET)))

(LEMMA
 (FORALL ((M MAP))
   (= (IN-U-SET (RANGE M))
      (MEMBER-OF
        (U-SET (RANGE M))))))

(LEMMA
  (FORALL ((M MAP))
    (EXISTS-SOME
      (MEMBER-OF
        (U-SET (RANGE M))))))
```

```
(IN-CONTEXT
    ((LET-BE M MAP))
  (IN-CONTEXT
      ((LET-BE G (DOMAIN M))
       (LET-BE S (U-SET G)))
    (NOTE (IS S SET))
    (NOTE (= (IN-U-SET G)
             (MEMBER-OF S)))
    (NOTE
      (EXISTS-SOME (MEMBER-OF S))))
  (IN-CONTEXT
      ((LET-BE G (RANGE M))
       (LET-BE S (U-SET G)))
    (NOTE (IS S SET))
    (NOTE (= (IN-U-SET G)
             (MEMBER-OF S)))
    (NOTE
      (EXISTS-SOME (MEMBER-OF S)))))
```

```
(LEMMA
  (FORALL ((M MAP))
    (IS (MAP-RULE M) RULE)))
```

```
(IN-CONTEXT
    ((LET-BE M MAP)
     (LET-BE R (MAP-RULE M))
     (LET-BE S1 (U-SET (DOMAIN M)))
     (LET-BE S2 (U-SET (RANGE M))))
  (NOTE (IS R RULE)))
```

```
(DEFTERM (APPLY-MAP-TO-SET
           (F MAP)
           (S (SUBSET-OF (U-SET (DOMAIN F)))))
  (THE-SET-OF-ALL
    (WRITABLE-AS (APPLY-MAP F X)
      (X (MEMBER-OF S)))))
```

```
(LEMMA
  (FORALL ((M MAP)
           (S (SUBSET-OF
                (U-SET (DOMAIN M)))))
    (IS (APPLY-MAP-TO-SET M S)
        (SUBSET-OF
          (U-SET (RANGE M))))))
```

```
(IN-CONTEXT
    ((LET-BE M MAP)
     (LET-BE DSET (U-SET (DOMAIN M)))
     (LET-BE RSET (U-SET (RANGE M)))
     (LET-BE S (SUBSET-OF DSET))
     (LET-BE S2 (APPLY-MAP-TO-SET M S))
     (PUSH-GOAL
       (IS S2 (SUBSET-OF RSET))))
  (IN-CONTEXT
      ((SUPPOSE
         (EXISTS-SOME (MEMBER-OF S2)))
       (LET-BE X (MEMBER-OF S2))
       (WRITE-AS X (APPLY-MAP M Y)
         (Y (MEMBER-OF S))))
    (NOTE-GOAL))
  (NOTE-GOAL))
```

```
(DEFTERM (IMAGE (F MAP))
  (APPLY-MAP-TO-SET F (U-SET (DOMAIN F))))
```

```
(LEMMA
  (FORALL ((M MAP))
    (= (IMAGE M)
       (THE-SET-OF-ALL
         (WRITABLE-AS (APPLY-MAP M X)
           (X (IN-U-SET
                (DOMAIN M))))))))

(LEMMA
  (FORALL ((M MAP))
    (EXISTS-SOME
      (MEMBER-OF (IMAGE M)))))

(LEMMA
  (FORALL ((M MAP))
    (IS (IMAGE M)
        (NON-EMPTY-SUBSET-OF
          (U-SET (RANGE M))))))
```

```
(IN-CONTEXT
    ((LET-BE M MAP)
     (LET-BE S (U-SET (DOMAIN M)))
     (LET-BE S2 (IMAGE M)))
  (NOTE
    (= (IMAGE M)
       (THE-SET-OF-ALL
         (WRITABLE-AS (APPLY-MAP M X)
           (X (IN-U-SET
                (DOMAIN M)))))))
  (IN-CONTEXT
      ((LET-BE S3 (U-SET (RANGE M)))
       (LET-BE X (IN-U-SET (DOMAIN M))))
    (NOTE
      (EXISTS-SOME (MEMBER-OF (IMAGE M))))
    (NOTE
      (IS S2 (NON-EMPTY-SUBSET-OF S3)))))
```

```
(DEFTERM (PREIMAGE (F MAP)
                   (S (SUBSET-OF
                        (U-SET (RANGE F)))))
  (THE-SET-OF-ALL (X (MEMBER-OF
                       (U-SET (DOMAIN F))))
    (IS (APPLY-MAP F X) (MEMBER-OF S))))
```

```
(LEMMA
  (FORALL ((F MAP)
           (S (NON-EMPTY-SUBSET-OF
                (IMAGE F))))
    (IS S
        (SUBSET-OF
          (U-SET (RANGE F))))))
```

```
(IN-CONTEXT
    ((LET-BE F MAP)
     (LET-BE ISET (IMAGE F))
     (LET-BE S (NON-EMPTY-SUBSET-OF
                 (IMAGE F)))
     (LET-BE RSET (U-SET (RANGE F))))
  (NOTE (IS S (SUBSET-OF RSET)))
  (NOTE (EXISTS-SOME (MEMBER-OF S))))
```

```
(LEMMA
  (FORALL ((F MAP)
           (S (NON-EMPTY-SUBSET-OF
(IMAGE F))))
    (EXISTS-SOME (MEMBER-OF S))))
```

```
(LEMMA
  (FORALL ((F MAP)
           (Y (MEMBER-OF (IMAGE F))))
    (EXISTS-SOME
      (MEMBER-OF
        (PREIMAGE F (MAKE-SET Y))))))
```

```
(LEMMA
  (FORALL ((F MAP)
           (Y (MEMBER-OF (IMAGE F))))
    (= (PREIMAGE F (MAKE-SET Y))
       (THE-SET-OF-ALL
            (X (IN-U-SET (DOMAIN F)))
         (= (APPLY-MAP F X) Y)))))
```

```
(IN-CONTEXT
    ((LET-BE F MAP)
     (LET-BE ISET (IMAGE F))
     (LET-BE Y (MEMBER-OF ISET))
     (LET-BE SY (MAKE-SET Y))
     (LET-BE PRE-Y1 (PREIMAGE F SY))
     (LET-BE PRE-Y2
       (THE-SET-OF-ALL (X (IN-U-SET
                            (DOMAIN F)))
         (= (APPLY-MAP F X) Y))))
  (IN-CONTEXT
      ((WRITE-AS Y (APPLY-MAP F X)
         (X (IN-U-SET (DOMAIN F)))))
    (NOTE
      (EXISTS-SOME (MEMBER-OF PRE-Y1))))
  (IN-CONTEXT
      ((PUSH-GOAL (= PRE-Y1 PRE-Y2)))
    (IN-CONTEXT
        ((LET-BE X (MEMBER-OF PRE-Y1))
         (LET-BE FX (APPLY-MAP F X)))
      (NOTE (IS PRE-Y1 (SUBSET-OF PRE-Y2)))
      (NOTE
        (EXISTS-SOME (MEMBER-OF PRE-Y2))))
    (IN-CONTEXT
        ((LET-BE X (MEMBER-OF PRE-Y2)))
      (NOTE-GOAL))))
```

```
(DEFTYPE INJECTION
  (LAMBDA ((F MAP))
    (IS (MAP-RULE F)
        INJECTIVE-RULE)))
```

```
(LEMMA
  (FORALL ((M MAP))
    (=> (FORALL ((X (MEMBER-OF
                     (IMAGE M))))
          (IS (PREIMAGE M (MAKE-SET X))
              SINGLETON-SET))
        (IS M INJECTION))))
```

```
(IN-CONTEXT
    ((LET-BE M MAP)
     (SUPPOSE
       (FORALL ((Y (MEMBER-OF (IMAGE M))))
         (IS (PREIMAGE M (MAKE-SET Y))
             SINGLETON-SET)))
     (PUSH-GOAL (IS M INJECTION)))
  (IN-CONTEXT
      ((LET-BE R (MAP-RULE M))
       (LET-BE S1 (U-SET (DOMAIN M)))
       (LET-BE S2 (U-SET (RANGE M)))
       (LET-BE X (IN-U-SET (DOMAIN M)))
       (LET-BE MX (APPLY-MAP M X)))
    (IN-CONTEXT
        ((LET-BE PRE-MX
            (PREIMAGE M (MAKE-SET MX))))
      (NOTE (EXACTLY-ONE (MEMBER-OF PRE-MX))))
    (IN-CONTEXT
        ((LET-BE X2 (IN-U-SET (DOMAIN M))
            (= (APPLY-RULE R X2)
               (APPLY-MAP M X)))
         (LET-BE X3 (IN-U-SET (DOMAIN M))
            (= (APPLY-RULE R X3)
               (APPLY-MAP M X))))
      (NOTE-GOAL))))
```

```
(DEFTYPE (INJECTION-BETWEEN (G SET-STRUCTURE)
                            (H SET-STRUCTURE))
  (AND-TYPE (MAP-BETWEEN G H)
            INJECTION))

(DEFTYPE SURJECTION
  (LAMBDA ((F MAP))
    (= (IMAGE F)
       (U-SET (RANGE F)))))

(DEFTYPE (SURJECTION-BETWEEN (G SET-STRUCTURE)
                             (H SET-STRUCTURE))
  (AND-TYPE (MAP-BETWEEN G H)
            SURJECTION))

(DEFTYPE BIJECTION
  (AND-TYPE SURJECTION
            INJECTION))

(DEFTYPE (BIJECTION-BETWEEN (G SET-STRUCTURE)
                            (H SET-STRUCTURE))
  (AND-TYPE (MAP-BETWEEN G H)
            BIJECTION))

(DEFTERM (IDENTITY-MAP (W SET-STRUCTURE))
  (MAKE-MAP
    W
    W
    (THE-RULE ((X (IN-U-SET W)))
       X)))
```

```
(LEMMA
  (FORALL ((W SET-STRUCTURE))
    (IS (IDENTITY-MAP W)
        (MAP-BETWEEN W W))))

(LEMMA
  (FORALL ((W SET-STRUCTURE)
           (X (MEMBER-OF (U-SET W))))
    (= (APPLY-MAP (IDENTITY-MAP W) X)
       X)))
```

```
(IN-CONTEXT ((LET-BE W SET-STRUCTURE)
             (LET-BE R
               (THE-RULE ((X (IN-U-SET W)))
                 X))
             (LET-BE S (U-SET W))
             (LET-BE X (MEMBER-OF S))
             (LET-BE I (IDENTITY-MAP W)))
  (NOTE (IS I (MAP-BETWEEN W W)))
  (NOTE (= (APPLY-MAP I X) X)))
```

```
(LEMMA
  (FORALL ((W SET-STRUCTURE))
    (IS (IDENTITY-MAP W) BIJECTION)))
```

```
(IN-CONTEXT
    ((LET-BE W SET-STRUCTURE)
     (LET-BE I (IDENTITY-MAP W))
     (PUSH-GOAL (IS I BIJECTION)))

  (IN-CONTEXT
      ((PUSH-GOAL (IS I SURJECTION)))
    (IN-CONTEXT
        ((LET-BE ISET1 (IMAGE I))
         (LET-BE ISET2 (U-SET W))
         (PUSH-GOAL (= ISET1 ISET2)))
      (IN-CONTEXT
          ((LET-BE X (MEMBER-OF ISET2)))
        (NOTE-GOAL)))
    (NOTE-GOAL))

  (IN-CONTEXT
      ((PUSH-GOAL (IS I INJECTION))
       (LET-BE X (IN-U-SET (RANGE I)))
       (LET-BE PRE-X
         (PREIMAGE I (MAKE-SET X)))
       (LET-BE PREX1 (MEMBER-OF PRE-X))
       (LET-BE PREX2 (MEMBER-OF PRE-X)))
    (NOTE (EXACTLY-ONE
            (MEMBER-OF PRE-X)))
    (NOTE-GOAL))

  (NOTE-GOAL))
```

```
(LEMMA (EXISTS-SOME INJECTION))
```

```
(IN-CONTEXT ((LET-BE M BIJECTION))
  (NOTE (EXISTS-SOME INJECTION)))
```

```
(LEMMA
  (FORALL ((M INJECTION)
           (Y (MEMBER-OF (IMAGE M))))
    (EXACTLY-ONE (X (IN-U-SET
                      (DOMAIN M)))
      (= (APPLY-MAP M X)
         Y))))
```

```
(IN-CONTEXT
    ((LET-BE M INJECTION)
     (LET-BE Y  (MEMBER-OF (IMAGE M)))
     (PUSH-GOAL
       (EXACTLY-ONE (X (IN-U-SET (DOMAIN M)))
          (= (APPLY-MAP M X)
             Y))))
  (IN-CONTEXT
      ((LET-BE R (MAP-RULE M))
       (WRITE-AS R (INJECTIVE-RULE-BETWEEN DSET S3)
         (DSET SET)
         (S3 SET)))
    (IN-CONTEXT
        ((WRITE-AS Y (APPLY-MAP M X)
           (X (IN-U-SET (DOMAIN M)))))
      (NOTE (EXISTS (S2 (IN-U-SET (DOMAIN M)))
              (= (APPLY-MAP M S2)
                 Y))))
    (IN-CONTEXT
        ((LET-BE X1 (IN-U-SET (DOMAIN M))
           (= (APPLY-MAP M X1) Y))
         (LET-BE X2 (IN-U-SET (DOMAIN M))
           (= (APPLY-MAP M X2) Y)))
      (NOTE-GOAL))))
```

```
(DEFTYPE (STRUCTURE-CONTAINING (S SET))
  (LAMBDA ((W SET-STRUCTURE))
    (IS S (SUBSET-OF (U-SET W)))))
```

```
(LEMMA
  (FORALL ((S NON-EMPTY-SET))
    (IS (MAKE-SET-STRUCTURE S)
        (STRUCTURE-CONTAINING S))))
```

```
(IN-CONTEXT
    ((LET-BE S NON-EMPTY-SET)
     (LET-BE W (MAKE-SET-STRUCTURE S)))
  (NOTE (IS W (STRUCTURE-CONTAINING S))))
```

```
(DEFTERM (SET!-RANGE
           (F MAP)
           (W (STRUCTURE-CONTAINING (IMAGE F))))
  (MAKE-MAP (DOMAIN F) W (MAP-RULE F)))
```

```
(LEMMA
  (FORALL ((F MAP))
    (EXISTS-SOME
      (STRUCTURE-CONTAINING
        (IMAGE F)))))

(LEMMA
  (FORALL ((F MAP)
           (W (STRUCTURE-CONTAINING
                (IMAGE F))))
    (IS W SET-STRUCTURE)))
```

```
(IN-CONTEXT ((LET-BE F MAP))
  (IN-CONTEXT ((LET-BE ISET (IMAGE F)))
    (NOTE
      (EXISTS-SOME
        (STRUCTURE-CONTAINING (IMAGE F))))
    (IN-CONTEXT
        ((LET-BE W
           (STRUCTURE-CONTAINING (IMAGE F))))
      (NOTE (IS W SET-STRUCTURE)))))
```

```
(LEMMA
  (FORALL ((F MAP)
           (W (STRUCTURE-CONTAINING
                (IMAGE F)))
           (X (MEMBER-OF (IMAGE F))))
    (IS X (IN-U-SET W))))

(LEMMA
  (FORALL ((F MAP)
           (W (STRUCTURE-CONTAINING
                (IMAGE F))))
    (IS (MAP-RULE F)
        (RULE-BETWEEN (U-SET (DOMAIN F))
                      (U-SET W)))))

(LEMMA
  (FORALL ((F MAP)
           (W (STRUCTURE-CONTAINING
                (IMAGE F))))
    (IS (SET!-RANGE F W)
        (MAP-BETWEEN (DOMAIN F) W))))

(LEMMA
  (FORALL ((F MAP)
           (W (STRUCTURE-CONTAINING
                (IMAGE F))))
    (IS (SET!-RANGE F W)
        MAP)))

(LEMMA
  (FORALL ((F MAP)
           (W (STRUCTURE-CONTAINING
                (IMAGE F))))
    (= (DOMAIN (SET!-RANGE F W))
       (DOMAIN F))))

(LEMMA
  (FORALL ((F MAP)
           (W (STRUCTURE-CONTAINING
                (IMAGE F))))
    (= (RANGE (SET!-RANGE F W))
       W)))

(LEMMA
  (FORALL ((F MAP)
           (W (STRUCTURE-CONTAINING
                (IMAGE F))))
    (= (MAP-RULE (SET!-RANGE F W))
       (MAP-RULE F))))

(LEMMA
  (FORALL ((F MAP)
           (W (STRUCTURE-CONTAINING
                (IMAGE F)))
           (X (IN-U-SET
                (DOMAIN
                  (SET!-RANGE F W)))))
    (= (APPLY-MAP (SET!-RANGE F W)
                  X)
       (APPLY-MAP F X))))
```

```
(IN-CONTEXT
   ((LET-BE F MAP)
    ((LET-BE W
          (STRUCTURE-CONTAINING
            (IMAGE F)))
     (LET-BE R (MAP-RULE F))))
  (IN-CONTEXT
     ((LET-BE X (MEMBER-OF
                  (IMAGE-OF F)))
      (LET-BE S1 (U-SET W))
      (LET-BE S2 (IMAGE F)))
    (NOTE (IS X (IN-U-SET W))))
  (IN-CONTEXT
     ((PUSH-GOAL
        (IS R (RULE-BETWEEN
                (U-SET (DOMAIN F))
                (U-SET W))))
      (LET-BE DSET (U-SET (DOMAIN F)))
      (LET-BE WSET (U-SET W))
      (LET-BE X (MEMBER-OF DSET))
      (LET-BE RX (APPLY-RULE R X)))
    (NOTE-GOAL))
  (IN-CONTEXT
     ((LET-BE F2 (SET!-RANGE F W)))
    (IN-CONTEXT
       ((LET-BE DSTRUCT (DOMAIN F)))
      (NOTE
        (IS F2 (MAP-BETWEEN DSTRUCT W)))
      (NOTE (IS F2 MAP))
      (NOTE (= (DOMAIN F2) (DOMAIN F)))
      (NOTE (= (RANGE F2) W))
      (NOTE (= (MAP-RULE F2)
               (MAP-RULE F))))
    (IN-CONTEXT
       ((LET-BE X (IN-U-SET (DOMAIN F2))))
      (NOTE (= (APPLY-MAP F2 X)
               (APPLY-MAP F X))))))
```

```
(LEMMA
  (FORALL ((F MAP)
           (W (STRUCTURE-CONTAINING
                (IMAGE F))))
    (= (IMAGE F)
       (IMAGE (SET!-RANGE F W)))))
```

```
(IN-CONTEXT
    ((LET-BE F MAP)
     (LET-BE W (STRUCTURE-CONTAINING
                 (IMAGE F)))
     (LET-BE F2 (SET!-RANGE F W))
     (LET-BE ISET (IMAGE F))
     (LET-BE ISET2 (IMAGE F2))
     (PUSH-GOAL (= ISET ISET2)))
  (IN-CONTEXT
      ((LET-BE X (MEMBER-OF ISET))
       (WRITE-AS X (APPLY-MAP F Y)
         (Y (IN-U-SET (DOMAIN F)))))
    (NOTE (IS ISET (SUBSET-OF ISET2))))
  (IN-CONTEXT
      ((LET-BE X (MEMBER-OF ISET2))
       (WRITE-AS X (APPLY-MAP F2 Y)
         (Y (IN-U-SET (DOMAIN F2)))))
    (NOTE (IS ISET2 (SUBSET-OF ISET))))
  (NOTE-GOAL))
```

A.4 Relations, Choice, and Relation Structures

Relations are implemented as non-deterministic rules. More specifically, a relation is implemented as a rule that maps an object to a set of "possible values". Objects x and y are related under the relation r just in case y is a member of the set $r(x)$.

A relation r is "total" just in case for all x in the rule domain of r the set $r(x)$ is not empty. A *choice function* for a total relation r is a rule r' such that for all x in the rule domain of r, $r'(x)$ is a member of $r(x)$. The axiom of choice (as stated here) says that every total relation has at least one choice function.

Transitive, symmetric, antisymmetric, reflexive and irreflexive relations are defined in the standard ways and some standard facts are proven, e.g. a transitive irreflexive relation is antisymmetric.

A relation structure is a set structure with a slot that contains a relation on the underlying set. This section contains a surprising number of trivial facts about relation srtuctures.

```
(DEFTYPE RELATION
   (LAMBDA ((R RULE))
     (FORALL ((X (MEMBER-OF (RULE-DOMAIN R))))
       (IS (APPLY-RULE R X) SET))))

(DEFTYPE (RELATED-TO (X (MEMBER-OF (RULE-DOMAIN R)))
                     (R RELATION))
  (MEMBER-OF (APPLY-RULE R X)))

(DEFTYPE (RELATION-RANGE (R RELATION))
  (FAMILY-UNION (RULE-RANGE R)))

(DEFTYPE TOTAL-RELATION
  (LAMBDA ((R RELATION))
    (FORALL ((X (MEMBER-OF (RULE-DOMAIN R))))
      (EXISTS-SOME (RELATED-TO X R)))))

(DEFTYPE (CHOICE-FUNCTION-FOR (R TOTAL-RELATION))
  (LAMBDA ((R2 (RULE-BETWEEN
                (RULE-DOMAIN R)
                (RELATION-RANGE R))))
    (FORALL ((X (MEMBER-OF (RULE-DOMAIN R1))))
      (IS (APPLY-RULE R2 X)
          (MEMBER-OF (APPLY-RULE R X))))))


;the axiom of choice:

(AXIOM
  (FORALL ((R TOTAL-RELATION))
    (EXISTS-SOME (CHOICE-FUNCTION-FOR R))))


(DEFTYPE (RELATION-ON (S SET))
  (RULE-BETWEEN S (POWER-SET S)))
```

```
(LEMMA
  (FORALL ((S SET))
    (EXISTS-SOME (RELATION-ON S))))

(LEMMA
  (FORALL ((S SET)
           (R (RELATION-ON S)))
    (IS R RELATION)))

(LEMMA
  (FORALL ((S SET)
           (R (RELATION-ON S)))
    (= (RULE-DOMAIN R) S)))
```

```
(IN-CONTEXT
    ((LET-BE S SET)
     (LET-BE P (POWER-SET S)))
  (NOTE (EXISTS-SOME (RELATION-ON S)))

  (IN-CONTEXT ((LET-BE R (RELATION-ON S)))
    (IN-CONTEXT ((PUSH-GOAL (IS R RELATION)))
      (IN-CONTEXT
          ((SUPPOSE
              (EXISTS-SOME (MEMBER-OF S)))
           (LET-BE X (MEMBER-OF S))
           (LET-BE Y (APPLY-RULE R X)))
        (NOTE-GOAL))
      (NOTE-GOAL))

    (NOTE (= (RULE-DOMAIN R) S))))
```

```
(LEMMA
  (FORALL ((S NON-EMPTY-SET)
           (R RELATION))
    (=> (AND (FORALL ((X (MEMBER-OF S)))
               (IS (APPLY-RULE R X)
                   (SUBSET-OF S)))
             (= (RULE-DOMAIN R) S))
        (IS R (RELATION-ON S)))))
```

```
(IN-CONTEXT
    ((LET-BE S NON-EMPTY-SET)
     (LET-BE R RELATION)
     (SUPPOSE (= (RULE-DOMAIN R) S))
     (SUPPOSE (FORALL ((X (MEMBER-OF S)))
                (IS (APPLY-RULE R X)
                    (SUBSET-OF S))))
     (PUSH-GOAL (IS R (RELATION-ON S))))
  (IN-CONTEXT ((LET-BE X (MEMBER-OF S))
               (LET-BE Y
                       (APPLY-RULE R X))
               (LET-BE D (POWER-SET S)))
    (NOTE-GOAL)))
```

```
(LEMMA
  (FORALL ((S NON-EMPTY-SET)
           (X (MEMBER-OF S))
           (R (RELATION-ON S))
           (Y (RELATED-TO X R)))
    (IS Y (MEMBER-OF S))))
```

```
(IN-CONTEXT
    ((LET-BE S NON-EMPTY-SET)
     (LET-BE R (RELATION-ON S))
     (LET-BE X (MEMBER-OF S))
     (PUSH-GOAL (IS-EVERY (RELATED-TO X R)
                          (MEMBER-OF S))))
  (IN-CONTEXT
      ((SUPPOSE
          (EXISTS-SOME (RELATED-TO X R)))
       (LET-BE Y
               (RELATED-TO X R))
       (LET-BE P (POWER-SET S))
       (LET-BE S2
               (APPLY-RULE R X)))
    (NOTE-GOAL))
  (NOTE-GOAL))
```

```
(DEFTERM (PROVIDE-RELATION (R (RELATION-ON (U-SET W)))
                           (W SET-STRUCTURE))
  (ASSIGN 'RELATION R W))

(DEFTYPE RELATION-STRUCTURE
  (LAMBDA ((W SET-STRUCTURE))
    (AND (IS 'RELATION
             (SIGNATURE-SYMBOL W))
         (IS (STRUCTURE-COMPONENT W 'RELATION)
             (RELATION-ON (U-SET W))))))

(DEFTERM (GET-RELATION (S RELATION-STRUCTURE))
  (STRUCTURE-COMPONENT S 'RELATION))
```

```
(LEMMA
  (FORALL ((W SET-STRUCTURE))
    (EXISTS-SOME
      (RELATION-ON (U-SET W)))))

(LEMMA
  (FORALL ((W SET-STRUCTURE)
           (R (RELATION-ON (U-SET W))))
    (IS (PROVIDE-RELATION R W)
        RELATION-STRUCTURE)))

(LEMMA
  (FORALL ((W SET-STRUCTURE)
           (R (RELATION-ON (U-SET W))))
    (= (GET-RELATION
         (PROVIDE-RELATION R W))
       R)))

(LEMMA
  (FORALL ((W SET-STRUCTURE)
           (R (RELATION-ON (U-SET W))))
    (= (U-SET (PROVIDE-RELATION R W))
       (U-SET W))))
```

```
(IN-CONTEXT ((LET-BE W SET-STRUCTURE)
             (LET-BE S (U-SET W)))
  (NOTE
    (EXISTS-SOME (RELATION-ON (U-SET W))))
  (IN-CONTEXT
      ((LET-BE R (RELATION-ON (U-SET W)))
       (LET-BE W2 (PROVIDE-RELATION R W))
       (LET-BE SYM1 'RELATION)
       (LET-BE SYM2 'U-SET))
    (NOTE (IS W2 RELATION-STRUCTURE))
    (NOTE (= (GET-RELATION W2) R))
    (NOTE (= (U-SET W2) (U-SET W)))))
```

```
(DEFTERM (MAKE-RELATION-STRUCTURE (R (RELATION-ON S))
                                  (S SET))
  (PROVIDE-RELATION R (MAKE-SET-STRUCTURE S)))
```

```
(LEMMA
  (FORALL ((S NON-EMPTY-SET)
           (R (RELATION-ON S)))
    (IS (MAKE-RELATION-STRUCTURE R S)
        RELATION-STRUCTURE)))

(LEMMA
  (FORALL ((S NON-EMPTY-SET)
           (R (RELATION-ON S)))
    (= (GET-RELATION
         (MAKE-RELATION-STRUCTURE R S))
       R)))

(LEMMA
  (FORALL ((S NON-EMPTY-SET)
           (R (RELATION-ON S)))
    (= (U-SET
         (MAKE-RELATION-STRUCTURE R S))
       S)))
```

```
(IN-CONTEXT
    ((LET-BE S NON-EMPTY-SET)
     (LET-BE R (RELATION-ON S))
     (LET-BE W (MAKE-RELATION-STRUCTURE R S))
     (LET-BE W2 (MAKE-SET-STRUCTURE S)))
  (NOTE (IS W RELATION-STRUCTURE))
  (NOTE (= (GET-RELATION W) R))
  (NOTE (= (U-SET W) S)))
```

```
(DEFTERM (RESTRICT-RELATION-STRUCTURE
           (R RELATION-STRUCTURE)
           (S (NON-EMPTY-SUBSET-OF (U-SET R))))
  (MAKE-RELATION-STRUCTURE
    (RESTRICT-RELATION (GET-RELATION R) S) S))
```

```
(LEMMA
  (FORALL ((R RELATION)
           (S2 (SUBSET-OF
                 (RULE-DOMAIN R))))
    (IS (RESTRICT-RELATION R S2)
        (RELATION-ON S2))))

(LEMMA
  (FORALL ((R RELATION)
           (S2 (SUBSET-OF
                 (RULE-DOMAIN R)))
           (X1 (MEMBER-OF S2))
           (X2 (MEMBER-OF S2)))
    (IFF
      (IS X1
          (RELATED-TO X2 R))
      (IS X1
        (RELATED-TO X2
          (RESTRICT-RELATION R S2))))))
```

```
(IN-CONTEXT
    ((LET-BE R RELATION)
     (LET-BE S (RULE-DOMAIN R))
     (LET-BE S2 (SUBSET-OF S))
     (LET-BE R2 (RESTRICT-RELATION R S2)))

  (IN-CONTEXT
      ((PUSH-GOAL (IS R2 (RELATION-ON S2))))
    (IN-CONTEXT
        ((SUPPOSE
           (EXISTS-SOME (MEMBER-OF S2)))
         (LET-BE X (MEMBER-OF S2))
         (LET-BE S3 (APPLY-RULE R X))
         (LET-BE S4 (APPLY-RULE R2 X)))
      (NOTE-GOAL))
    (IN-CONTEXT
        ((SUPPOSE
           (NOT
             (EXISTS-SOME (MEMBER-OF S2))))
         (LET-BE P (POWER-SET S2)))
      (NOTE-GOAL))
    (NOTE-GOAL))

  (IN-CONTEXT
      ((PUSH-GOAL
         (FORALL ((X (MEMBER-OF S2))
                  (Y (MEMBER-OF S2)))
           (IFF (IS X (RELATED-TO Y R))
                (IS X (RELATED-TO Y R2))))))
    (IN-CONTEXT
        ((SUPPOSE
           (EXISTS-SOME (MEMBER-OF S2)))
         (LET-BE X (MEMBER-OF S2))
         (LET-BE Y (MEMBER-OF S2))
         (LET-BE SR (APPLY-RULE R Y))
         (LET-BE SR2 (APPLY-RULE R2 Y)))
      (IN-CONTEXT
          ((SUPPOSE (IS X (RELATED-TO Y R))))
        (NOTE-GOAL))
      (NOTE-GOAL))
    (NOTE-GOAL)))
```

```
(LEMMA
  (FORALL ((W RELATION-STRUCTURE))
    (EXISTS-SOME
       (NON-EMPTY-SUBSET-OF
              (U-SET W)))))

(LEMMA
  (FORALL ((W RELATION-STRUCTURE)
           (S2 (NON-EMPTY-SUBSET-OF
                 (U-SET W))))
    (IS S2 NON-EMPTY-SET)))

(LEMMA
  (FORALL ((W RELATION-STRUCTURE)
           (S2 (NON-EMPTY-SUBSET-OF
                 (U-SET W))))
    (IS (RESTRICT-RELATION
          (GET-RELATION W)
          S2)
        (RELATION-ON S2))))

(LEMMA
  (FORALL ((W RELATION-STRUCTURE)
           (S2 (NON-EMPTY-SUBSET-OF
                 (U-SET W)))
           (X1 (MEMBER-OF S2))
           (X2 (MEMBER-OF S2)))
    (IFF
      (IS X1
        (RELATED-TO X2
                    (GET-RELATION W)))
      (IS X1
        (RELATED-TO X2
          (RESTRICT-RELATION
            (GET-RELATION W)
            S2))))))
```

```
(IN-CONTEXT ((LET-BE W RELATION-STRUCTURE)
             (LET-BE S (U-SET W)))
  (NOTE
    (EXISTS-SOME
       (NON-EMPTY-SUBSET-OF (U-SET W))))
  (IN-CONTEXT
      ((LET-BE S2 (NON-EMPTY-SUBSET-OF S)))
    (NOTE (IS S2 NON-EMPTY-SET))
    (IN-CONTEXT
        ((LET-BE R
                  (RESTRICT-RELATION
                    (GET-RELATION W)
                    S2))
         (LET-BE R2 (GET-RELATION W)))

      (NOTE (IS (RESTRICT-RELATION
                  (GET-RELATION W)
                  S2)
                (RELATION-ON S2)))
      (NOTE
        (FORALL ((?:X (MEMBER-OF S2))
                 (X (MEMBER-OF S2)))
          (IFF
            (IS X
                (RELATED-TO ?:X (GET-RELATION W)))
            (IS X
                (RELATED-TO ?:X
                            (RESTRICT-RELATION
                              (GET-RELATION W)
                              S2)))))))))
```

```
(LEMMA
  (FORALL ((W RELATION-STRUCTURE)
           (S2 (NON-EMPTY-SUBSET-OF
                 (U-SET W))))
    (IS (RESTRICT-RELATION-STRUCTURE
          W
          S2)
        RELATION-STRUCTURE)))
```

```
(IN-CONTEXT
    ((LET-BE W RELATION-STRUCTURE)
     (LET-BE S (U-SET W))
     (LET-BE S2 (NON-EMPTY-SUBSET-OF S))
     (LET-BE R
       (RESTRICT-RELATION (GET-RELATION W) S2))
     (LET-BE W2
       (RESTRICT-RELATION-STRUCTURE W S2)))

  (NOTE (IS (RESTRICT-RELATION-STRUCTURE W S2)
            RELATION-STRUCTURE))
  (NOTE (= (GET-RELATION
             (RESTRICT-RELATION-STRUCTURE W S2))
           (RESTRICT-RELATION
             (GET-RELATION W)
              S2)))
  (NOTE (= (U-SET
             (RESTRICT-RELATION-STRUCTURE W S2))
           S2)))
```

```
(LEMMA
  (FORALL ((W RELATION-STRUCTURE)
           (S2 (NON-EMPTY-SUBSET-OF
                 (U-SET W))))
    (= (GET-RELATION
         (RESTRICT-RELATION-STRUCTURE
           W
           S2))
       (RESTRICT-RELATION
         (GET-RELATION W) S2))))
```

```
(LEMMA
  (FORALL ((W RELATION-STRUCTURE)
           (S2 (NON-EMPTY-SUBSET-OF
                 (U-SET W))))
    (= (U-SET
         (RESTRICT-RELATION-STRUCTURE
           W
           S2))
       S2)))
```

```
(LEMMA
  (FORALL
     ((W RELATION-STRUCTURE)
      (S2 (NON-EMPTY-SUBSET-OF
            (U-SET W)))
      (X (IN-U-SET
           (RESTRICT-RELATION-STRUCTURE
             W
             S2))))

    (IS X (IN-U-SET W))))
```

```
(IN-CONTEXT
    ((LET-BE W RELATION-STRUCTURE)
     (LET-BE S2
             (NON-EMPTY-SUBSET-OF (U-SET W)))
     (LET-BE W2
             (RESTRICT-RELATION-STRUCTURE W S2))
     (LET-BE X (IN-U-SET W2))
     (LET-BE S (U-SET W)))
  (NOTE (IS X (IN-U-SET W))))
```

```
(DEFTYPE (RIGHT-ADJACENT (X (IN-U-SET R))
                         (R RELATION-STRUCTURE))
  (RELATED-TO X (GET-RELATION R)))
```

```
(LEMMA
  (FORALL ((W RELATION-STRUCTURE)
           (X (IN-U-SET W))
           (Y (RIGHT-ADJACENT X W)))
    (IS Y (IN-U-SET W))))
```

```
(IN-CONTEXT
    ((LET-BE W RELATION-STRUCTURE)
     (LET-BE X (IN-U-SET W))
     (PUSH-GOAL (IS-EVERY (RIGHT-ADJACENT X W)
                          (IN-U-SET W))))
  (IN-CONTEXT
      ((SUPPOSE
         (EXISTS-SOME (RIGHT-ADJACENT X W)))
       (LET-BE Y (RIGHT-ADJACENT X W))
       (LET-BE S (U-SET W))
       (LET-BE R (GET-RELATION W)))
    (NOTE-GOAL))
  (NOTE-GOAL))
```

```
(DEFTYPE (LEFT-ADJACENT (Y (IN-U-SET R))
                        (R RELATION-STRUCTURE))
  (LAMBDA ((X (IN-U-SET R)))
    (IS Y (RIGHT-ADJACENT X R))))


(DEFTYPE (REFLEXIVE-RELATION-ON (S SET))
  (LAMBDA ((R (RELATION-ON S)))
    (FORALL ((X (MEMBER-OF S)))
      (IS X (RELATED-TO X R)))))

(DEFTYPE (IRREFLEXIVE-RELATION-ON (S SET))
  (LAMBDA ((R (RELATION-ON S)))
    (FORALL ((X (MEMBER-OF S)))
      (NOT (IS X (RELATED-TO X R))))))

(DEFTYPE (SYMMETRIC-RELATION-ON (S SET))
  (LAMBDA ((R (RELATION-ON S)))
    (FORALL ((X (MEMBER-OF S))
             (Y (MEMBER-OF S)))
      (IFF (IS X (RELATED-TO Y R))
           (IS Y (RELATED-TO X R))))))

(DEFTYPE (ANTISYMMETRIC-RELATION-ON (S SET))
  (LAMBDA ((R (RELATION-ON S)))
    (FORALL ((X (MEMBER-OF S))
             (Y (OTHER-MEMBER S X)))
      (NOT (AND (IS X (RELATED-TO Y R))
                (IS Y (RELATED-TO X R)))))))

(DEFTYPE (TRANSITIVE-RELATION-ON (S SET))
  (LAMBDA ((R (RELATION-ON S)))
    (FORALL ((X (MEMBER-OF S))
             (Y (RELATED-TO X R)))
      (IS-EVERY (RELATED-TO Y R) (RELATED-TO X R)))))
```

```
(DEFTYPE (EQUIVALENCE-RELATION-ON (S SET))
  (AND-TYPE (SYMMETRIC-RELATION-ON S)
            (TRANSITIVE-RELATION-ON S)
            (REFLEXIVE-RELATION-ON S)))

(DEFTYPE EQUIVALENCE-RELATION
  (WRITABLE-AS R
    (R (EQUIVALENCE-RELATION-ON S))
    (S SET)))

(DEFTERM (THE-TOTAL-RELATION-ON (S SET))
  (THE-RULE ((X (MEMBER-OF S))) S))
```

```
(LEMMA
  (FORALL ((S NON-EMPTY-SET))
    (IS (THE-TOTAL-RELATION-ON S)
        (EQUIVALENCE-RELATION-ON S))))
```

```
(IN-CONTEXT
    ((LET-BE S NON-EMPTY-SET)
     (LET-BE R (THE-TOTAL-RELATION-ON S))
     (PUSH-GOAL
       (IS R (EQUIVALENCE-RELATION-ON S))))
  (IN-CONTEXT ((LET-BE X (MEMBER-OF S)))
    (NOTE (IS R (REFLEXIVE-RELATION-ON S)))
    (IN-CONTEXT ((LET-BE Y (MEMBER-OF S)))
      (NOTE (IS R (SYMMETRIC-RELATION-ON S))))
    (IN-CONTEXT ((LET-BE Y (RELATED-TO X R)))
      (NOTE (IS R TRANSITIVE-RELATION-ON S))))
  (NOTE-GOAL))
```

```
(LEMMA
 (FORALL
     ((S NON-EMPTY-SET)
      (R (TRANSITIVE-RELATION-ON S)))
   (=>
     (IS R
       (IRREFLEXIVE-RELATION-ON S))
     (IS R
       (ANTISYMMETRIC-RELATION-ON S)))))
```

```
(IN-CONTEXT
    ((LET-BE S NON-EMPTY-SET)
     (LET-BE R (TRANSITIVE-RELATION-ON S))
     (SUPPOSE
       (IS R (IRREFLEXIVE-RELATION-ON S)))
     (PUSH-GOAL
       (IS R (ANTISYMMETRIC-RELATION-ON S))))
  (IN-CONTEXT ((LET-BE X (MEMBER-OF S)))
    (IN-CONTEXT
        ((PUSH-GOAL
           (FORALL ((Y (OTHER-MEMBER S X)))
             (NOT (AND (IS X (RELATED-TO Y R))
                       (IS Y (RELATED-TO X R)))))))
      (IN-CONTEXT
          ((SUPPOSE
             (EXISTS-SOME (OTHER-MEMBER S X)))
           ;the above supposition constrains x
           ;and prevents full generalization
           (LET-BE Y (OTHER-MEMBER S X)))
        (NOTE (NOT (AND (IS X (RELATED-TO Y R))
                        (IS Y (RELATED-TO X R)))))
        (NOTE-GOAL))
      (NOTE-GOAL))
    (NOTE-GOAL)))
```

A.5 Partial Orders and Zorn's Lemma

A partial order is defined here as a transitive irreflexive relation (every such relation is also antisymmetric). A poset (partially ordered set) is a relation structure whose relation is a partial order on the underlying set. Given a poset p and an element x of the underlying set of p the types (LESS-THAN x p) and (LESS-OR-EQUAL-TO x p) are defined in the obvious way. A total order is a partial order in which every two elements are ordered.

Let p be a poset, s a subset of the underlying set of p, and x an element of the underlying set of p. We say that x is a maximial element of s if it is an element of s and no element of s is greater than x. We say that x is the greatest member of s if it is a member of s and all members of s are less than or equal to x. We say that x is an upper bound of s is every member of s is less than or equal to x. The notions of minimal member, least member, and lower bound are defined similarly. We say that x is a least upper bound of s if it is the least member of the set of all upper bounds of s; greatest lower bounds are defined similarly.

A *chain* in a poset p is a subset s of p which is totally ordered by order relation of p. An inductive order is a partial order in which every chain has an upper bound. Zorn's lemma states that if p is an inductive order and x is a member of the underlying set of p then there is a maximal member of p which is greater than or equal to x. Zorn's lemma can be proven from the axiom of choice but we take it as an axiom.

```
(DEFTYPE (PARTIAL-ORDER-ON (S SET))
  (AND-TYPE (TRANSITIVE-RELATION-ON S)
            (IRREFLEXIVE-RELATION-ON S)))

(DEFTERM (THE-EMPTY-RELATION-ON (S SET))
  (THE-RULE ((X (MEMBER-OF S)))
    THE-EMPTY-SET))
```

```
(LEMMA
  (FORALL ((S NON-EMPTY-SET))
    (IS (THE-EMPTY-RELATION-ON S)
        (PARTIAL-ORDER-ON S))))
```

```
(IN-CONTEXT
  ((LET-BE S NON-EMPTY-SET)
   (LET-BE R (THE-EMPTY-RELATION-ON S))
   (PUSH-GOAL (IS R (PARTIAL-ORDER-ON S))))
  (IN-CONTEXT
      ((LET-BE X (MEMBER-OF S)))
    (IN-CONTEXT
        ((LET-BE S2 (APPLY-RULE R X)))
      (NOTE (IS R (RELATION-ON S))))
    (NOTE-GOAL)))
```

```
(DEFTYPE POSET
  (LAMBDA ((S RELATION-STRUCTURE))
    (IS (GET-RELATION S)
        (PARTIAL-ORDER-ON (U-SET S)))))
```

```
(LEMMA (EXISTS-SOME POSET))
```

```
(IN-CONTEXT
    ((LET-BE S NON-EMPTY-SET)
     (LET-BE R (PARTIAL-ORDER-ON S))
     (LET-BE W (MAKE-RELATION-STRUCTURE R S)))
  (NOTE (EXISTS-SOME POSET)))
```

```
(DEFTYPE (LESS-THAN (X (IN-U-SET W)) (W POSET))
  (LEFT-ADJACENT X W))
```

```
(LEMMA
  (FORALL ((P POSET)
           (X (IN-U-SET P)))
    (NOT (IS X
             (LESS-THAN X P)))))

(LEMMA
  (FORALL ((P POSET)
           (X (IN-U-SET P))
           (Y (IN-U-SET P)))
    (NOT
      (AND
        (IS X
            (LESS-THAN Y P))
        (IS Y
            (LESS-THAN X P))))))

(LEMMA
  (FORALL ((P POSET)
           (X (IN-U-SET P))
           (Y (LESS-THAN X P))
           (Z (LESS-THAN Y P)))
    (IS Z (LESS-THAN X P))))
```

```
(IN-CONTEXT
    ((LET-BE P POSET)
     (LET-BE X (IN-U-SET P)))
  (IN-CONTEXT
      ((PUSH-GOAL
         (NOT (IS X  (LESS-THAN X P))))
       (LET-BE R (GET-RELATION P))
       (LET-BE S (U-SET P)))
    (NOTE-GOAL))
  (IN-CONTEXT
      ((LET-BE Y (IN-U-SET P))
       (PUSH-GOAL
         (NOT
           (AND (IS X (LESS-THAN Y P))
                (IS Y (LESS-THAN X P)))))
       (LET-BE R (GET-RELATION P))
       (LET-BE S (U-SET P)))
    (NOTE-GOAL))
  (IN-CONTEXT
      ((PUSH-GOAL
         (FORALL ((Y (LESS-THAN X P)))
           (IS-EVERY (LESS-THAN Y P)
                     (LESS-THAN X P)))))
    (IN-CONTEXT
        ((SUPPOSE
           (EXISTS-SOME (LESS-THAN X P)))
         (LET-BE Y (LESS-THAN X P)))
      (IN-CONTEXT
          ((SUPPOSE
             (EXISTS-SOME (LESS-THAN Y P)))
           (LET-BE Z (LESS-THAN Y P))
           (LET-BE R (GET-RELATION P))
           (LET-BE S (U-SET P)))
        (NOTE (IS-EVERY (LESS-THAN Y P)
                        (LESS-THAN X P))))
      (NOTE (IS-EVERY (LESS-THAN Y P)
                      (LESS-THAN X P)))
      (NOTE-GOAL))
    (NOTE-GOAL)))
```

```
(DEFTYPE (GREATER-THAN (X (IN-U-SET W)) (W POSET))
  (RIGHT-ADJACENT X W))
```

```
(LEMMA
  (FORALL ((P POSET)
           (X (IN-U-SET P))
           (Y (GREATER-THAN X P)))
    (IS X (LESS-THAN Y P))))
```

```
(IN-CONTEXT
    ((LET-BE P POSET)
     (LET-BE X (IN-U-SET P))
     (PUSH-GOAL
       (FORALL ((Y (GREATER-THAN X P)))
         (IS X (LESS-THAN Y P)))))
  (IN-CONTEXT
      ((SUPPOSE
         (EXISTS-SOME (GREATER-THAN X P)))
       (LET-BE Y (GREATER-THAN X P))
       (LET-BE R (GET-RELATION P))
       (LET-BE S (U-SET P)))
    (NOTE-GOAL))
  (NOTE-GOAL))
```

```
(DEFTYPE (LESS-OR-EQUAL-TO (X (IN-U-SET W)) (W POSET))
  (OR-TYPE (LESS-THAN X W) (EQUAL-TO X)))
```

```
(LEMMA
  (FORALL ((P POSET)
           (X (IN-U-SET P))
           (Y (LESS-OR-EQUAL-TO X P)))
    (IS Y (IN-U-SET P))))

(LEMMA
  (FORALL ((P POSET)
           (X (IN-U-SET P))
           (Y (LESS-OR-EQUAL-TO X P))
           (Z (LESS-OR-EQUAL-TO Y P)))
    (IS Z (LESS-OR-EQUAL-TO X P))))
```

```
(IN-CONTEXT
    ((LET-BE P POSET)
     (LET-BE X (IN-U-SET P))
     (LET-BE Y (LESS-OR-EQUAL-TO X P)))
  (IN-CONTEXT
      ((PUSH-GOAL (IS Y (IN-U-SET P))))
    (IN-CONTEXT
        ((SUPPOSE (IS Y (LESS-THAN X P))))
      (NOTE-GOAL))
    (NOTE-GOAL))
  (IN-CONTEXT
      ((LET-BE Z (LESS-OR-EQUAL-TO Y P))
       (PUSH-GOAL
         (IS Z (LESS-OR-EQUAL-TO X P))))
    (IN-CONTEXT ((SUPPOSE (= Y X)))
      (NOTE-GOAL))
    (IN-CONTEXT
        ((SUPPOSE (IS Y (LESS-THAN X P))))
      (IN-CONTEXT
          ((SUPPOSE (IS Z (LESS-THAN Y P))))
        (NOTE-GOAL))
      (NOTE-GOAL))
    (NOTE-GOAL)))
```

```
(LEMMA
  (FORALL ((P POSET)
           (X (IN-U-SET P))
           (Y (LESS-OR-EQUAL-TO X P)))
    (=> (IS X (LESS-OR-EQUAL-TO Y P))
        (= X Y))))
```

```
(IN-CONTEXT
    ((LET-BE P POSET)
     (LET-BE X (IN-U-SET P))
     (LET-BE Y (LESS-OR-EQUAL-TO X P))
     (SUPPOSE
       (IS X (LESS-OR-EQUAL-TO Y P))))
  (NOTE (= X Y)))
```

```
(DEFTYPE (GREATER-OR-EQUAL-TO (X (IN-U-SET W))
                              (W POSET))
  (OR-TYPE (GREATER-THAN X W) (EQUAL-TO X)))
```

```
(LEMMA
  (FORALL
      ((P POSET)
       (X (IN-U-SET P))
       (Y (GREATER-OR-EQUAL-TO X P)))
    (IS Y (IN-U-SET P))))
```

```
(IN-CONTEXT
    ((LET-BE P POSET)
     (LET-BE X (IN-U-SET P))
     (LET-BE Y (GREATER-OR-EQUAL-TO X P))
     (PUSH-GOAL (IS Y (IN-U-SET P))))
  (IN-CONTEXT
      ((SUPPOSE (IS Y (GREATER-THAN X P))))
    (NOTE-GOAL))
  (NOTE-GOAL))
```

```
(LEMMA
 (FORALL ((P POSET)
          (Y (IN-U-SET P))
          (X (IN-U-SET P)))
   (=> (IS Y
         (LESS-OR-EQUAL-TO X P))
       (IS X
         (GREATER-OR-EQUAL-TO Y P)))))

(LEMMA
  (FORALL ((P POSET)
           (Y (IN-U-SET P))
           (X (IN-U-SET P)))
    (=> (IS Y
           (GREATER-OR-EQUAL-TO X P))
        (IS X
           (LESS-OR-EQUAL-TO Y P)))))
```

```
(IN-CONTEXT ((LET-BE P POSET)
             (LET-BE X (IN-U-SET P))
             (LET-BE Y (IN-U-SET P)))
  (IN-CONTEXT
      ((SUPPOSE
         (IS Y (LESS-OR-EQUAL-TO X P)))
       (PUSH-GOAL
         (IS X (GREATER-OR-EQUAL-TO Y P))))
    (IN-CONTEXT
        ((SUPPOSE (IS Y  (LESS-THAN X P))))
      (NOTE-GOAL))
    (NOTE-GOAL))
  (IN-CONTEXT
      ((SUPPOSE
         (IS Y (GREATER-OR-EQUAL-TO X P)))
       (PUSH-GOAL
         (IS X (LESS-OR-EQUAL-TO Y P))))
    (IN-CONTEXT
        ((SUPPOSE
            (IS Y (GREATER-THAN X P))))
      (NOTE-GOAL))
    (NOTE-GOAL)))
```

```
(DEFTERM (RESTRICT-ORDER
           (O POSET)
           (S (NON-EMPTY-SUBSET-OF
                (U-SET O))))
  (RESTRICT-RELATION-STRUCTURE O S))
```

```
(LEMMA
  (FORALL
      ((S1 NON-EMPTY-SET)
       (R1 (TRANSITIVE-RELATION-ON S1))
       (S2 (SUBSET-OF S1)))
    (IS (RESTRICT-RELATION R1 S2)
        (TRANSITIVE-RELATION-ON S2))))
```

```
(IN-CONTEXT
    ((LET-BE S1 NON-EMPTY-SET)
     (LET-BE R1 (TRANSITIVE-RELATION-ON S1))
     (LET-BE S2 (SUBSET-OF S1))
     (LET-BE R2 (RESTRICT-RELATION R1 S2))
     (PUSH-GOAL
       (IS R2 (TRANSITIVE-RELATION-ON S2))))
  (IN-CONTEXT
      ((SUPPOSE
          (EXISTS-SOME (MEMBER-OF S2)))
       (LET-BE X (MEMBER-OF S2)))
    (IN-CONTEXT
        ((PUSH-GOAL
           (FORALL ((Y (RELATED-TO X R2)))
             (IS-EVERY (RELATED-TO Y R2)
                       (RELATED-TO X R2)))))
      (IN-CONTEXT ((SUPPOSE
                     (EXISTS-SOME
                       (RELATED-TO X R2)))
                   (LET-BE Y (RELATED-TO X R2)))
        (IN-CONTEXT
            ((PUSH-GOAL
               (IS-EVERY (RELATED-TO Y R2)
                         (RELATED-TO X R2))))
          (IN-CONTEXT
              ((SUPPOSE
                  (EXISTS-SOME
                     (RELATED-TO Y R2)))
               (LET-BE Z (RELATED-TO Y R2)))
            (NOTE-GOAL))
          (NOTE-GOAL))
        (NOTE-GOAL))
      (NOTE-GOAL))
    (NOTE-GOAL))
  (NOTE-GOAL))
```

```
(LEMMA
  (FORALL ((S SET))
    (EXISTS-SOME
      (IRREFLEXIVE-RELATION-ON S))))
```

```
(IN-CONTEXT
    ((LET-BE S SET)
     (LET-BE R (THE-EMPTY-RELATION-ON S)))
  (NOTE
    (EXISTS-SOME
      (IRREFLEXIVE-RELATION-ON S))))
```

```
(LEMMA
  (FORALL
      ((S1 NON-EMPTY-SET)
       (R1 (IRREFLEXIVE-RELATION-ON S1))
       (S2 (SUBSET-OF S1)))
    (IS (RESTRICT-RELATION R1 S2)
        (IRREFLEXIVE-RELATION-ON S2))))
```

```
(IN-CONTEXT
    ((LET-BE S1 NON-EMPTY-SET)
     (LET-BE R1 (IRREFLEXIVE-RELATION-ON S1))
     (LET-BE S2 (SUBSET-OF S1))
     (LET-BE R2 (RESTRICT-RELATION R1 S2))
     (PUSH-GOAL
       (IS R2 (IRREFLEXIVE-RELATION-ON S2))))
  (IN-CONTEXT
      ((SUPPOSE
          (EXISTS-SOME (MEMBER-OF S2)))
       (LET-BE X (MEMBER-OF S2)))
    (NOTE-GOAL))
  (NOTE-GOAL))
```

```
(LEMMA
  (FORALL ((P POSET)
           (S2 (NON-EMPTY-SUBSET-OF
                 (U-SET P))))
    (IS (RESTRICT-ORDER P S2)
        POSET)))

(LEMMA
  (FORALL ((P POSET)
           (S2 (NON-EMPTY-SUBSET-OF
                 (U-SET P)))
           (X (IN-U-SET
                (RESTRICT-ORDER P S2))))
    (IS X (IN-U-SET P))))

(LEMMA
  (FORALL ((P POSET)
           (S2 (NON-EMPTY-SUBSET-OF
                 (U-SET P)))
           (Y (IN-U-SET
                (RESTRICT-ORDER P S2)))
           (X (IN-U-SET
                (RESTRICT-ORDER P S2))))
    (IFF
      (IS X
        (LESS-THAN Y
          (RESTRICT-ORDER P S2)))
      (IS X
        (LESS-THAN Y P)))))
```

```
(IN-CONTEXT
    ((LET-BE P POSET)
     (LET-BE S2 (NON-EMPTY-SUBSET-OF (U-SET P)))
     (LET-BE P2 (RESTRICT-ORDER P S2)))
  (IN-CONTEXT
      ((LET-BE S1 (U-SET P))
       (LET-BE R1 (GET-RELATION P))
       (LET-BE R2 (GET-RELATION P2)))
    (NOTE (IS P2 POSET))
    (IN-CONTEXT ((LET-BE X (IN-U-SET P2))
                 (LET-BE Y (IN-U-SET P2)))
      (NOTE (IS X (IN-U-SET P)))
      (NOTE
        (IFF
          (IS X (LESS-THAN Y P2))
          (IS X (LESS-THAN Y P)))))))
```

```
(LEMMA
  (FORALL ((P POSET)
           (S2 (NON-EMPTY-SUBSET-OF
                 (U-SET P)))
           (Y (IN-U-SET
                (RESTRICT-ORDER P S2)))
           (X (IN-U-SET
                (RESTRICT-ORDER P S2))))
    (IFF
      (IS X
        (LESS-OR-EQUAL-TO Y
          (RESTRICT-ORDER P S2)))
      (IS X
        (LESS-OR-EQUAL-TO Y P)))))
```

```
(IN-CONTEXT
    ((LET-BE P POSET)
     (LET-BE S2 (NON-EMPTY-SUBSET-OF
                  (U-SET P)))
     (LET-BE P2 (RESTRICT-ORDER P S2)))
    (IN-CONTEXT
        ((LET-BE X (IN-U-SET P2))
         (LET-BE Y (IN-U-SET P2)))
      (IN-CONTEXT
          ((PUSH-GOAL
             (IFF (IS X (LESS-OR-EQUAL-TO Y P2))
                  (IS X (LESS-OR-EQUAL-TO Y P)))))
        (IN-CONTEXT
            ((SUPPOSE
               (IS X (LESS-OR-EQUAL-TO Y P2))))
          (IN-CONTEXT ((SUPPOSE (= X Y)))
            (NOTE-GOAL))
          (NOTE-GOAL))
        (IN-CONTEXT
            ((SUPPOSE
               (IS X (LESS-OR-EQUAL-TO Y P))))
          (IN-CONTEXT ((SUPPOSE (= X Y)))
            (NOTE-GOAL))
          (NOTE-GOAL))
        (NOTE-GOAL))))
```

```
(DEFTYPE (TOTAL-ORDER-ON (S SET))
  (LAMBDA ((R (PARTIAL-ORDER-ON S)))
    (FORALL ((X (MEMBER-OF S))
             (Y (MEMBER-OF S)))
      (OR (= X Y)
          (IS X (RELATED-TO Y R))
          (IS Y (RELATED-TO X R))))))

(DEFTYPE TOTALLY-ORDERED-SET
  (LAMBDA ((S RELATION-STRUCTURE))
    (IS (GET-RELATION S)
        (TOTAL-ORDER-ON (U-SET S)))))
```

```
(LEMMA (EXISTS-SOME                      (IN-CONTEXT
TOTALLY-ORDERED-SET))                        ((LET-BE S SINGLETON-SET)
                                              (LET-BE R (THE-EMPTY-RELATION-ON S))
                                              (LET-BE W (MAKE-RELATION-STRUCTURE R S)))
                                           (NOTE (EXISTS-SOME TOTALLY-ORDERED-SET)))
```

```
(LEMMA                                   (IN-CONTEXT
  (FORALL ((W TOTALLY-ORDERED-SET))          ((LET-BE W TOTALLY-ORDERED-SET)
    (IS W POSET)))                            (PUSH-GOAL (IS W POSET))
                                              (LET-BE R (GET-RELATION W))
                                              (LET-BE S (U-SET W)))
                                           (NOTE-GOAL))
```

```
(LEMMA                                   (IN-CONTEXT
  (FORALL ((W TOTALLY-ORDERED-SET)           ((LET-BE W TOTALLY-ORDERED-SET)
           (X (IN-U-SET W))                   (LET-BE X (IN-U-SET W))
           (Y (IN-U-SET W)))                  (LET-BE Y (IN-U-SET W))
    (OR (IS X                                 (LET-BE R (GET-RELATION W))
            (LESS-OR-EQUAL-TO Y W))           (LET-BE S (U-SET W)))
        (IS Y                              (IN-CONTEXT
            (LESS-OR-EQUAL-TO X W)))))        ((PUSH-GOAL
                                                 (OR (IS X (LESS-OR-EQUAL-TO Y W))
                                                     (IS Y (LESS-OR-EQUAL-TO X W)))))
                                            (IN-CONTEXT ((SUPPOSE (= X Y)))
                                              (NOTE-GOAL))
                                            (IN-CONTEXT ((SUPPOSE (IS X (LESS-THAN Y W))))
                                              (NOTE-GOAL))
                                            (NOTE-GOAL)))
```

```
(DEFTYPE (MINIMAL-ELEMENT-OF (W POSET))
  (LAMBDA ((X (IN-U-SET W)))
    (NOT (EXISTS-SOME (LESS-THAN X W)))))

(DEFTYPE (MAXIMAL-ELEMENT-OF (W POSET))
  (LAMBDA ((X (IN-U-SET W)))
    (NOT (EXISTS-SOME (GREATER-THAN X W)))))

(DEFTYPE (UPPER-BOUND-OF (S (SUBSET-OF (U-SET W)))
                         (W POSET))
  (LAMBDA ((A (IN-U-SET W)))
    (IS-EVERY (MEMBER-OF S)
              (LESS-OR-EQUAL-TO A W))))
```

```
(LEMMA
  (FORALL ((W POSET)
           (X (IN-U-SET W))
           (Y (IN-U-SET W)))

    (IS-EVERY
      (AND-TYPE
        (GREATER-OR-EQUAL-TO X W)
        (GREATER-OR-EQUAL-TO Y W))
      (UPPER-BOUND-OF (MAKE-SET X Y)
                      W))))
```

```
(IN-CONTEXT
    ((LET-BE W POSET)
     (LET-BE X (IN-U-SET W))
     (LET-BE Y (IN-U-SET W))
     (PUSH-GOAL
       (IS-EVERY
         (AND-TYPE (GREATER-OR-EQUAL-TO X W)
                   (GREATER-OR-EQUAL-TO Y W))
         (UPPER-BOUND-OF (MAKE-SET X Y) W))))
  (IN-CONTEXT
      ((SUPPOSE
         (EXISTS-SOME
           (AND-TYPE
             (GREATER-OR-EQUAL-TO X W)
             (GREATER-OR-EQUAL-TO Y W))))
       (LET-BE Z (AND-TYPE
                   (GREATER-OR-EQUAL-TO X W)
                   (GREATER-OR-EQUAL-TO Y W))))
    (IN-CONTEXT
        ((PUSH-GOAL
           (IS Z
             (UPPER-BOUND-OF (MAKE-SET X Y) W))))

      (IN-CONTEXT ((LET-BE S (MAKE-SET X Y))
                   (LET-BE Z2 (MEMBER-OF S)))
        (IN-CONTEXT
            ((PUSH-GOAL
               (IS Z (GREATER-OR-EQUAL-TO Z2 W))))
          (IN-CONTEXT ((SUPPOSE (= Z2 X)))
            (NOTE-GOAL))
          (NOTE-GOAL))
        (NOTE-GOAL))
      (NOTE-GOAL))
    (NOTE-GOAL))
  (NOTE-GOAL))
```

```
(LEMMA
  (FORALL ((W POSET)
           (Y (IN-U-SET W))
           (X (IN-U-SET W)))
    (IS-EVERY
      (UPPER-BOUND-OF (MAKE-SET X Y) W)
      (GREATER-OR-EQUAL-TO X W))))
```

```
(IN-CONTEXT
    ((LET-BE W POSET)
     (LET-BE X (IN-U-SET W))
     (LET-BE Y (IN-U-SET W))
     (LET-BE S (MAKE-SET X Y))
     (PUSH-GOAL
       (IS-EVERY (UPPER-BOUND-OF S W)
                 (GREATER-OR-EQUAL-TO X W))))
  (IN-CONTEXT
      ((SUPPOSE
         (EXISTS-SOME (UPPER-BOUND-OF S W)))
       (LET-BE Z (UPPER-BOUND-OF S W)))
    (NOTE-GOAL))
  (NOTE-GOAL))
```

```
(LEMMA
  (FORALL ((P POSET)
           (X (IN-U-SET P))
           (S (SUBSET-OF (U-SET P))))
    (IS-EVERY
      (AND-TYPE
        (GREATER-OR-EQUAL-TO X P)
        (UPPER-BOUND-OF S P))
      (UPPER-BOUND-OF
        (INSERT X S)
        P))))
```

```
(IN-CONTEXT
    ((LET-BE P POSET)
     (LET-BE S (SUBSET-OF (U-SET P)))
     (LET-BE X (IN-U-SET P))
     (PUSH-GOAL
       (IS-EVERY
         (AND-TYPE (GREATER-OR-EQUAL-TO X P)
                   (UPPER-BOUND-OF S P))
         (UPPER-BOUND-OF (INSERT X S) P))))
  (IN-CONTEXT
      ((SUPPOSE
         (EXISTS-SOME
           (AND-TYPE (GREATER-OR-EQUAL-TO X P)
                     (UPPER-BOUND-OF S P))))
       (LET-BE Y
         (AND-TYPE (GREATER-OR-EQUAL-TO X P)
                   (UPPER-BOUND-OF S P))))
    (IN-CONTEXT
        ((PUSH-GOAL
           (IS Y (UPPER-BOUND-OF (INSERT X S) P))))
      (IN-CONTEXT ((LET-BE S2 (INSERT X S))
                   (LET-BE Z (MEMBER-OF S2)))
        (IN-CONTEXT
            ((PUSH-GOAL
               (IS Y (GREATER-OR-EQUAL-TO Z P))))
          (IN-CONTEXT ((SUPPOSE (= Z X)))
            (NOTE-GOAL))
          (NOTE-GOAL))
        (NOTE-GOAL)))
    (NOTE-GOAL))
  (NOTE-GOAL))
```

```
(DEFTYPE (LOWER-BOUND-OF (S (SUBSET-OF (U-SET W)))
                         (W POSET))
  (LAMBDA ((A (IN-U-SET W)))
    (IS-EVERY (MEMBER-OF S) (GREATER-OR-EQUAL-TO A W))))
```

```
(LEMMA
  (FORALL ((W POSET)
           (X (IN-U-SET W))
           (Y (IN-U-SET W)))
    (IS-EVERY
      (AND-TYPE
        (LESS-OR-EQUAL-TO X W)
        (LESS-OR-EQUAL-TO Y W))
      (LOWER-BOUND-OF
        (MAKE-SET X Y)
        W))))
```

```
(IN-CONTEXT
    ((LET-BE W POSET)
     (LET-BE X (IN-U-SET W))
     (LET-BE Y (IN-U-SET W))
     (PUSH-GOAL
       (IS-EVERY
         (AND-TYPE (LESS-OR-EQUAL-TO X W)
                   (LESS-OR-EQUAL-TO Y W))
         (LOWER-BOUND-OF
           (MAKE-SET X Y)
           W))))

  (IN-CONTEXT
      ((SUPPOSE
         (EXISTS-SOME
           (AND-TYPE
             (LESS-OR-EQUAL-TO X W)
             (LESS-OR-EQUAL-TO Y W))))
       (LET-BE Z (AND-TYPE
                   (LESS-OR-EQUAL-TO X W)
                   (LESS-OR-EQUAL-TO Y W))))
    (IN-CONTEXT
        ((PUSH-GOAL
           (IS Z
             (LOWER-BOUND-OF
                (MAKE-SET X Y)
                W))))

      (IN-CONTEXT ((LET-BE S (MAKE-SET X Y))
                   (LET-BE Z2 (MEMBER-OF S)))
        (IN-CONTEXT
            ((PUSH-GOAL
               (IS Z (LESS-OR-EQUAL-TO Z2 W))))
          (IN-CONTEXT ((SUPPOSE (= Z2 X)))
            (NOTE-GOAL))
          (NOTE-GOAL))
        (NOTE-GOAL))
      (NOTE-GOAL))
    (NOTE-GOAL))
  (NOTE-GOAL))
```

```
(LEMMA
  (FORALL ((W POSET)
           (Y (IN-U-SET W))
           (X (IN-U-SET W)))
    (IS-EVERY
      (LOWER-BOUND-OF (MAKE-SET X Y) W)
      (LESS-OR-EQUAL-TO X W))))
```

```
(IN-CONTEXT
    ((LET-BE W POSET)
     (LET-BE X (IN-U-SET W))
     (LET-BE Y (IN-U-SET W))
     (LET-BE S (MAKE-SET X Y))
     (PUSH-GOAL
       (IS-EVERY (LOWER-BOUND-OF S W)
                 (LESS-OR-EQUAL-TO X W))))
  (IN-CONTEXT
      ((SUPPOSE
         (EXISTS-SOME (LOWER-BOUND-OF S W)))
       (LET-BE Z (LOWER-BOUND-OF S W)))
    (NOTE-GOAL))
  (NOTE-GOAL))
```

```
(LEMMA
  (FORALL ((P POSET)
           (X (IN-U-SET P))
           (S (SUBSET-OF (U-SET P))))
    (IS-EVERY
      (AND-TYPE
        (LESS-OR-EQUAL-TO X P)
        (LOWER-BOUND-OF S P))
      (LOWER-BOUND-OF
        (INSERT X S)
        P))))
```

```
(IN-CONTEXT
    ((LET-BE P POSET)
     (LET-BE S (SUBSET-OF (U-SET P)))
     (LET-BE X (IN-U-SET P))
     (PUSH-GOAL
       (IS-EVERY
         (AND-TYPE (LESS-OR-EQUAL-TO X P)
                   (LOWER-BOUND-OF S P))
         (LOWER-BOUND-OF (INSERT X S) P))))
  (IN-CONTEXT
      ((SUPPOSE
         (EXISTS-SOME
           (AND-TYPE (LESS-OR-EQUAL-TO X P)
                     (LOWER-BOUND-OF S P))))
       (LET-BE Y
         (AND-TYPE (LESS-OR-EQUAL-TO X P)
                   (LOWER-BOUND-OF S P))))
    (IN-CONTEXT
        ((PUSH-GOAL
           (IS Y (LOWER-BOUND-OF (INSERT X S) P))))
      (IN-CONTEXT ((LET-BE S2 (INSERT X S))
                   (LET-BE Z (MEMBER-OF S2)))
        (IN-CONTEXT
            ((PUSH-GOAL
               (IS Y (LESS-OR-EQUAL-TO Z P))))
          (IN-CONTEXT ((SUPPOSE (= Z X)))
            (NOTE-GOAL))
          (NOTE-GOAL))
        (NOTE-GOAL)))
    (NOTE-GOAL))
  (NOTE-GOAL))
```

```
(DEFTYPE (LEAST-MEMBER-OF (S (SUBSET-OF (U-SET W)))
                          (W POSET))
  (LAMBDA ((X (MEMBER-OF S)))
    (IS-EVERY (MEMBER-OF S)
              (GREATER-OR-EQUAL-TO X W))))
```

```
(LEMMA
  (FORALL ((W POSET)
           (S (SUBSET-OF-U-SET W)))
    (IS-EVERY (LEAST-MEMBER-OF S W)
              (IN-U-SET W))))

(LEMMA
  (FORALL ((W POSET)
           (S (SUBSET-OF (U-SET W))))
    (AT-MOST-ONE
      (LEAST-MEMBER-OF S W))))
```

```
(IN-CONTEXT
    ((LET-BE W POSET)
     (LET-BE S (SUBSET-OF-U-SET W)))
  (IN-CONTEXT
      ((PUSH-GOAL
          (IS-EVERY (LEAST-MEMBER-OF S W)
                    (IN-U-SET W))))
    (IN-CONTEXT
        ((SUPPOSE
           (EXISTS-SOME
             (LEAST-MEMBER-OF S W)))
          (LET-BE X (LEAST-MEMBER-OF S W))
          (LET-BE S (U-SET W)))
      (NOTE-GOAL))
    (NOTE-GOAL))
  (IN-CONTEXT
      ((PUSH-GOAL
          (AT-MOST-ONE (LEAST-MEMBER-OF S W))))
    (IN-CONTEXT
        ((SUPPOSE
           (EXISTS-SOME (LEAST-MEMBER-OF S W)))
          (LET-BE X (LEAST-MEMBER-OF S W))
          (LET-BE Y (LEAST-MEMBER-OF S W)))
      (NOTE-GOAL))
    (NOTE-GOAL)))
```

```
(DEFTYPE (GREATEST-MEMBER-OF (S (SUBSET-OF (U-SET W)))
                             (W POSET))
  (LAMBDA ((X (MEMBER-OF S)))
    (IS-EVERY (MEMBER-OF S)
              (LESS-OR-EQUAL-TO X W))))
```

```
(LEMMA
 (FORALL ((W POSET)
          (S (SUBSET-OF-U-SET W))
          (X (GREATEST-MEMBER-OF S W)))
   (IS X (IN-U-SET W))))

(LEMMA
  (FORALL ((W POSET)
           (S (SUBSET-OF (U-SET W))))
    (AT-MOST-ONE
      (GREATEST-MEMBER-OF S W))))
```

```
(IN-CONTEXT
    ((LET-BE W POSET)
     (LET-BE S (SUBSET-OF-U-SET W)))
  (IN-CONTEXT
      ((PUSH-GOAL
         (IS-EVERY (GREATEST-MEMBER-OF S W)
                   (IN-U-SET W))))
    (IN-CONTEXT
        ((SUPPOSE
           (EXISTS-SOME
             (GREATEST-MEMBER-OF S W)))
         (LET-BE X (GREATEST-MEMBER-OF S W))
         (LET-BE S (U-SET W)))
      (NOTE-GOAL))
    (NOTE-GOAL))
  (IN-CONTEXT
      ((PUSH-GOAL
         (AT-MOST-ONE (GREATEST-MEMBER-OF S W))))
    (IN-CONTEXT
        ((SUPPOSE
           (EXISTS-SOME
             (GREATEST-MEMBER-OF S W)))
         (LET-BE X (GREATEST-MEMBER-OF S W))
         (LET-BE Y (GREATEST-MEMBER-OF S W)))
      (NOTE-GOAL))
    (NOTE-GOAL)))
```

```
(DEFTYPE (LEAST-UPPER-BOUND-OF (S (SUBSET-OF (U-SET W)))
                               (W POSET))
  (LEAST-MEMBER-OF
    (THE-SET-OF-ALL (UPPER-BOUND-OF S W)) W))
```

```
(LEMMA
  (FORALL ((W POSET)
           (S (SUBSET-OF (U-SET W))))
    (IS (THE-SET-OF-ALL
          (UPPER-BOUND-OF S W))
        (SUBSET-OF (U-SET W)))))

(LEMMA
  (FORALL ((W POSET)
           (S (SUBSET-OF (U-SET W))))
    (AT-MOST-ONE
      (LEAST-UPPER-BOUND-OF S W))))

(LEMMA
  (FORALL
      ((W POSET)
       (S (SUBSET-OF (U-SET W)))
       (X (LEAST-UPPER-BOUND-OF S W)))
    (IS X (UPPER-BOUND-OF S W))))

(LEMMA
  (FORALL ((W POSET)
           (S (SUBSET-OF (U-SET W))))
    (=> (EXISTS-SOME
          (UPPER-BOUND-OF S W))
        (FORALL
            ((X (UPPER-BOUND-OF S W)))
          (=>
           (IS-EVERY
             (UPPER-BOUND-OF S W)
             (GREATER-OR-EQUAL-TO X W))
           (IS X
             (LEAST-UPPER-BOUND-OF S
                W)))))))

(LEMMA
  (FORALL ((W POSET)
           (S (SUBSET-OF (U-SET W))))
    (=> (EXISTS-SOME
          (LEAST-UPPER-BOUND-OF S W))
        (FORALL
            ((Y (UPPER-BOUND-OF S W)))
          (IS-EVERY
            (LEAST-UPPER-BOUND-OF S W)
            (LESS-OR-EQUAL-TO Y W))))))
```

```
(IN-CONTEXT
    ((LET-BE W POSET)
     (LET-BE S (SUBSET-OF (U-SET W)))
     (LET-BE S2 (THE-SET-OF-ALL
                  (UPPER-BOUND-OF S W))))
  (IN-CONTEXT ((LET-BE S3 (U-SET W))
               (PUSH-GOAL (IS S2 (SUBSET-OF S3))))
    (IN-CONTEXT ((SUPPOSE
                   (EXISTS-SOME (MEMBER-OF S2)))
                 (LET-BE X (MEMBER-OF S2)))
      (NOTE-GOAL))
    (NOTE-GOAL))

  (NOTE (AT-MOST-ONE (LEAST-UPPER-BOUND-OF S W)))

  (IN-CONTEXT
      ((PUSH-GOAL
         (IS-EVERY (LEAST-UPPER-BOUND-OF S W)
                   (UPPER-BOUND-OF S W))))
    (IN-CONTEXT
        ((SUPPOSE
           (EXISTS-SOME
             (LEAST-UPPER-BOUND-OF S W)))
         (LET-BE X (LEAST-UPPER-BOUND-OF S W)))
      (NOTE-GOAL))
    (NOTE-GOAL))

  (IN-CONTEXT
      ((SUPPOSE
         (EXISTS-SOME (UPPER-BOUND-OF S W)))
       (LET-BE X (UPPER-BOUND-OF S W))
       (SUPPOSE
         (IS-EVERY (UPPER-BOUND-OF S W)
                   (GREATER-OR-EQUAL-TO X W))))
    (NOTE (IS X (LEAST-UPPER-BOUND-OF S W))))

  (IN-CONTEXT
      ((SUPPOSE
         (EXISTS-SOME
           (LEAST-UPPER-BOUND-OF S W)))
       (LET-BE X (LEAST-UPPER-BOUND-OF S W))
       (LET-BE Y (UPPER-BOUND-OF S W)))
    (NOTE (IS X (LESS-OR-EQUAL-TO Y W)))))
```

```
(LEMMA
  (FORALL ((W POSET)
           (X (IN-U-SET W))
           (Y (IN-U-SET W)))
    (AT-MOST-ONE
      (LEAST-UPPER-BOUND-OF
        (MAKE-SET X Y)
        W))))

(LEMMA
  (FORALL ((W POSET)
           (X (IN-U-SET W))
           (Y (IN-U-SET W)))
    (=>

     (EXISTS-SOME
       (LEAST-UPPER-BOUND-OF
         (MAKE-SET X Y)
         W))

     (FORALL ((Z2 (UPPER-BOUND-OF
                   (MAKE-SET X Y)
                   W)))
       (IS (THE (LEAST-UPPER-BOUND-OF
                 (MAKE-SET X Y)
                 W))
           (LESS-OR-EQUAL-TO Z2
                 W))))))
```

```
(IN-CONTEXT
    ((LET-BE W POSET)
     (LET-BE X (IN-U-SET W))
     (LET-BE Y (IN-U-SET W))
     (LET-BE S (MAKE-SET X Y)))

  (NOTE (AT-MOST-ONE
          (LEAST-UPPER-BOUND-OF S W)))

  (IN-CONTEXT
      ((SUPPOSE
         (EXISTS-SOME
           (LEAST-UPPER-BOUND-OF S W)))
       (LET-BE Z
         (THE (LEAST-UPPER-BOUND-OF S W))))
    (IN-CONTEXT
        ((LET-BE Z2 (UPPER-BOUND-OF S W)))
      (NOTE (IS Z (LESS-OR-EQUAL-TO Z2 W))))))
```

```
(DEFTYPE (GREATEST-LOWER-BOUND-OF
           (S (SUBSET-OF (U-SET W)))
           (W POSET))
  (GREATEST-MEMBER-OF
    (THE-SET-OF-ALL
      (LOWER-BOUND-OF S W))
    W))
```

```
(LEMMA
  (FORALL ((W POSET)
           (S (SUBSET-OF (U-SET W))))
    (IS (THE-SET-OF-ALL
          (LOWER-BOUND-OF S W))
        (SUBSET-OF (U-SET W)))))

(LEMMA
  (FORALL ((W POSET)
           (S (SUBSET-OF (U-SET W))))
    (AT-MOST-ONE
      (GREATEST-LOWER-BOUND-OF S W))))

(LEMMA
 (FORALL
    ((W POSET)
     (S (SUBSET-OF (U-SET W)))
     (X (GREATEST-LOWER-BOUND-OF S W)))
  (IS X (LOWER-BOUND-OF S W))))

(LEMMA
  (FORALL ((W POSET)
           (S (SUBSET-OF (U-SET W))))
    (=> (EXISTS-SOME
          (LOWER-BOUND-OF S W))
        (FORALL
          ((X (LOWER-BOUND-OF S W)))
          (=>
            (IS-EVERY
              (LOWER-BOUND-OF S W)
              (LESS-OR-EQUAL-TO X W))

            (IS X
              (GREATEST-LOWER-BOUND-OF S
                 W)))))))

(LEMMA
  (FORALL ((W POSET)
           (S (SUBSET-OF (U-SET W))))
    (=> (EXISTS-SOME
          (GREATEST-LOWER-BOUND-OF S W))
      (FORALL
          ((Y (LOWER-BOUND-OF S W)))
       (IS-EVERY
         (GREATEST-LOWER-BOUND-OF S W)
         (GREATER-OR-EQUAL-TO Y W))))))
```

```
(IN-CONTEXT
    ((LET-BE W POSET)
     (LET-BE S (SUBSET-OF (U-SET W)))
     (LET-BE S2 (THE-SET-OF-ALL
                  (LOWER-BOUND-OF S W))))
  (IN-CONTEXT ((LET-BE S3 (U-SET W))
               (PUSH-GOAL (IS S2 (SUBSET-OF S3))))
    (IN-CONTEXT ((SUPPOSE
                  (EXISTS-SOME (MEMBER-OF S2)))
                 (LET-BE X (MEMBER-OF S2)))
      (NOTE-GOAL))
    (NOTE-GOAL))

  (NOTE
    (AT-MOST-ONE
      (GREATEST-LOWER-BOUND-OF S W)))

  (IN-CONTEXT
      ((PUSH-GOAL
         (IS-EVERY (GREATEST-LOWER-BOUND-OF S W)
                   (LOWER-BOUND-OF S W))))
    (IN-CONTEXT
        ((SUPPOSE
           (EXISTS-SOME
             (GREATEST-LOWER-BOUND-OF S W)))
         (LET-BE X (GREATEST-LOWER-BOUND-OF S W)))
      (NOTE-GOAL))
    (NOTE-GOAL))

  (IN-CONTEXT
      ((SUPPOSE
         (EXISTS-SOME (LOWER-BOUND-OF S W)))
       (LET-BE X (LOWER-BOUND-OF S W))
       (SUPPOSE
         (IS-EVERY (LOWER-BOUND-OF S W)
                   (LESS-OR-EQUAL-TO X W))))
    (NOTE (IS X (GREATEST-LOWER-BOUND-OF S W))))

  (IN-CONTEXT
      ((SUPPOSE
         (EXISTS-SOME
           (GREATEST-LOWER-BOUND-OF S W)))
       (LET-BE X (GREATEST-LOWER-BOUND-OF S W))
       (LET-BE Y (LOWER-BOUND-OF S W)))
    (NOTE (IS X (GREATER-OR-EQUAL-TO Y W)))))
```

```
(LEMMA
  (FORALL ((W POSET)
           (X (IN-U-SET W))
           (Y (IN-U-SET W)))
    (AT-MOST-ONE
      (GREATEST-LOWER-BOUND-OF
        (MAKE-SET X Y)
        W))))
```

```
(IN-CONTEXT
    ((LET-BE W POSET)
     (LET-BE X (IN-U-SET W))
     (LET-BE Y (IN-U-SET W))
     (LET-BE S (MAKE-SET X Y)))

  (NOTE
    (AT-MOST-ONE
      (GREATEST-LOWER-BOUND-OF S W)))
```

```
(LEMMA
  (FORALL ((W POSET)
           (X (IN-U-SET W))
           (Y (IN-U-SET W)))
    (=>
      (EXISTS-SOME
        (GREATEST-LOWER-BOUND-OF
          (MAKE-SET X Y)
          W))

      (FORALL
          ((Z2 (LOWER-BOUND-OF
                 (MAKE-SET X Y)
                 W)))
        (IS (THE
              (GREATEST-LOWER-BOUND-OF
                   (MAKE-SET X Y)
                   W))
          (GREATER-OR-EQUAL-TO Z2
              W))))))
```

```
  (IN-CONTEXT
      ((SUPPOSE
         (EXISTS-SOME
           (GREATEST-LOWER-BOUND-OF S W)))
       (LET-BE Z
         (THE (GREATEST-LOWER-BOUND-OF S W))))
    (IN-CONTEXT
        ((LET-BE Z2 (LOWER-BOUND-OF S W)))
      (NOTE (IS Z (GREATER-OR-EQUAL-TO Z2 W))))))
```

```
(DEFTYPE (CHAIN-IN (P POSET))
  (LAMBDA ((S (NON-EMPTY-SUBSET-OF (U-SET P))))
    (IS (RESTRICT-ORDER P S)
        TOTALLY-ORDERED-SET)))
```

```
(LEMMA
  (FORALL ((P POSET)
           (X (IN-U-SET P)))
    (IS (MAKE-SET X)
        (CHAIN-IN P))))
```

```
(IN-CONTEXT ((LET-BE P POSET)
             (LET-BE X (IN-U-SET P))
             (LET-BE S (MAKE-SET X))
             (PUSH-GOAL (IS S (CHAIN-IN P))))
  (LET-BE ((RCHAIN (RESTRICT-ORDER P S))
           (LET-BE REL (GET-RELATION RCHAIN)))
          (NOTE-GOAL)))
```

```
(LEMMA
  (FORALL ((P1 POSET)
           (C (CHAIN-IN P1))
           (X (MEMBER-OF C))
           (Y (MEMBER-OF C)))
    (OR (IS X
          (LESS-OR-EQUAL-TO Y P1))
        (IS Y
          (LESS-OR-EQUAL-TO X P1)))))
```

```
(IN-CONTEXT ((LET-BE P1 POSET)
             (LET-BE C (CHAIN-IN P1))
             (LET-BE P2 (RESTRICT-ORDER P1 C))
             (LET-BE X (MEMBER-OF C))
             (LET-BE Y (MEMBER-OF C)))
  (NOTE (OR (IS X (LESS-OR-EQUAL-TO Y P1))
            (IS Y (LESS-OR-EQUAL-TO X P1)))))
```

```
(DEFTYPE INDUCTIVE-ORDER
  (LAMBDA ((R POSET))
    (FORALL ((S (CHAIN-IN R)))
      (EXISTS-SOME (UPPER-BOUND-OF S R)))))

;We take Zorn's Lemma as an axiom
(AXIOM
  (FORALL ((R INDUCTIVE-ORDER)
           (X (IN-U-SET R)))
    (EXISTS-SOME
      (AND-TYPE (MAXIMAL-ELEMENT-OF R)
                (GREATER-OR-EQUAL-TO X R)))))
```

A.6 Lattices

A lattice is a poset in which every pair of elements has both a least upper bound and a greatest lower bound. The greatest lower bound and least upper bound of two elements are called the meet and join respectively. A complete lattice is a poset in which every subset of the underlying set has a least upper bound. We prove that in a complete lattice every subset also has a greatest lower bound.

The inclusion order on a family of sets F is a poset whose underlying set is the family F and where x is less than or equal to y just in case x is a subset of y. For any set s the inclusion order on the power set of s is a complete lattice such that for any subset F of the power set of s the least upper bound and greatest lower bound of F are resectively the union and intersection over F. The poset which is the inclusion order on the power set of s is called a *power set lattice.*

The meet and join functions are monotone in each argument, i.e. increasing an argument never decreases the meet or join. The meet of x and the meet of y and z is the greatest lower bound of the set x, y, z and thus the meet function is associative. The join function is similarly associative.

```
(DEFTYPE LATTICE
  (LAMBDA ((W POSET))
    (FORALL ((X (IN-U-SET W))
             (Y (IN-U-SET W)))
      (AND
        (EXISTS-SOME
          (LEAST-UPPER-BOUND-OF (MAKE-SET X Y) W))
        (EXISTS-SOME
          (GREATEST-LOWER-BOUND-OF (MAKE-SET X Y) W))))))

(DEFTERM (JOIN (X (IN-U-SET L))
               (Y (IN-U-SET L))
               (L LATTICE))
  (THE (LEAST-UPPER-BOUND-OF (MAKE-SET X Y) L)))

(DEFTERM (MEET (X (IN-U-SET L))
               (Y (IN-U-SET L))
               (L LATTICE))
  (THE (GREATEST-LOWER-BOUND-OF (MAKE-SET X Y) L)))

(DEFTYPE COMPLETE-LATTICE
  (LAMBDA ((W POSET))
    (FORALL ((S (SUBSET-OF (U-SET W))))
      (EXISTS-SOME (LEAST-UPPER-BOUND-OF S W)))))
```

```
(LEMMA (EXISTS-SOME COMPLETE-LATTICE))   (IN-CONTEXT
                                             ((PUSH-GOAL (EXISTS-SOME COMPLETE-LATTICE))
                                              (LET-BE S SINGLETON-SET)
                                              (LET-BE R (THE-EMPTY-RELATION-ON S))
                                              (LET-BE W (MAKE-RELATION-STRUCTURE R S))
                                              (LET-BE S2 (SUBSET-OF (U-SET W))))
                                           (IN-CONTEXT
                                               ((PUSH-GOAL
                                                  (EXISTS-SOME
                                                    (LEAST-UPPER-BOUND-OF S2 W))))
                                             (IN-CONTEXT
                                                 ((SUPPOSE (EXISTS-SOME (MEMBER-OF S2)))
                                                  (LET-BE X (MEMBER-OF S2))
                                                  (LET-BE Y (UPPER-BOUND-OF S2 W)))
                                                (NOTE-GOAL))
                                             (IN-CONTEXT
                                                 ((SUPPOSE
                                                    (NOT (EXISTS-SOME (MEMBER-OF S2))))
                                                  (LET-BE X (MEMBER-OF S))
                                                  (LET-BE Y (UPPER-BOUND-OF S2 W)))
                                               (NOTE-GOAL))
                                             (NOTE-GOAL))
                                           (NOTE-GOAL))
```

```
(LEMMA
  (FORALL ((W COMPLETE-LATTICE)
           (S (SUBSET-OF (U-SET W))))
    (EXISTS-SOME
      (GREATEST-LOWER-BOUND-OF S W))))
```

```
(IN-CONTEXT
    ((LET-BE W COMPLETE-LATTICE)
     (LET-BE S (SUBSET-OF (U-SET W)))
     (PUSH-GOAL
       (EXISTS-SOME
        (GREATEST-LOWER-BOUND-OF S W))))

  (IN-CONTEXT
      ((LET-BE S2
         (THE-SET-OF-ALL (LOWER-BOUND-OF S W)))
       (LET-BE X
         (THE (LEAST-UPPER-BOUND-OF S2 W))))

    (IN-CONTEXT
        ((PUSH-GOAL (IS X (LOWER-BOUND-OF S W))))
      (IN-CONTEXT ((SUPPOSE
                     (EXISTS-SOME (MEMBER-OF S)))
                   (LET-BE Y (MEMBER-OF S)))
        (IN-CONTEXT
            ((PUSH-GOAL
               (IS Y (UPPER-BOUND-OF S2 W))))
          (IN-CONTEXT
              ((SUPPOSE
                 (EXISTS-SOME (MEMBER-OF S2)))
               (LET-BE Z (MEMBER-OF S2)))
            (NOTE-GOAL))
          (NOTE-GOAL)))
      (NOTE-GOAL))

    (NOTE-GOAL)))
```

```
(LEMMA
  (FORALL ((W COMPLETE-LATTICE))
    (IS W LATTICE)))
```

```
(IN-CONTEXT ((LET-BE W COMPLETE-LATTICE)
             (PUSH-GOAL (IS W LATTICE)))
  (IN-CONTEXT ((LET-BE X (IN-U-SET W))
               (LET-BE Y (IN-U-SET W))
               (LET-BE SXY (MAKE-SET X Y)))
    (NOTE-GOAL)))
```

```
(DEFTERM (INCLUSION-ORDER (F FAMILY-OF-SETS))
  (MAKE-RELATION-STRUCTURE
     (THE-RULE ((S (MEMBER-OF F)))
       (THE-SET-OF-ALL
         (AND-TYPE (MEMBER-OF F)
                   (PROPER-SUPERSET-OF S))))
     F))
```

```
(LEMMA
  (FORALL ((F FAMILY-OF-SETS))
    (IS (THE-RULE ((S (MEMBER-OF F)))
          (THE-SET-OF-ALL
           (AND-TYPE
            (MEMBER-OF F)
            (PROPER-SUPERSET-OF S))))
      (RELATION-ON F))))

(LEMMA
  (FORALL ((F FAMILY-OF-SETS))
    (IS (INCLUSION-ORDER F) POSET)))
```

```
(IN-CONTEXT
    ((LET-BE F FAMILY-OF-SETS)
     (LET-BE R
        (THE-RULE ((S (MEMBER-OF F)))
          (THE-SET-OF-ALL
            (AND-TYPE (MEMBER-OF F)
                      (PROPER-SUPERSET-OF S))))))

  (IN-CONTEXT
      ((PUSH-GOAL (IS R (RELATION-ON F)))
       (LET-BE S (MEMBER-OF F))
       (LET-BE F2 (APPLY-RULE R S)))
    (IN-CONTEXT
        ((PUSH-GOAL (IS F2 (SUBSET-OF F))))
      (IN-CONTEXT
          ((SUPPOSE
              (EXISTS-SOME (MEMBER-OF F2)))
           (LET-BE S2 (MEMBER-OF F2)))
        (NOTE-GOAL))
      (NOTE-GOAL))
    (NOTE-GOAL))

  (IN-CONTEXT
      ((PUSH-GOAL
         (IS (INCLUSION-ORDER F) POSET)))
    (IN-CONTEXT
        ((PUSH-GOAL
           (IS R (PARTIAL-ORDER-ON F)))
         (LET-BE S1 (MEMBER-OF F)))
      (IN-CONTEXT
          ((PUSH-GOAL
              (FORALL ((S2 (RELATED-TO S1 R)))
                (IS-EVERY (RELATED-TO S2 R)
                          (RELATED-TO S1 R)))))
        (IN-CONTEXT
            ((SUPPOSE
                (EXISTS-SOME (RELATED-TO S1 R)))
             (LET-BE S2 (RELATED-TO S1 R)))
          (IN-CONTEXT
              ((PUSH-GOAL
                  (IS-EVERY (RELATED-TO S2 R)
                            (RELATED-TO S1 R))))
            (IN-CONTEXT
                ((SUPPOSE
                    (EXISTS-SOME
                      (RELATED-TO S2 R)))
                 (LET-BE S3 (RELATED-TO S2 R)))
              (NOTE (IS S3 (NOT-EQUAL-TO S1)))
              (NOTE-GOAL))
            (NOTE-GOAL))
          (NOTE-GOAL))
        (NOTE-GOAL))
      (NOTE-GOAL))
    (IN-CONTEXT
        ((LET-BE W (INCLUSION-ORDER F)))
      (NOTE-GOAL))))
```

```
(LEMMA (FORALL ((F FAMILY-OF-SETS))
         (= (U-SET
              (INCLUSION-ORDER F))
            F)))

(LEMMA
  (FORALL ((F FAMILY-OF-SETS)
           (S2 (MEMBER-OF F))
           (S1 (MEMBER-OF F)))
    (IFF
      (IS S1
          (LESS-OR-EQUAL-TO
            S2
            (INCLUSION-ORDER F)))
      (IS S1
        (SUBSET-OF S2)))))
```

```
(IN-CONTEXT
    ((LET-BE F FAMILY-OF-SETS)
     (LET-BE R
       (THE-RULE ((S (MEMBER-OF F)))
         (THE-SET-OF-ALL
           (AND-TYPE (MEMBER-OF F)
                     (PROPER-SUPERSET-OF S)))))
     (LET-BE W (INCLUSION-ORDER F)))

  (NOTE (= (U-SET W) F))

  (IN-CONTEXT
      ((LET-BE S1 (MEMBER-OF F))
       (LET-BE S2 (MEMBER-OF F))
       (PUSH-GOAL
         (IFF (IS S1 (LESS-OR-EQUAL-TO S2 W))
              (IS S1 (SUBSET-OF S2)))))

    (IN-CONTEXT
        ((SUPPOSE
           (IS S1 (LESS-OR-EQUAL-TO S2 W))))
      (IN-CONTEXT ((SUPPOSE (= S1 S2)))
        (NOTE-GOAL))
      (NOTE-GOAL))

    (IN-CONTEXT ((SUPPOSE (IS S1 (SUBSET-OF S2))))
      (IN-CONTEXT ((SUPPOSE (= S1 S2)))
        (NOTE-GOAL))
      (NOTE-GOAL))

    (NOTE-GOAL)))
```

```
(DEFTERM (POWER-SET-LATTICE (S NON-EMPTY-SET))
  (INCLUSION-ORDER (POWER-SET S)))

(DEFTYPE POWER-LATTICE
  (WRITABLE-AS (POWER-SET-LATTICE S)
    (S NON-EMPTY-SET)))
```

```
(LEMMA
  (FORALL ((B POWER-LATTICE))
    (IS B POSET)))

(LEMMA
  (FORALL ((B POWER-LATTICE))
    (IS (U-SET B)
        FAMILY-OF-SETS)))

(LEMMA
  (FORALL
      ((B POWER-LATTICE)
       (S NON-EMPTY-SET
          (= B (POWER-SET-LATTICE S))))
    (= S
       (FAMILY-UNION
         (U-SET B)))))

(LEMMA
  (FORALL ((B POWER-LATTICE))
    (IS (FAMILY-UNION (U-SET B))
        NON-EMPTY-SET)))

(LEMMA
  (FORALL ((B POWER-LATTICE))
    (= (U-SET B)
       (POWER-SET
         (FAMILY-UNION
           (U-SET B))))))
```

```
(IN-CONTEXT
    ((LET-BE B POWER-LATTICE)
     (WRITE-AS B (POWER-SET-LATTICE S)
       (S NON-EMPTY-SET))
     (LET-BE P (U-SET B))
     (LET-BE P2 (POWER-SET S)))
  (NOTE (IS B POSET))
  (NOTE (IS (U-SET B) FAMILY-OF-SETS))
  (NOTE (= S (FAMILY-UNION (U-SET B))))
  (NOTE (IS (FAMILY-UNION (U-SET B))
            NON-EMPTY-SET))
  (NOTE
    (= (U-SET B)
       (POWER-SET
         (FAMILY-UNION (U-SET B))))))
```

```
(LEMMA
  (FORALL ((B POWER-LATTICE)
           (S2 (IN-U-SET B)))
    (IS S2 SET)))

(LEMMA
  (FORALL ((B POWER-LATTICE)
           (S2 (IN-U-SET B)))
    (IS S2
        (SUBSET-OF
          (FAMILY-UNION
            (U-SET B))))))
```

```
(IN-CONTEXT
    ((LET-BE B POWER-LATTICE)
     (WRITE-AS B (POWER-SET-LATTICE S)
       (S NON-EMPTY-SET))
     (LET-BE P (U-SET B))
     (LET-BE S2 (IN-U-SET B)))
  (NOTE (IS S2 SET))
  (NOTE
    (IS S2
        (SUBSET-OF
          (FAMILY-UNION (U-SET B))))))
```

```
(LEMMA
  (FORALL ((B POWER-LATTICE)
           (S2 (SUBSET-OF
                  (FAMILY-UNION
                    (U-SET B)))))
    (IS S2 (IN-U-SET B))))
```

```
(IN-CONTEXT
    ((LET-BE B POWER-LATTICE)
     (WRITE-AS B (POWER-SET-LATTICE S)
       (S NON-EMPTY-SET))
     (LET-BE P (U-SET B)))

  (IN-CONTEXT
      ((LET-BE S2
               (SUBSET-OF
                 (FAMILY-UNION (U-SET B)))))
    (NOTE (IS S2 (IN-U-SET B))))
```

```
(LEMMA
  (FORALL
      ((B POWER-LATTICE)
       (S2 (IN-U-SET B))
       (S3 (LESS-OR-EQUAL-TO S2 B)))
    (IS S3 (SUBSET-OF S2))))
```

```
(LEMMA
  (FORALL ((B POWER-LATTICE)
           (S2 (IN-U-SET B))
           (S3 (SUBSET-OF S2)))
    (IS S3
        (LESS-OR-EQUAL-TO S2 B))))
```

```
  (IN-CONTEXT ((LET-BE S2 (IN-U-SET B)))
    (IN-CONTEXT
        ((LET-BE S3 (LESS-OR-EQUAL-TO S2 B)))
      (NOTE (IS S3 (SUBSET-OF S2))))
    (IN-CONTEXT ((LET-BE S3 (SUBSET-OF S2)))
      (NOTE (IS S3 (LESS-OR-EQUAL-TO S2 B)))))
```

```
(LEMMA
  (FORALL ((B POWER-LATTICE)
           (F (NON-EMPTY-SUBSET-OF
                (U-SET B))))
    (IS F FAMILY-OF-SETS)))
```

```
  (IN-CONTEXT
      ((LET-BE F (NON-EMPTY-SUBSET-OF (U-SET B)))
       (PUSH-GOAL (IS F FAMILY-OF-SETS)))
    (IN-CONTEXT ((LET-BE S (MEMBER-OF F)))
      (NOTE-GOAL))))
```

```
(LEMMA
  (FORALL ((B POWER-LATTICE)
           (F (NON-EMPTY-SUBSET-OF
                (U-SET B))))
    (IS (FAMILY-UNION F)
        (LEAST-UPPER-BOUND-OF F B))))
```

```
(IN-CONTEXT
    ((LET-BE B POWER-LATTICE)
     (LET-BE F (NON-EMPTY-SUBSET-OF (U-SET B)))
     (LET-BE LUB (FAMILY-UNION F))
     (PUSH-GOAL
       (IS LUB (LEAST-UPPER-BOUND-OF F B))))
  (IN-CONTEXT
      ((PUSH-GOAL (IS LUB (IN-U-SET B)))
       (LET-BE S (FAMILY-UNION (U-SET B))))
    (NOTE-GOAL))
  (IN-CONTEXT
      ((PUSH-GOAL
         (IS LUB (UPPER-BOUND-OF F B)))
       (LET-BE S (MEMBER-OF F)))
    (NOTE-GOAL))
  (IN-CONTEXT
      ((LET-BE S (UPPER-BOUND-OF F B)))
    (IN-CONTEXT
        ((PUSH-GOAL
           (IS-EVERY (MEMBER-OF F)
                     (SUBSET-OF S)))
         (LET-BE S2 (MEMBER-OF F)))
      (NOTE-GOAL))
    (NOTE-GOAL))
```

```
(LEMMA
  (FORALL ((B POWER-LATTICE)
           (F (NON-EMPTY-SUBSET-OF
                (U-SET B))))
    (IS (FAMILY-INTERSECTION F)
        (GREATEST-LOWER-BOUND-OF F B))))
```

```
(IN-CONTEXT
    ((LET-BE B POWER-LATTICE)
     (LET-BE F (NON-EMPTY-SUBSET-OF (U-SET B)))
     (LET-BE GLB (FAMILY-INTERSECTION F))
     (PUSH-GOAL
       (IS GLB
           (GREATEST-LOWER-BOUND-OF F B))))
  (IN-CONTEXT
      ((PUSH-GOAL (IS GLB (IN-U-SET B)))
       (LET-BE S (FAMILY-UNION (U-SET B)))
       (LET-BE S2 (MEMBER-OF F)))
    (NOTE-GOAL))
  (IN-CONTEXT
      ((PUSH-GOAL (IS GLB (LOWER-BOUND-OF F B)))
       (LET-BE S (MEMBER-OF F)))
              (NOTE-GOAL))
  (IN-CONTEXT
      ((LET-BE S (LOWER-BOUND-OF F B)))
    (IN-CONTEXT
        ((PUSH-GOAL
           (IS-EVERY (MEMBER-OF F)
                     (SUPERSET-OF S)))
         (LET-BE S2 (MEMBER-OF F)))
      (NOTE-GOAL))
    (NOTE-GOAL)))
```

```
(LEMMA
  (FORALL ((B POWER-LATTICE))
    (IS B COMPLETE-LATTICE)))
```

```
(IN-CONTEXT
    ((LET-BE B POWER-LATTICE)
     (PUSH-GOAL (IS B COMPLETE-LATTICE)))
  (IN-CONTEXT
      ((LET-BE F (SUBSET-OF (U-SET B))))
    (IN-CONTEXT
        ((PUSH-GOAL
            (EXISTS-SOME
              (LEAST-UPPER-BOUND-OF F B))))
      (IN-CONTEXT
          ((SUPPOSE
              (EXISTS-SOME (MEMBER-OF F))))
        (IN-CONTEXT
            ((LET-BE S (U-SET B)))
          (NOTE+GENERALIZE
            (IS F
                (NON-EMPTY-SUBSET-OF
                  (U-SET B)))))
        (NOTE-GOAL))
      (IN-CONTEXT
          ((SUPPOSE
              (NOT (EXISTS-SOME (MEMBER-OF F))))
            (LET-BE ESET THE-EMPTY-SET))
        (IN-CONTEXT
            ((PUSH-GOAL (IS ESET (IN-U-SET B)))
              (LET-BE S (FAMILY-UNION (U-SET B))))
          (NOTE-GOAL))
        (IN-CONTEXT
            ((LET-BE S (UPPER-BOUND-OF F B)))
          (NOTE-GOAL)))
      (NOTE-GOAL))
  (NOTE-GOAL)))
```

```
(LEMMA
  (FORALL ((B POWER-LATTICE)
           (S1 (IN-U-SET B))
           (S2 (IN-U-SET B)))
    (= (JOIN S1 S2 B)
       (UNION S1 S2))))

(LEMMA
  (FORALL ((B POWER-LATTICE)
           (S1 (IN-U-SET B))
           (S2 (IN-U-SET B)))
    (= (MEET S1 S2 B)
       (INTERSECTION S1 S2))))
```

```
(IN-CONTEXT ((LET-BE B POWER-LATTICE)
             (LET-BE S1 (IN-U-SET B))
             (LET-BE S2 (IN-U-SET B)))

  (IN-CONTEXT
      ((PUSH-GOAL
         (= (JOIN S1 S2 B)
            (UNION S1 S2))))
    (IN-CONTEXT
        ((LET-BE S3 (MAKE-SET S1 S2)))
      (NOTE
        (EXACTLY-ONE
          (LEAST-UPPER-BOUND-OF S3 B)))
      (NOTE
        (IS (UNION S1 S2)
            (LEAST-UPPER-BOUND-OF S3 B))))
    (IN-CONTEXT
        ((LET-BE J (JOIN S1 S2 B))
         (LET-BE U (UNION S1 S2)))
      (NOTE-GOAL)))

  (IN-CONTEXT
      ((PUSH-GOAL
         (= (MEET S1 S2 B)
            (INTERSECTION S1 S2))))
    (IN-CONTEXT
        ((LET-BE S3 (MAKE-SET S1 S2)))
      (NOTE
        (EXACTLY-ONE
          (GREATEST-LOWER-BOUND-OF S3 B)))
      (NOTE
        (IS (INTERSECTION S1 S2)
            (GREATEST-LOWER-BOUND-OF S3 B))))
    (IN-CONTEXT
        ((LET-BE J (MEET S1 S2 B))
         (LET-BE U (INTERSECTION S1 S2)))
      (NOTE-GOAL))))
```

```
(LEMMA
  (FORALL ((L LATTICE)
           (X (IN-U-SET L))
           (Y (IN-U-SET L)))
    (IS (MEET X Y L)
        (LESS-OR-EQUAL-TO X L))))

(LEMMA
  (FORALL ((L LATTICE)
           (X (IN-U-SET L))
           (Y (IN-U-SET L)))
    (IS (JOIN X Y L)
        (GREATER-OR-EQUAL-TO X L))))

(LEMMA
  (FORALL ((L LATTICE)
           (X (IN-U-SET L))
           (Y (IN-U-SET L)))
    (IS-EVERY
      (AND-TYPE
        (LESS-OR-EQUAL-TO X L)
        (LESS-OR-EQUAL-TO Y L))
      (LESS-OR-EQUAL-TO
        (MEET X Y L)
        L))))

(LEMMA
  (FORALL ((L LATTICE)
           (X (IN-U-SET L))
           (Y (IN-U-SET L)))
    (IS-EVERY
      (AND-TYPE
        (GREATER-OR-EQUAL-TO X L)
        (GREATER-OR-EQUAL-TO Y L))
      (GREATER-OR-EQUAL-TO
        (JOIN X Y L)
        L))))
```

```
(IN-CONTEXT
    ((LET-BE L LATTICE)
     (LET-BE X (IN-U-SET L))
     (LET-BE Y (IN-U-SET L))
     (LET-BE S (MAKE-SET X Y)))

  (IN-CONTEXT
      ((PUSH-GOAL
          (IS (MEET X Y L)
              (LESS-OR-EQUAL-TO X L)))
        (LET-BE M (MEET X Y L)))
    (NOTE-GOAL))

  (IN-CONTEXT
      ((PUSH-GOAL
          (IS (JOIN X Y L)
              (GREATER-OR-EQUAL-TO X L)))
        (LET-BE J (JOIN X Y L)))
    (NOTE-GOAL))

  (IN-CONTEXT
      ((PUSH-GOAL
          (IS-EVERY
              (AND-TYPE
                (LESS-OR-EQUAL-TO X L)
                (LESS-OR-EQUAL-TO Y L))
            (LESS-OR-EQUAL-TO (MEET X Y L) L))))
    (IN-CONTEXT
        ((SUPPOSE
            (EXISTS-SOME
              (AND-TYPE
                  (LESS-OR-EQUAL-TO X L)
                  (LESS-OR-EQUAL-TO Y L))))
          (LET-BE Z
           (AND-TYPE (LESS-OR-EQUAL-TO X L)
                     (LESS-OR-EQUAL-TO Y L)))
          (LET-BE M (MEET X Y L)))
      (NOTE-GOAL))
    (NOTE-GOAL))

  (IN-CONTEXT
      ((PUSH-GOAL
          (IS-EVERY
            (AND-TYPE
              (GREATER-OR-EQUAL-TO X L)
              (GREATER-OR-EQUAL-TO Y L))
            (GREATER-OR-EQUAL-TO (JOIN X Y L) L))))
    (IN-CONTEXT
        ((SUPPOSE
            (EXISTS-SOME
              (AND-TYPE (GREATER-OR-EQUAL-TO X L)
                        (GREATER-OR-EQUAL-TO Y L))))
          (LET-BE Z (AND-TYPE
                      (GREATER-OR-EQUAL-TO X L)
                      (GREATER-OR-EQUAL-TO Y L)))
          (LET-BE J (JOIN X Y L)))
      (NOTE-GOAL))
    (NOTE-GOAL)))
```

```
(LEMMA
  (FORALL ((L LATTICE)
           (Y (IN-U-SET L))
           (X (IN-U-SET L)))
    (IFF (IS X
            (LESS-OR-EQUAL-TO Y L))
         (= (MEET X Y L)
            X))))

(LEMMA
  (FORALL ((L LATTICE)
           (Y (IN-U-SET L))
           (X (IN-U-SET L)))
    (IFF (IS X
            (GREATER-OR-EQUAL-TO Y L))
         (= (JOIN X Y L)
            X))))

(LEMMA
  (FORALL ((L LATTICE)
           (X (IN-U-SET L))
           (Y (IN-U-SET L)))
    (= (JOIN (MEET X Y L)
             Y
             L)
       Y)))

(LEMMA
  (FORALL ((L LATTICE)
           (X (IN-U-SET L))
           (Y (IN-U-SET L)))
    (= (MEET (JOIN X Y L)
             Y
             L)
       Y)))
```

```
(IN-CONTEXT ((LET-BE L LATTICE)
             (LET-BE X (IN-U-SET L))
             (LET-BE Y (IN-U-SET L)))
  (IN-CONTEXT
      ((PUSH-GOAL
         (IFF (IS X (LESS-OR-EQUAL-TO Y L))
              (= (MEET X Y L) X))))

    ;the ony-if case is trivial
    (IN-CONTEXT ((SUPPOSE (= (MEET X Y L) X)))
      (NOTE-GOAL))

    (IN-CONTEXT
        ((SUPPOSE
           (IS X (LESS-OR-EQUAL-TO Y L))))
      ;in this case it is obvious that x
      ;is a lower bound, thus we only need
      ;to show that x is the greatest lower
      ;bound
      (IN-CONTEXT
          ((LET-BE Z
             (UPPER-BOUND-OF (MAKE-SET X Y) L))
           (LET-BE S (MAKE-SET X Y)))
        (NOTE-GOAL)))

    (NOTE-GOAL))

  (IN-CONTEXT
      ((PUSH-GOAL
         (IFF (IS X (GREATER-OR-EQUAL-TO Y L))
              (= (JOIN X Y L) X))))

    (IN-CONTEXT ((SUPPOSE (= (JOIN X Y L) X)))
      (NOTE-GOAL))

    (IN-CONTEXT
        ((SUPPOSE
           (IS X (GREATER-OR-EQUAL-TO Y L))))
      (IN-CONTEXT
          ((LET-BE Z
             (UPPER-BOUND-OF (MAKE-SET X Y) L))
           (LET-BE S (MAKE-SET X Y)))
        (NOTE-GOAL)))
    (NOTE-GOAL))

  (IN-CONTEXT
      ((PUSH-GOAL (= (JOIN (MEET X Y L) Y L)
                     Y))
       (LET-BE M (MEET X Y L)))
    (NOTE-GOAL))

  (IN-CONTEXT
      ((PUSH-GOAL (= (MEET (JOIN X Y L) Y L)
                     Y))
       (LET-BE J (JOIN X Y L)))
    (NOTE-GOAL)))
```

```
(LEMMA
  (FORALL ((L LATTICE)
           (X (IN-U-SET L))
           (X2 (LESS-OR-EQUAL-TO X L))
           (Y (IN-U-SET L)))
    (IS (MEET X2 Y L)
        (LESS-OR-EQUAL-TO (MEET X Y L)
                          L))))

(LEMMA
  (FORALL ((L LATTICE)
           (X (IN-U-SET L))
           (X2 (LESS-OR-EQUAL-TO X L))
           (Y (IN-U-SET L)))
    (IS (JOIN X Y L)
        (GREATER-OR-EQUAL-TO
          (JOIN X2 Y L)
          L))))
```

```
(IN-CONTEXT
    ((LET-BE L LATTICE)
     (LET-BE X (IN-U-SET L))
     (LET-BE Y (IN-U-SET L))
     (LET-BE X2 (LESS-OR-EQUAL-TO X L)))
  (IN-CONTEXT
      ((PUSH-GOAL
         (IS (MEET X2 Y L)
             (LESS-OR-EQUAL-TO (MEET X Y L) L))))
    (IN-CONTEXT ((LET-BE M (MEET X2 Y L)))
      (NOTE-GOAL)))
  (IN-CONTEXT
      ((PUSH-GOAL
         (IS (JOIN X Y L)
             (GREATER-OR-EQUAL-TO
               (JOIN X2 Y L)
               L))))
    (IN-CONTEXT ((LET-BE J (JOIN X Y L)))
      (NOTE-GOAL))))
```

```
(LEMMA
  (FORALL ((L LATTICE)
           (X (IN-U-SET L))
           (Y (IN-U-SET L))
           (Z (IN-U-SET L)))
    (= (MEET Z (MEET X Y L) L)
       (THE
         (GREATEST-LOWER-BOUND-OF
           (MAKE-SET X Y Z)
           L)))))

(LEMMA
  (FORALL ((L LATTICE)
           (X (IN-U-SET L))
           (Y (IN-U-SET L))
           (Z (IN-U-SET L)))
    (= (JOIN Z (JOIN X Y L) L)
       (THE
         (LEAST-UPPER-BOUND-OF
           (MAKE-SET X Y Z)
           L)))))
```

```
(IN-CONTEXT
    ((LET-BE L LATTICE)
     (LET-BE X (IN-U-SET L))
     (LET-BE Y (IN-U-SET L))
     (LET-BE Z (IN-U-SET L))
     (LET-BE SXY (MAKE-SET X Y))
     (LET-BE SXYZ (MAKE-SET X Y Z)))
  ;meet is associative
  (IN-CONTEXT
      ((LET-BE MXY (MEET X Y L))
       (LET-BE MXYZ (MEET Z MXY L))
       (PUSH-GOAL
         (= MXYZ
            (THE
              (GREATEST-LOWER-BOUND-OF SXYZ L)))))
    ;it is already a lower bound so we must show
    ;that it is the greatest
    (IN-CONTEXT
        ((LET-BE LBOUND (LOWER-BOUND-OF SXYZ L)))
      (NOTE-GOAL)))
  ;join is associative
  (IN-CONTEXT
      ((LET-BE JXY (JOIN X Y L))
       (LET-BE JXYZ (JOIN Z JXY L))
       (PUSH-GOAL
         (= JXYZ
            (THE
              (LEAST-UPPER-BOUND-OF SXYZ L)))))
    (IN-CONTEXT
        ((LET-BE UBOUND (UPPER-BOUND-OF SXYZ L)))
      (NOTE-GOAL))))
```

```
(LEMMA
  (FORALL ((L LATTICE)
           (Y (IN-U-SET L))
           (X (IN-U-SET L)))
    (= (MEET X Y L)
       (MEET Y X L))))

(LEMMA
  (FORALL ((L LATTICE)
           (Y (IN-U-SET L))
           (X (IN-U-SET L)))
    (= (JOIN X Y L)
       (JOIN Y X L))))

(LEMMA
 (FORALL ((L LATTICE)
          (Z (IN-U-SET L))
          (X (IN-U-SET L))
          (Y (IN-U-SET L)))
   (= (MEET (MEET X Y L) Z L)
      (MEET Z (MEET X Y L) L))))

(LEMMA
  (FORALL ((L LATTICE)
           (Z (IN-U-SET L))
           (X (IN-U-SET L))
           (Y (IN-U-SET L)))
    (= (JOIN (JOIN X Y L) Z L)
       (JOIN Z (JOIN X Y L) L))))

(LEMMA
  (FORALL ((L LATTICE)
           (X (IN-U-SET L))
           (Y (IN-U-SET L))
           (Z (IN-U-SET L)))
    (= (MEET X (MEET Y Z L) L)
       (MEET (MEET X Y L) Z L))))

(LEMMA
  (FORALL ((L LATTICE)
           (X (IN-U-SET L))
           (Y (IN-U-SET L))
           (Z (IN-U-SET L)))
    (= (JOIN X (JOIN Y Z L) L)
       (JOIN (JOIN X Y L) Z L))))
```

```
(IN-CONTEXT ((LET-BE L LATTICE)
             (LET-BE X (IN-U-SET L))
             (LET-BE Y (IN-U-SET L)))
  (NOTE (= (MEET X Y L)
           (MEET Y X L)))
  (NOTE (= (JOIN X Y L)
           (JOIN Y X L)))
  (IN-CONTEXT ((LET-BE Z (IN-U-SET L)))
    (IN-CONTEXT ((LET-BE MXY (MEET X Y L)))
      (NOTE (= (MEET MXY Z L)
               (MEET Z MXY L))))
    (IN-CONTEXT ((LET-BE JXY (JOIN X Y L)))
      (NOTE (= (JOIN JXY Z L)
               (JOIN Z JXY L))))
    (NOTE (= (MEET X (MEET Y Z L) L)
             (MEET (MEET X Y L) Z L)))
    (NOTE (= (JOIN X (JOIN Y Z L) L)
             (JOIN (JOIN X Y L) Z L)))))
```

A.7 Bounded, Distributive and Complemented Lattices

A bounded lattice is a lattice with a greatest and a least member where the greatest member is distinct from the least member (singleton lattices are ruled out). If L is a bounded lattice and x and y are elements of L we say that x and y are complements if their meet is the least member of L and there join is the greatest member of L. A complemented lattice is a bounded lattice in which every element has at least one complement.

A distributive lattice is lattice in which meet distributes over join and vice versa. In a bounded distributive lattice every element has at most one complement. A Boolean lattice is a complemented distributive lattice. We prove deMorgan's laws for Boolean lattices and establish several distinct characterizations of the lattice order relation.

We also show that every power set lattice is a Boolean lattice.

```
(DEFTYPE BOUNDED-LATTICE
  (LAMBDA ((L LATTICE))
    (AND
      (EXISTS-SOME
        (GREATEST-MEMBER-OF (U-SET L) L))
      (EXISTS-SOME
        (LEAST-MEMBER-OF (U-SET L) L))
      (NOT
        (= (THE (GREATEST-MEMBER-OF (U-SET L) L))
           (THE (LEAST-MEMBER-OF (U-SET L) L)))))))
```

```
(LEMMA
 (FORALL ((L LATTICE))
   (AT-MOST-ONE
     (GREATEST-MEMBER-OF (U-SET L) L))))
```

```
(IN-CONTEXT ((LET-BE L LATTICE)
             (LET-BE S (U-SET L)))
  (NOTE (AT-MOST-ONE (GREATEST-MEMBER-OF S L)))
  (NOTE (AT-MOST-ONE (LEAST-MEMBER-OF S L))))
```

```
(LEMMA
  (FORALL ((L LATTICE))
    (AT-MOST-ONE
      (LEAST-MEMBER-OF (U-SET L) L))))
```

```
(DEFTERM (TOP (L BOUNDED-LATTICE))
  (THE (GREATEST-MEMBER-OF (U-SET L) L)))

(DEFTERM (BOTTOM (L BOUNDED-LATTICE))
  (THE (LEAST-MEMBER-OF (U-SET L) L)))
```

```
(LEMMA
  (FORALL ((L POWER-LATTICE))
    (NOT (= (FAMILY-UNION (U-SET L))
            THE-EMPTY-SET))))

(LEMMA
  (FORALL ((L POWER-LATTICE))
    (IS L BOUNDED-LATTICE)))

(LEMMA
  (FORALL ((L POWER-LATTICE))
    (= (TOP L)
       (FAMILY-UNION (U-SET L)))))

(LEMMA
  (FORALL ((L POWER-LATTICE))
    (= (BOTTOM L) THE-EMPTY-SET)))
```

```
(IN-CONTEXT
    ((LET-BE L POWER-LATTICE)
     (LET-BE F (U-SET L))
     (LET-BE T (FAMILY-UNION F))
     (LET-BE BOT THE-EMPTY-SET)
     (LET-BE X (IN-U-SET L)))
  (NOTE (NOT (= T BOT)))
  (NOTE (IS L BOUNDED-LATTICE))
  (NOTE (= (TOP L) T))
  (NOTE (= (BOTTOM L) BOT)))
```

```
(LEMMA
  (FORALL ((L BOUNDED-LATTICE))
    (IS (TOP L)
        (IN-U-SET L))))

(LEMMA
  (FORALL ((L BOUNDED-LATTICE)
           (X (IN-U-SET L)))
    (IS X
        (LESS-OR-EQUAL-TO (TOP L)
           L))))

(LEMMA
  (FORALL ((L BOUNDED-LATTICE)
           (X (IN-U-SET L)))
    (= X
       (MEET X (TOP L) L))))

(LEMMA
  (FORALL ((L BOUNDED-LATTICE)
           (X (IN-U-SET L)))
    (= (TOP L)
       (JOIN X (TOP L) L))))

(LEMMA
  (FORALL ((L BOUNDED-LATTICE))
    (IS (BOTTOM L)
        (IN-U-SET L))))

(LEMMA
  (FORALL ((L BOUNDED-LATTICE)
           (X (IN-U-SET L)))
    (IS X
        (GREATER-OR-EQUAL-TO
          (BOTTOM L)
          L))))

(LEMMA
  (FORALL ((L BOUNDED-LATTICE)
           (X (IN-U-SET L)))
    (= X
       (JOIN X (BOTTOM L) L))))
(LEMMA
  (FORALL ((L BOUNDED-LATTICE)
           (X (IN-U-SET L)))
    (= (BOTTOM L)
       (MEET X (BOTTOM L) L))))
```

```
(IN-CONTEXT ((LET-BE L BOUNDED-LATTICE)
             (LET-BE X (IN-U-SET L))
             (LET-BE S (U-SET L)))

  (IN-CONTEXT ((LET-BE T (TOP L)))
    (NOTE (IS T (IN-U-SET L)))
    (NOTE (IS X (LESS-OR-EQUAL-TO T L)))
    (NOTE (= X (MEET X T L)))
    (NOTE (= T (JOIN X T L))))

  (IN-CONTEXT ((LET-BE F (BOTTOM L)))
    (NOTE (IS F (IN-U-SET L)))
    (NOTE (IS X (GREATER-OR-EQUAL-TO F L)))
    (NOTE (= X (JOIN X F L)))
    (NOTE (= F (MEET X F L)))))
```

```
(DEFTYPE DISTRIBUTIVE-LATTICE
  (LAMBDA ((L LATTICE))
    (FORALL ((X (IN-U-SET L))
             (Y (IN-U-SET L))
             (Z (IN-U-SET L)))
      (AND (= (JOIN X (MEET Y Z L) L)
              (MEET (JOIN X Y L) (JOIN X Z L) L))
           (= (MEET X (JOIN Y Z L) L)
              (JOIN (MEET X Y L) (MEET X Z L) L))))))
```

```
(LEMMA
  (FORALL ((L POWER-LATTICE)
           (S1 (IN-U-SET L))
           (S2 (IN-U-SET L))
           (S3 (IN-U-SET L)))
    (= (JOIN S1 (MEET S2 S3 L) L)
       (UNION
         S1
         (INTERSECTION S2 S3)))))

(LEMMA
  (FORALL ((L POWER-LATTICE)
           (S2 (IN-U-SET L))
           (S1 (IN-U-SET L))
           (S3 (IN-U-SET L)))
    (= (MEET (JOIN S1 S2 L)
             (JOIN S1 S3 L)
             L)
       (INTERSECTION
         (UNION S1 S2)
         (UNION S1 S3)))))

(LEMMA
  (FORALL ((L POWER-LATTICE)
           (S1 (IN-U-SET L))
           (S2 (IN-U-SET L))
           (S3 (IN-U-SET L)))
    (= (MEET S1 (JOIN S2 S3 L) L)
       (INTERSECTION S1 (UNION S2
S3)))))

(LEMMA
  (FORALL ((L POWER-LATTICE)
           (S2 (IN-U-SET L))
           (S1 (IN-U-SET L))
           (S3 (IN-U-SET L)))
    (= (JOIN (MEET S1 S2 L)
             (MEET S1 S3 L)
             L)
       (UNION (INTERSECTION S1 S2)
              (INTERSECTION S1 S3)))))

(LEMMA
  (FORALL ((L POWER-LATTICE))
    (IS L DISTRIBUTIVE-LATTICE)))
```

```
(IN-CONTEXT ((LET-BE L POWER-LATTICE)
             (LET-BE S1 (IN-U-SET L))
             (LET-BE S2 (IN-U-SET L))
             (LET-BE S3 (IN-U-SET L)))

  (IN-CONTEXT ((LET-BE M23 (MEET S2 S3 L)))
    (NOTE (= (JOIN S1 M23 L)
             (UNION S1 (INTERSECTION S2 S3)))))

  (IN-CONTEXT ((LET-BE J12 (JOIN S1 S2 L))
               (LET-BE J13 (JOIN S1 S3 L)))
    (NOTE (= (MEET J12 J13 L)
             (INTERSECTION (UNION S1 S2)
                           (UNION S1 S3)))))

  (IN-CONTEXT ((LET-BE J23 (JOIN S2 S3 L)))
    (NOTE (= (MEET S1 J23 L)
             (INTERSECTION S1 (UNION S2 S3)))))

  (IN-CONTEXT ((LET-BE M12 (MEET S1 S2 L))
               (LET-BE M13 (MEET S1 S3 L)))
    (NOTE (= (JOIN M12 M13 L)
             (INTERSECTION (UNION S1 S2)
                           (UNION S1 S3)))))

  (NOTE (IS L DISTRIBUTIVE-LATTICE)))
```

```
(DEFTYPE (COMPLEMENT-OF (X (IN-U-SET L))
                        (L BOUNDED-LATTICE))
  (LAMBDA ((Y (IN-U-SET L)))
    (AND (= (MEET X Y L)
            (BOTTOM L))
         (= (JOIN X Y L)
            (TOP L)))))
```

```
(DEFTYPE COMPLEMENTED-LATTICE
  (LAMBDA ((L BOUNDED-LATTICE))
    (FORALL ((X (IN-U-SET L)))
      (EXISTS-SOME
        (COMPLEMENT-OF X L)))))
```

```
(LEMMA
  (FORALL ((L POWER-LATTICE)
           (S1 (IN-U-SET L)))
    (IS (SET-DIFFERENCE
          (FAMILY-UNION (U-SET L))
          S1)
      (COMPLEMENT-OF S1 L))))
```

```
(IN-CONTEXT
    ((LET-BE L POWER-LATTICE)
     (LET-BE UNIVERSE (FAMILY-UNION (U-SET L)))
     (LET-BE S1 (IN-U-SET L))
     (LET-BE S2 (SET-DIFFERENCE UNIVERSE S1)))
  (NOTE (IS S2 (COMPLEMENT-OF S1 L)))
  (NOTE (IS L COMPLEMENTED-LATTICE)))
```

```
(LEMMA
  (FORALL ((L POWER-LATTICE))
    (IS L COMPLEMENTED-LATTICE)))
```

```
(LEMMA
  (EXISTS-SOME
    (AND-TYPE DISTRIBUTIVE-LATTICE
              BOUNDED-LATTICE)))
```

```
(IN-CONTEXT ((LET-BE L POWER-LATTICE))
  (NOTE
    (EXISTS-SOME
      (AND-TYPE DISTRIBUTIVE-LATTICE
                BOUNDED-LATTICE))))
```

```
(LEMMA
  (FORALL ((L (AND-TYPE
                DISTRIBUTIVE-LATTICE
                BOUNDED-LATTICE))
           (X (IN-U-SET L)))
    (AT-MOST-ONE (COMPLEMENT-OF X L))))
```

```
(IN-CONTEXT
    ((LET-BE L (AND-TYPE DISTRIBUTIVE-LATTICE
                         BOUNDED-LATTICE))
     (LET-BE X (IN-U-SET L))
     (PUSH-GOAL (AT-MOST-ONE (COMPLEMENT-OF X L))))
   (IN-CONTEXT
       ((SUPPOSE
          (EXISTS-SOME (COMPLEMENT-OF X L)))
        (LET-BE Y1 (COMPLEMENT-OF X L))
        (LET-BE Y2 (COMPLEMENT-OF X L)))
     (NOTE-GOAL))
   (NOTE-GOAL))
```

```
(DEFTYPE BOOLEAN-LATTICE
  (AND-TYPE DISTRIBUTIVE-LATTICE
            COMPLEMENTED-LATTICE))
```

```
(DEFTERM (COMPLEMENT
           (X (IN-U-SET B))
           (B BOOLEAN-LATTICE))
  (THE (COMPLEMENT-OF X B)))
```

```
(LEMMA
  (EXISTS-SOME BOOLEAN-LATTICE))
```

```
(IN-CONTEXT ((LET-BE L POWER-LATTICE))
  (NOTE (EXISTS-SOME BOOLEAN-LATTICE)))
```

```
(LEMMA
  (FORALL ((B BOOLEAN-LATTICE)
           (X (IN-U-SET B))
           (Y (IN-U-SET B)))
    (= (COMPLEMENT (MEET X Y B) B)
       (JOIN (COMPLEMENT X B)
             (COMPLEMENT Y B)
             B))))
(LEMMA
  (FORALL ((B BOOLEAN-LATTICE)
           (X (IN-U-SET B))
           (Y (IN-U-SET B)))
    (= (COMPLEMENT (JOIN X Y B) B)
       (MEET (COMPLEMENT X B)
             (COMPLEMENT Y B)
             B))))
```

```
(IN-CONTEXT
    ((LET-BE B BOOLEAN-LATTICE)
     (LET-BE X (IN-U-SET B))
     (LET-BE Y (IN-U-SET B))
     (LET-BE CX (COMPLEMENT X B))
     (LET-BE CY (COMPLEMENT Y B)))
  (IN-CONTEXT ((LET-BE M (MEET X Y B))
               (LET-BE J (JOIN CX CY B)))
    (NOTE (= (COMPLEMENT M B) J)))
  (IN-CONTEXT ((LET-BE J (JOIN X Y B))
               (LET-BE M (MEET CX CY B)))
    (NOTE (= (COMPLEMENT J B) M))))
```

```
(LEMMA
  (FORALL ((B BOOLEAN-LATTICE)
           (X (IN-U-SET B))
           (Y (IN-U-SET B)))
    (= (MEET X Y B)
       (COMPLEMENT
         (JOIN (COMPLEMENT X B)
               (COMPLEMENT Y B)
               B)
               B))))
(LEMMA
  (FORALL ((B BOOLEAN-LATTICE)
           (X (IN-U-SET B))
           (Y (IN-U-SET B)))
    (= (JOIN X Y B)
       (COMPLEMENT
         (MEET (COMPLEMENT X B)
               (COMPLEMENT Y B)
               B)
         B))))
```

```
(IN-CONTEXT ((LET-BE B BOOLEAN-LATTICE)
             (LET-BE X (IN-U-SET B))
             (LET-BE Y (IN-U-SET B)))
  (IN-CONTEXT ((LET-BE M (MEET X Y B))
               (LET-BE J (JOIN (COMPLEMENT X B)
                               (COMPLEMENT Y B)
                               B)))
    (NOTE (= M (COMPLEMENT J B))))
  (IN-CONTEXT ((LET-BE J (JOIN X Y B))
               (LET-BE M (MEET (COMPLEMENT X B)
                               (COMPLEMENT Y B)
                               B)))
    (NOTE (= J (COMPLEMENT M B)))))
```

```
;the following are equivalent:
;
; (IS X (LESS-OR-EQUAL-TO Y B))
;
; (IS (COMPLEMENT Y B)
;      (LESS-OR-EQUAL-TO
;        (COMPLEMENT X B)
;        B))
;
; (= (MEET X (COMPLEMENT Y B) B)
;     (BOTTOM B))
;
; (= (JOIN (COMPLEMENT X B) Y B)
;    (TOP B))

(LEMMA
  (FORALL ((B BOOLEAN-LATTICE)
           (X (IN-U-SET B))
           (Y (IN-U-SET B)))
    (=> (IS X
            (LESS-OR-EQUAL-TO Y B))
        (IS (COMPLEMENT Y B)
            (LESS-OR-EQUAL-TO
              (COMPLEMENT X B)
              B)))))

(LEMMA
  (FORALL ((B BOOLEAN-LATTICE)
           (X (IN-U-SET B))
           (Y (IN-U-SET B)))
    (=> (IS (COMPLEMENT Y B)
            (LESS-OR-EQUAL-TO
              (COMPLEMENT X B)
              B))
        (= (MEET X (COMPLEMENT Y B) B)
           (BOTTOM B)))))

(LEMMA
  (FORALL ((B BOOLEAN-LATTICE)
           (X (IN-U-SET B))
           (Y (IN-U-SET B)))
    (=> (= (MEET X (COMPLEMENT Y B) B)
           (BOTTOM B))
        (= (JOIN (COMPLEMENT X B) Y B)
           (TOP B)))))

(LEMMA
  (FORALL ((B BOOLEAN-LATTICE)
           (Y (IN-U-SET B))
           (X (IN-U-SET B)))
    (=> (= (JOIN (COMPLEMENT X B) Y B)
           (TOP B))
        (IS X
            (LESS-OR-EQUAL-TO Y B)))))

(IN-CONTEXT ((LET-BE B BOOLEAN-LATTICE)
             (LET-BE X (IN-U-SET B))
             (LET-BE Y (IN-U-SET B)))

  (IN-CONTEXT
      ((SUPPOSE (IS X (LESS-OR-EQUAL-TO Y B)))
       (PUSH-GOAL (IS (COMPLEMENT Y B)
                      (LESS-OR-EQUAL-TO
                        (COMPLEMENT X B)
                        B))))
    (IN-CONTEXT ((LET-BE CX (COMPLEMENT X B))
                 (LET-BE CY (COMPLEMENT Y B)))
      (NOTE-GOAL)))

  (IN-CONTEXT
      ((SUPPOSE (IS (COMPLEMENT Y B)
                    (LESS-OR-EQUAL-TO
                      (COMPLEMENT X B)
                      B)))
       (PUSH-GOAL (= (MEET X (COMPLEMENT Y B) B)
                     (BOTTOM B))))
    (IN-CONTEXT ((LET-BE CX (COMPLEMENT X B))
                 (LET-BE CY (COMPLEMENT Y B)))
      (NOTE-GOAL)))

  (IN-CONTEXT
      ((SUPPOSE (= (MEET X (COMPLEMENT Y B) B)
                   (BOTTOM B)))
       (PUSH-GOAL (= (JOIN (COMPLEMENT X B) Y B)
                     (TOP B))))
    (IN-CONTEXT ((LET-BE CY (COMPLEMENT Y B))
                 (LET-BE J
                   (JOIN (COMPLEMENT X B) Y B)))
      (NOTE-GOAL)))

  (IN-CONTEXT
      ((SUPPOSE (= (JOIN (COMPLEMENT X B) Y B)
                   (TOP B)))
       (PUSH-GOAL (IS X (LESS-OR-EQUAL-TO Y B))))
    (IN-CONTEXT ((LET-BE CX (COMPLEMENT X B))
                 (LET-BE M (MEET X Y B)))
      (NOTE-GOAL))))
```

A.8 Sublattices

A lattice subset of a Boolean lattice is a subset that is closed under the meet and join operations of the lattice. The poset which results from restricting the order in L to lattice subset of L is called a *lattice subalgebra* of L. We prove that a lattice subalgebra of L is a lattice with the same lattice operations as L.

A Boolean subset of Boolean lattice is a lattice subset which is also closed under taking complements; from deMorgan's laws it is sufficient that the subset be closed under intersection and complement or union and completement. The poset which results from restricting the order of a boolean lattice L to a Boolean subset of L is called a *Boolean subalgebra* of L. We prove that a Boolean subalgebra of L is a Boolean lattice with the same Boolean operations as L.

```
(DEFTYPE (FINITE-MEET-SUBSET-OF (B LATTICE))
  (LAMBDA ((S (NON-EMPTY-SUBSET-OF (U-SET B))))
    (FORALL ((X (MEMBER-OF S)))
      (FORALL ((Y (MEMBER-OF S)))
        (IS (MEET X Y B) (MEMBER-OF S))))))

(DEFTYPE (FINITE-JOIN-SUBSET-OF (B LATTICE))
  (LAMBDA ((S (NON-EMPTY-SUBSET-OF (U-SET B))))
    (FORALL ((X (MEMBER-OF S)))
      (FORALL ((Y (MEMBER-OF S)))
        (IS (JOIN X Y B) (MEMBER-OF S))))))

(DEFTYPE (LATTICE-SUBSET-OF (L LATTICE))
  (AND-TYPE (FINITE-MEET-SUBSET-OF L)
            (FINITE-JOIN-SUBSET-OF L)))
```

```
(LEMMA
  (FORALL ((L LATTICE))
    (IS (U-SET L)
        (LATTICE-SUBSET-OF L))))
```

```
(IN-CONTEXT ((LET-BE L LATTICE)
             (LET-BE S (U-SET L))
             (PUSH-GOAL
               (IS S (LATTICE-SUBSET-OF L))))
  (IN-CONTEXT ((LET-BE X (IN-U-SET L))
               (LET-BE Y (IN-U-SET L)))
    (IN-CONTEXT ((LET-BE M (MEET X Y L)))
      (NOTE (IS M (MEMBER-OF S))))
    (IN-CONTEXT ((LET-BE J (JOIN X Y L)))
      (NOTE (IS J (MEMBER-OF S))))
    (NOTE-GOAL)))
```

```
(DEFTYPE (LATTICE-SUBALGEBRA-OF (L LATTICE))
  (WRITABLE-AS (RESTRICT-ORDER L S)
    (S (LATTICE-SUBSET-OF L))))
```

```
(LEMMA                                    (IN-CONTEXT ((LET-BE L LATTICE)
  (FORALL ((L LATTICE))                               (LET-BE S (U-SET L))
    (EXISTS-SOME                                      (LET-BE L2 (RESTRICT-ORDER L S)))
      (LATTICE-SUBALGEBRA-OF L))))          (NOTE
                                              (EXISTS-SOME
                                                (LATTICE-SUBALGEBRA-OF L))))
```

```
(LEMMA                                    (IN-CONTEXT
  (FORALL                                     ((LET-BE L1 LATTICE)
      ((L1 LATTICE)                            (LET-BE L2 (LATTICE-SUBALGEBRA-OF L1))
       (L2 (LATTICE-SUBALGEBRA-OF L1)))        (WRITE-AS L2 (RESTRICT-ORDER L1 S)
    (IS (U-SET L2)                                       (S (LATTICE-SUBSET-OF L1))))
        (LATTICE-SUBSET-OF L1))))           (NOTE (IS (U-SET L2) (LATTICE-SUBSET-OF L1))))
```

```
(LEMMA
  (FORALL
      ((L1 LATTICE)
       (L2 (LATTICE-SUBALGEBRA-OF L1))
       (X (IN-U-SET L2)))
    (IS X (IN-U-SET L1))))

(LEMMA
  (FORALL
      ((L1 LATTICE)
       (L2 (LATTICE-SUBALGEBRA-OF L1))
       (X (IN-U-SET L2))
       (Y (IN-U-SET L2)))
    (IS (JOIN X Y L1)
        (LEAST-UPPER-BOUND-OF
          (MAKE-SET X Y)
          L2))))

(LEMMA
  (FORALL
      ((L1 LATTICE)
       (L2 (LATTICE-SUBALGEBRA-OF L1))
       (X (IN-U-SET L2))
       (Y (IN-U-SET L2)))
    (IS (MEET X Y L1)
        (GREATEST-LOWER-BOUND-OF
          (MAKE-SET X Y)
          L2))))

(LEMMA
  (FORALL
      ((L1 LATTICE)
       (L2 (LATTICE-SUBALGEBRA-OF L1)))
    (IS L2 LATTICE)))

(LEMMA
  (FORALL
      ((L1 LATTICE)
       (L2 (LATTICE-SUBALGEBRA-OF L1))
       (X (IN-U-SET L2))
       (Y (IN-U-SET L2)))
    (= (JOIN X Y L1)
       (JOIN X Y L2))))

(LEMMA
  (FORALL
      ((L1 LATTICE)
       (L2 (LATTICE-SUBALGEBRA-OF L1))
       (X (IN-U-SET L2))
       (Y (IN-U-SET L2)))
    (= (MEET X Y L1)
       (MEET X Y L2))))
```

```
(IN-CONTEXT
    ((LET-BE L1 LATTICE)
     (LET-BE L2 (LATTICE-SUBALGEBRA-OF L1))
     (LET-BE X (IN-U-SET L2))
     (LET-BE Y (IN-U-SET L2))
     (WRITE-AS L2 (RESTRICT-ORDER L1 S)
               (S (LATTICE-SUBSET-OF L1))))

  (NOTE (IS X (IN-U-SET L1)))

  (IN-CONTEXT ((LET-BE S (MAKE-SET X Y)))
    (IN-CONTEXT
      ((LET-BE J (JOIN X Y L1))
       (PUSH-GOAL
         (IS J (LEAST-UPPER-BOUND-OF S L2)))
       (LET-BE Z (UPPER-BOUND-OF S L2)))
      (NOTE-GOAL))
    (IN-CONTEXT
      ((LET-BE M (MEET X Y L1))
       (PUSH-GOAL
         (IS M (GREATEST-LOWER-BOUND-OF S L2)))
       (LET-BE Z (LOWER-BOUND-OF S L2)))
      (NOTE-GOAL)))

  (NOTE (IS L2 LATTICE))

  (IN-CONTEXT ((LET-BE J (JOIN X Y L1)))
    (NOTE (= (JOIN X Y L1) (JOIN X Y L2))))

  (IN-CONTEXT ((LET-BE M (MEET X Y L1)))
    (NOTE (= (MEET X Y L1) (MEET X Y L2)))))
```

```
(LEMMA
  (FORALL
      ((L1 LATTICE)
       (L2 (LATTICE-SUBALGEBRA-OF L1))
       (Z (IN-U-SET L2))
       (X (IN-U-SET L2))
       (Y (IN-U-SET L2)))
    (= (MEET Z (JOIN X Y L2) L2)
       (MEET Z (JOIN X Y L1) L1))))

(LEMMA
  (FORALL
      ((L1 LATTICE)
       (L2 (LATTICE-SUBALGEBRA-OF L1))
       (Z (IN-U-SET L2))
       (X (IN-U-SET L2))
       (Y (IN-U-SET L2)))
    (= (JOIN Z (JOIN X Y L2) L2)
       (JOIN Z (JOIN X Y L1) L1))))

(LEMMA
  (FORALL
      ((L1 LATTICE)
       (L2 (LATTICE-SUBALGEBRA-OF L1))
       (X (IN-U-SET L2))
       (Z (IN-U-SET L2))
       (Y (IN-U-SET L2)))
    (= (MEET (JOIN X Y L2)
             (JOIN Z Y L2)
             L2)
       (MEET (JOIN X Y L1)
             (JOIN Z Y L1)
             L1))))
```

```
(IN-CONTEXT
    ((LET-BE L1 LATTICE)
     (LET-BE L2 (LATTICE-SUBALGEBRA-OF L1))
     (LET-BE X (IN-U-SET L2))
     (LET-BE Y (IN-U-SET L2))
     (LET-BE Z (IN-U-SET L2))
     (WRITE-AS L2 (RESTRICT-ORDER L1 S)
               (S (LATTICE-SUBSET-OF L1)))
     (LET-BE J (JOIN X Y L2)))

  (NOTE (= (MEET Z (JOIN X Y L2) L2)
           (MEET Z (JOIN X Y L1) L1)))
  (NOTE (= (JOIN Z (JOIN X Y L2) L2)
           (JOIN Z (JOIN X Y L1) L1)))
  (IN-CONTEXT ((LET-BE J2 (JOIN Z Y L2)))
    (NOTE (= (MEET (JOIN X Y L2)
                   (JOIN Z Y L2)
                   L2)
             (MEET (JOIN X Y L1)
                   (JOIN Z Y L1)
                   L1)))))
```

```
(LEMMA
  (FORALL
      ((L1 LATTICE)
       (L2 (LATTICE-SUBALGEBRA-OF L1))
       (Z (IN-U-SET L2))
       (X (IN-U-SET L2))
       (Y (IN-U-SET L2)))
    (= (JOIN Z (MEET X Y L2)  L2)
       (JOIN Z (MEET X Y L1)  L1))))

(LEMMA
  (FORALL
      ((L1 LATTICE)
       (L2 (LATTICE-SUBALGEBRA-OF L1))
       (Z (IN-U-SET L2))
       (X (IN-U-SET L2))
       (Y (IN-U-SET L2)))
    (= (MEET Z (MEET X Y L2) L2)
       (MEET Z (MEET X Y L1) L1))))

(LEMMA
  (FORALL
      ((L1 LATTICE)
       (L2 (LATTICE-SUBALGEBRA-OF L1))
       (X (IN-U-SET L2))
       (Z (IN-U-SET L2))
       (Y (IN-U-SET L2)))
    (= (JOIN (MEET X Y L2)
             (MEET Z Y L2)
             L2)
       (JOIN (MEET X Y L1)
             (MEET Z Y L1)
             L1))))

(LEMMA
 (FORALL ((L1 LATTICE))
  (=>
   (IS L1 DISTRIBUTIVE-LATTICE)
   (FORALL
       ((L2 (LATTICE-SUBALGEBRA-OF L1)))
     (IS L2 DISTRIBUTIVE-LATTICE)))))
```

```
(IN-CONTEXT
    ((LET-BE L1 LATTICE)
     (LET-BE L2 (LATTICE-SUBALGEBRA-OF L1))
     (LET-BE X (IN-U-SET L2))
     (LET-BE Y (IN-U-SET L2))
     (LET-BE Z (IN-U-SET L2))
     (WRITE-AS L2 (RESTRICT-ORDER L1 S)
               (S (LATTICE-SUBSET-OF L1))))

  (IN-CONTEXT ((LET-BE M (MEET X Y L2)))
    (NOTE (= (JOIN Z (MEET X Y L2) L2)
             (JOIN Z (MEET X Y L1) L1)))
    (NOTE (= (MEET Z (MEET X Y L2) L2)
             (MEET Z (MEET X Y L1) L1)))
    (IN-CONTEXT ((LET-BE M2 (MEET Z Y L2)))
      (NOTE (= (JOIN (MEET X Y L2)
                     (MEET Z Y L2)
                     L2)
               (JOIN (MEET X Y L1)
                     (MEET Z Y L1)
                     L1)))))

  (IN-CONTEXT
      ((SUPPOSE (IS L1 DISTRIBUTIVE-LATTICE)))
    (NOTE (IS L2 DISTRIBUTIVE-LATTICE))))
```

```
(DEFTYPE (COMPLEMENTED-SUBSET-OF (B BOOLEAN-LATTICE))
  (LAMBDA ((S (NON-EMPTY-SUBSET-OF (U-SET B))))
    (FORALL ((X (MEMBER-OF S)))
      (IS (COMPLEMENT X B) (MEMBER-OF S)))))

(DEFTYPE (BOOLEAN-SUBSET-OF (B BOOLEAN-LATTICE))
  (AND-TYPE (FINITE-MEET-SUBSET-OF B)
            (FINITE-JOIN-SUBSET-OF B)
            (COMPLEMENTED-SUBSET-OF B)))
```

```
(LEMMA
  (FORALL ((B BOOLEAN-LATTICE)
           (S (SUBSET-OF (U-SET B))))
    (=>
      (IS S
        (AND-TYPE
          (FINITE-MEET-SUBSET-OF B)
          (COMPLEMENTED-SUBSET-OF B)))
      (IS S (BOOLEAN-SUBSET-OF B)))))
```

```
(IN-CONTEXT ((LET-BE B BOOLEAN-LATTICE)
             (LET-BE S (SUBSET-OF (U-SET B))))
  (IN-CONTEXT
    ((SUPPOSE
       (IS S
           (AND-TYPE
             (FINITE-MEET-SUBSET-OF B)
             (COMPLEMENTED-SUBSET-OF B))))
     (PUSH-GOAL (IS S (BOOLEAN-SUBSET-OF B))))
    (IN-CONTEXT ((LET-BE X (MEMBER-OF S))
                 (LET-BE Y (MEMBER-OF S)))
      (IN-CONTEXT ((LET-BE CX (COMPLEMENT X B))
                   (LET-BE CY (COMPLEMENT Y B)))
        (NOTE (IS (MEET CX CY B) (MEMBER-OF S))))
      (IN-CONTEXT ((LET-BE J (JOIN X Y B))
                   (LET-BE M
                     (MEET (COMPLEMENT X B)
                           (COMPLEMENT Y B) B)))
        (NOTE-GOAL)))))
```

```
(LEMMA
  (FORALL ((B BOOLEAN-LATTICE)
           (S (SUBSET-OF (U-SET B))))
    (=>
      (IS S
        (AND-TYPE
          (FINITE-JOIN-SUBSET-OF B)
          (COMPLEMENTED-SUBSET-OF B)))
      (IS S (BOOLEAN-SUBSET-OF B)))))
```

```
(IN-CONTEXT ((LET-BE B BOOLEAN-LATTICE)
             (LET-BE S (SUBSET-OF (U-SET B))))
  (IN-CONTEXT
    ((SUPPOSE
         (IS S
           (AND-TYPE (FINITE-JOIN-SUBSET-OF B)
                     (COMPLEMENTED-SUBSET-OF B))))
       (PUSH-GOAL (IS S (BOOLEAN-SUBSET-OF B))))
    (IN-CONTEXT ((LET-BE X (MEMBER-OF S))
                 (LET-BE Y (MEMBER-OF S)))
      (IN-CONTEXT ((LET-BE CX (COMPLEMENT X B))
                   (LET-BE CY (COMPLEMENT Y B)))
        (NOTE (IS (JOIN CX CY B) (MEMBER-OF S))))
      (IN-CONTEXT ((LET-BE M (MEET X Y B))
                   (LET-BE J
                     (JOIN (COMPLEMENT X B)
                           (COMPLEMENT Y B) B)))
        (NOTE-GOAL)))))
```

```
(DEFTYPE (BOOLEAN-SUBALGEBRA-OF (B BOOLEAN-LATTICE))
  (WRITABLE-AS (RESTRICT-ORDER B S)
    (S (BOOLEAN-SUBSET-OF B))))
```

```
(LEMMA
  (FORALL ((B BOOLEAN-LATTICE))
    (IS (U-SET B)
        (BOOLEAN-SUBSET-OF B))))
```

```
(IN-CONTEXT ((LET-BE B BOOLEAN-LATTICE)
             (LET-BE S (U-SET B))
             (PUSH-GOAL
               (IS S (BOOLEAN-SUBSET-OF B))))
  (IN-CONTEXT ((LET-BE X (IN-U-SET B)))
    (IN-CONTEXT ((LET-BE CX (COMPLEMENT X B)))
      (NOTE (IS CX (MEMBER-OF S))))
    (IN-CONTEXT ((LET-BE Y (IN-U-SET B)))
      (IN-CONTEXT ((LET-BE M (MEET X Y B)))
        (NOTE (IS M (MEMBER-OF S))))
      (NOTE-GOAL))))
```

```
(LEMMA
  (FORALL
      ((B1 BOOLEAN-LATTICE)
       (B2 (BOOLEAN-SUBALGEBRA-OF B1)))
    (IS B2 (LATTICE-SUBALGEBRA-OF B1))))

(LEMMA
  (FORALL
      ((B1 BOOLEAN-LATTICE)
       (B2 (BOOLEAN-SUBALGEBRA-OF B1)))
    (IS (TOP B1) (IN-U-SET B2))))
(LEMMA
  (FORALL
      ((B1 BOOLEAN-LATTICE)
       (B2 (BOOLEAN-SUBALGEBRA-OF B1)))
    (IS (BOTTOM B1) (IN-U-SET B2))))

(LEMMA
 (FORALL
     ((B1 BOOLEAN-LATTICE)
      (B2 (BOOLEAN-SUBALGEBRA-OF B1)))
  (IS (TOP B1)
    (GREATEST-MEMBER-OF (U-SET B2)
                        B2))))

(LEMMA
 (FORALL
     ((B1 BOOLEAN-LATTICE)
      (B2 (BOOLEAN-SUBALGEBRA-OF B1)))
  (IS (BOTTOM B1)
    (LEAST-MEMBER-OF (U-SET B2) B2))))

(LEMMA
  (FORALL
      ((B1 BOOLEAN-LATTICE)
       (B2 (BOOLEAN-SUBALGEBRA-OF B1)))
    (= (TOP B2) (TOP B1))))

(LEMMA
  (FORALL
      ((B1 BOOLEAN-LATTICE)
       (B2 (BOOLEAN-SUBALGEBRA-OF B1)))
    (= (BOTTOM B2) (BOTTOM B1))))

(LEMMA
  (FORALL
      ((B1 BOOLEAN-LATTICE)
       (B2 (BOOLEAN-SUBALGEBRA-OF B1)))
    (IS B2 COMPLEMENTED-LATTICE)))

(LEMMA
  (FORALL
      ((B1 BOOLEAN-LATTICE)
       (B2 (BOOLEAN-SUBALGEBRA-OF B1))
       (X (IN-U-SET B2)))
    (= (COMPLEMENT X B2)
       (COMPLEMENT X B1))))
```

```
(IN-CONTEXT
    ((LET-BE B1 BOOLEAN-LATTICE)
     (LET-BE B2 (BOOLEAN-SUBALGEBRA-OF B1))
     (WRITE-AS B2 (RESTRICT-ORDER B1 S)
       (S (BOOLEAN-SUBSET-OF B1))))

  (NOTE (IS B2 (LATTICE-SUBALGEBRA-OF B1)))

  (IN-CONTEXT ((LET-BE X (IN-U-SET B2)))
    (IN-CONTEXT ((LET-BE CX (COMPLEMENT X B1)))
      ;top = (join x cx b1)
      (NOTE (IS (TOP B1) (IN-U-SET B2)))
      ;bottom = (meet x cx b1)
      (NOTE (IS (BOTTOM B1) (IN-U-SET B2)))))

  (IN-CONTEXT ((LET-BE T (TOP B1))
               (LET-BE X (IN-U-SET B2)))
   (NOTE
    (IS T
        (GREATEST-MEMBER-OF (U-SET B2) B2))))

  (IN-CONTEXT ((LET-BE F (BOTTOM B1))
               (LET-BE X (IN-U-SET B2)))
    (NOTE
      (IS F
          (LEAST-MEMBER-OF (U-SET B2) B2))))

  (IN-CONTEXT ((LET-BE T (TOP B1)))
    (NOTE (= (TOP B2) (TOP B1))))

  (IN-CONTEXT ((LET-BE F (BOTTOM B1)))
    (NOTE (= (BOTTOM B2) (BOTTOM B1))))

  (IN-CONTEXT ((LET-BE X (IN-U-SET B2))
               (LET-BE CX (COMPLEMENT X B1)))
    (NOTE (IS B2 COMPLEMENTED-LATTICE))
    (NOTE (= (COMPLEMENT X B2)
             (COMPLEMENT X B1)))))
```

A.9 Lattice Morphisms

A *Boolean homomorphism* is a map between Boolean lattices which commutes with meet, join, and complementation. By deMorgan's laws it suffices that the map commute with meet and completentation or join and complementation. The image of a Boolean homorphism is a Boolean subset of the range lattice. A *Boolean isomorphism* is a bijective Boolean homomorphism.

```
(DEFTYPE LATTICE-MAP
  (LAMBDA ((H MAP))
    (AND (IS (DOMAIN H) LATTICE)
         (IS (RANGE H) LATTICE))))

(DEFTYPE MAP-WHICH-RESPECTS-JOIN
  (LAMBDA ((H LATTICE-MAP))
    (FORALL ((X (IN-MAP-DOMAIN H))
             (Y (IN-MAP-DOMAIN H)))
      (= (APPLY-MAP H (JOIN X Y (DOMAIN H)))
         (JOIN (APPLY-MAP H X)
               (APPLY-MAP H Y)
               (RANGE H))))))

(DEFTYPE MAP-WHICH-RESPECTS-MEET
  (LAMBDA ((H LATTICE-MAP))
    (FORALL ((X (IN-MAP-DOMAIN H))
             (Y (IN-MAP-DOMAIN H)))
      (= (APPLY-MAP H (MEET X Y (DOMAIN H)))
         (MEET (APPLY-MAP H X)
               (APPLY-MAP H Y)
               (RANGE H))))))

(DEFTYPE BOOLEAN-MAP
  (LAMBDA ((H LATTICE-MAP))
    (AND (IS (DOMAIN H)
             BOOLEAN-LATTICE)
         (IS (RANGE H)
             BOOLEAN-LATTICE))))

(DEFTYPE MAP-WHICH-RESPECTS-COMPLEMENT
  (LAMBDA ((H BOOLEAN-MAP))
    (FORALL ((X (IN-MAP-DOMAIN H)))
      (= (APPLY-MAP H (COMPLEMENT X (DOMAIN H)))
         (COMPLEMENT (APPLY-MAP H X)
                     (RANGE H))))))
```

```
(DEFTYPE BOOLEAN-HOMOMORPHISM
  (AND-TYPE MAP-WHICH-RESPECTS-JOIN
            MAP-WHICH-RESPECTS-MEET
            MAP-WHICH-RESPECTS-COMPLEMENT))

(DEFTYPE (BOOLEAN-HOMOMORPHISM-BETWEEN
           (B1 BOOLEAN-LATTICE)
           (B2 BOOLEAN-LATTICE))
  (LAMBDA ((H (MAP-BETWEEN B1 B2)))
    (IS H BOOLEAN-HOMOMORPHISM)))

(DEFTYPE BOOLEAN-ISOMORPHISM
  (AND-TYPE BOOLEAN-HOMOMORPHISM
            BIJECTION))

(DEFTYPE (BOOLEAN-ISOMORPHISM-BETWEEN
           (B1 BOOLEAN-LATTICE)
           (B2 BOOLEAN-LATTICE))
  (AND-TYPE (BOOLEAN-HOMOMORPHISM-BETWEEN B1 B2)
            BIJECTION))

(DEFTYPE (BOOLEAN-LATTICE-ISOMORPHIC-TO
           (B1 BOOLEAN-LATTICE))
  (LAMBDA ((B2 BOOLEAN-LATTICE))
    (EXISTS-SOME
      (BOOLEAN-ISOMORPHISM-BETWEEN B1 B2))))
```

```
(LEMMA                              (IN-CONTEXT ((LET-BE L LATTICE)
  (EXISTS-SOME LATTICE-MAP))                     (LET-BE I (IDENTITY-MAP L)))
                                      (NOTE (EXISTS-SOME LATTICE-MAP)))


(LEMMA                              (IN-CONTEXT ((LET-BE B BOOLEAN-LATTICE)
  (EXISTS-SOME BOOLEAN-MAP))                     (LET-BE I (IDENTITY-MAP B)))
                                      (NOTE (EXISTS-SOME BOOLEAN-MAP)))
```

```
(LEMMA
 (FORALL ((H BOOLEAN-MAP))
  (=>
   (AND
     (IS H
       MAP-WHICH-RESPECTS-COMPLEMENT)
     (IS H
       MAP-WHICH-RESPECTS-JOIN))

   (IS H MAP-WHICH-RESPECTS-MEET))))


(LEMMA
 (FORALL ((H BOOLEAN-MAP))
  (=>
   (AND
     (IS H
       MAP-WHICH-RESPECTS-COMPLEMENT)
     (IS H
       MAP-WHICH-RESPECTS-MEET))

   (IS H MAP-WHICH-RESPECTS-JOIN))))
```

```
(IN-CONTEXT ((LET-BE H BOOLEAN-MAP)
             (LET-BE B1 (DOMAIN H))
             (LET-BE B2 (RANGE H)))

  (IN-CONTEXT
      ((SUPPOSE
         (IS H MAP-WHICH-RESPECTS-JOIN))
       (SUPPOSE
         (IS H MAP-WHICH-RESPECTS-COMPLEMENT))
       (PUSH-GOAL
         (IS H MAP-WHICH-RESPECTS-MEET)))
    (IN-CONTEXT ((LET-BE X (IN-U-SET B1))
                 (LET-BE Y (IN-U-SET B1)))
      (IN-CONTEXT
          ((LET-BE CX (COMPLEMENT X B1))
           (LET-BE CY (COMPLEMENT Y B1))
           (LET-BE J (JOIN CX CY B1)))
        (NOTE (= (APPLY-MAP H (MEET X Y B1))
                 (COMPLEMENT
                   (JOIN (COMPLEMENT
                           (APPLY-MAP H X)
                           B2)
                         (COMPLEMENT
                           (APPLY-MAP H Y)
                           B2)
                         B2)
                   B2))))
      (IN-CONTEXT ((LET-BE HX (APPLY-MAP H X))
                   (LET-BE HY (APPLY-MAP H Y)))
        (NOTE-GOAL))))

  (IN-CONTEXT
      ((SUPPOSE
         (IS H MAP-WHICH-RESPECTS-MEET))
       (SUPPOSE
         (IS H MAP-WHICH-RESPECTS-COMPLEMENT))
       (PUSH-GOAL
         (IS H MAP-WHICH-RESPECTS-JOIN)))
    (IN-CONTEXT ((LET-BE X (IN-U-SET B1))
                 (LET-BE Y (IN-U-SET B1)))
      (IN-CONTEXT
          ((LET-BE CX (COMPLEMENT X B1))
           (LET-BE CY (COMPLEMENT Y B1))
           (LET-BE M (MEET CX CY B1)))
        (NOTE (= (APPLY-MAP H (JOIN X Y B1))
                 (COMPLEMENT
                   (MEET (COMPLEMENT
                           (APPLY-MAP H X)
                           B2)
                         (COMPLEMENT
                           (APPLY-MAP H Y)
                           B2)
                         B2)
                   B2))))
      (IN-CONTEXT ((LET-BE HX (APPLY-MAP H X))
                   (LET-BE HY (APPLY-MAP H Y)))
        (NOTE-GOAL)))))
```

```
(LEMMA
  (FORALL ((B BOOLEAN-LATTICE))
    (IS (IDENTITY-MAP B)
        BOOLEAN-HOMOMORPHISM)))
```

```
(IN-CONTEXT
    ((LET-BE B BOOLEAN-LATTICE)
     (LET-BE I (IDENTITY-MAP B))
     (PUSH-GOAL
       (IS I BOOLEAN-HOMOMORPHISM)))
  (IN-CONTEXT ((LET-BE X (IN-U-SET B))
               (LET-BE Y (IN-U-SET B)))
    (IN-CONTEXT ((LET-BE CX (COMPLEMENT X B)))
      (NOTE (IS I MAP-WHICH-RESPECTS-COMPLEMENT)))
    (IN-CONTEXT ((LET-BE J (JOIN X Y B)))
      (NOTE (IS I MAP-WHICH-RESPECTS-JOIN))))
  (NOTE-GOAL))
```

```
(LEMMA
  (FORALL ((H BOOLEAN-HOMOMORPHISM))
    (IS (IMAGE H)
        (BOOLEAN-SUBSET-OF
          (RANGE H)))))
```

```
(IN-CONTEXT ((LET-BE H BOOLEAN-HOMOMORPHISM)
             (LET-BE B1 (DOMAIN H))
             (LET-BE B2 (RANGE H))
             (LET-BE S (IMAGE H)))
  (IN-CONTEXT
      ((PUSH-GOAL
         (IS S (BOOLEAN-SUBSET-OF B2))))
    (IN-CONTEXT
        ((LET-BE X (MEMBER-OF S))
         (LET-BE Y (MEMBER-OF S))
         (WRITE-AS X (APPLY-MAP H PRE-X)
            (PRE-X (IN-U-SET (DOMAIN H))))
         (WRITE-AS Y (APPLY-MAP H PRE-Y)
            (PRE-Y (IN-U-SET (DOMAIN H)))))
      (IN-CONTEXT
          ((LET-BE PM
             (MEET PRE-X PRE-Y (DOMAIN H))))
        (NOTE
          (IS (MEET X Y B2)
              (MEMBER-OF S))))
      (IN-CONTEXT
          ((LET-BE PC
             (COMPLEMENT PRE-X (DOMAIN H))))
        (NOTE
          (IS (COMPLEMENT X B2)
              (MEMBER-OF S))))
      (NOTE-GOAL))))
```

```
(DEFTERM (BOOLEAN-IMAGE (H BOOLEAN-HOMOMORPHISM))
  (RESTRICT-ORDER (RANGE H) (IMAGE H)))
```

```
(LEMMA
  (FORALL ((H BOOLEAN-HOMOMORPHISM))
    (IS (BOOLEAN-IMAGE H)
        (BOOLEAN-SUBALGEBRA-OF
          (RANGE H)))))

(LEMMA
  (FORALL ((H BOOLEAN-HOMOMORPHISM))
    (IS (BOOLEAN-IMAGE H)
        BOOLEAN-LATTICE)))

(LEMMA
  (FORALL ((H BOOLEAN-HOMOMORPHISM))
    (IS (BOOLEAN-IMAGE H)
        (STRUCTURE-CONTAINING
          (IMAGE H)))))

(LEMMA
  (FORALL ((H BOOLEAN-HOMOMORPHISM))
    (= (U-SET (BOOLEAN-IMAGE H))
       (IMAGE H))))
```

```
(IN-CONTEXT ((LET-BE H BOOLEAN-HOMOMORPHISM)
             (LET-BE B2 (RANGE H))
             (LET-BE S2 (IMAGE H))
             (LET-BE B3 (BOOLEAN-IMAGE H)))
  (NOTE (IS B3 (BOOLEAN-SUBALGEBRA-OF B2)))
  (NOTE (IS B3 BOOLEAN-LATTICE))
  (NOTE (IS B3 (STRUCTURE-CONTAINING (IMAGE H))))
  (NOTE (= (U-SET B3) (IMAGE H))))
```

```
(LEMMA
 (FORALL
     ((H BOOLEAN-HOMOMORPHISM)
      (X (IN-U-SET (BOOLEAN-IMAGE H))))
   (= (COMPLEMENT X
          (BOOLEAN-IMAGE H))
      (COMPLEMENT X
          (RANGE H)))))

(LEMMA
  (FORALL
      ((H BOOLEAN-HOMOMORPHISM)
       (X (IN-U-SET
            (BOOLEAN-IMAGE H)))
       (Y (IN-U-SET
            (BOOLEAN-IMAGE H))))
    (= (JOIN X Y
         (BOOLEAN-IMAGE H))
       (JOIN X Y
         (RANGE H)))))

(LEMMA
  (FORALL
      ((H BOOLEAN-HOMOMORPHISM)
       (X (IN-U-SET
            (BOOLEAN-IMAGE H)))
       (Y (IN-U-SET
            (BOOLEAN-IMAGE H))))
    (= (MEET X Y
         (BOOLEAN-IMAGE H))
       (MEET X Y
         (RANGE H)))))

(LEMMA
  (FORALL ((H BOOLEAN-HOMOMORPHISM))
    (IS (SET!-RANGE H (BOOLEAN-IMAGE H))
        BOOLEAN-HOMOMORPHISM)))

(IN-CONTEXT
    ((LET-BE H BOOLEAN-HOMOMORPHISM)
     (LET-BE BIMAGE (BOOLEAN-IMAGE H))
     (LET-BE H2 (SET!-RANGE H BIMAGE)))

  (IN-CONTEXT ((LET-BE BRANGE (RANGE H))
               (LET-BE X (IN-U-SET BIMAGE))
               (LET-BE Y (IN-U-SET BIMAGE)))
    (NOTE (= (COMPLEMENT X BIMAGE)
             (COMPLEMENT X BRANGE)))
    (NOTE (= (JOIN X Y BIMAGE)
             (JOIN X Y BRANGE)))
    (NOTE (= (MEET X Y BIMAGE)
             (MEET X Y BRANGE))))

    (IN-CONTEXT
        ((PUSH-GOAL
           (IS H2 BOOLEAN-HOMOMORPHISM))
         (LET-BE BDOMAIN (DOMAIN H))
         (LET-BE X (IN-U-SET BDOMAIN))
         (LET-BE Y (IN-U-SET BDOMAIN))
         (LET-BE HX (APPLY-MAP H2 X))
         (LET-BE HY (APPLY-MAP H2 Y)))
      (IN-CONTEXT
          ((LET-BE CX (COMPLEMENT X BDOMAIN))
           (LET-BE HCX (APPLY-MAP H2 CX)))
        (NOTE
          (IS H2
            MAP-WHICH-RESPECTS-COMPLEMENT)))
      (IN-CONTEXT
          ((LET-BE MX (MEET X Y BDOMAIN))
           (LET-BE HMX (APPLY-MAP H2 MX)))
        (NOTE
          (IS H2 MAP-WHICH-RESPECTS-MEET)))
      (NOTE-GOAL)))
```

A.10 Filters and Ultrafilters

A *filter* in a bounded lattice L is a subset F of L which satisfies the following conditions:

- F does not contain the least member of L.
- If x is in F then every member of L greater than x is in F.
- If x and y are in L then the meet of x and y are in L.

If x is a member of a bounded lattice L then the *filter generated by* x is the set of all members of L greater than or equal to x. We show that the filter generated by x is a filter of L.

An utrafilter is a maximal filter, i.e. an ultrafilter of L is a filter of L which is not a proper subset of any other filter of L. We show that the set of all filters of L ordered under inclusion is an inductive order and thus by Zorn's lemma every filter is contained in some ultrafilter. We also show that if the join of x and y is a member of an ultrafilter F then either x is in F or y is in F. This implies that if F is an ultrafilter in a Boolean lattice L and x is any member of L, either x or the complement of x is a member of the ultrafilter F.

```
(DEFTYPE (FILTER-OF (L BOUNDED-LATTICE))
  (LAMBDA ((S (NON-EMPTY-SUBSET-OF (U-SET L))))
    (AND (NOT (IS (BOTTOM L) (MEMBER-OF S)))
         (FORALL ((X (MEMBER-OF S)))
           (IS-EVERY (GREATER-OR-EQUAL-TO X L)
                     (MEMBER-OF S)))
         (FORALL ((X (MEMBER-OF S)))
           (FORALL ((Y (MEMBER-OF S)))
             (IS (MEET X Y L)
                 (MEMBER-OF S)))))))

(DEFTYPE (NON-BOTTOM-MEMBER-OF (L BOUNDED-LATTICE))
  (LAMBDA ((X (IN-U-SET L)))
    (NOT (= X (BOTTOM L)))))
```

```
(LEMMA
  (FORALL ((L BOUNDED-LATTICE))
    (EXISTS-SOME
      (NON-BOTTOM-MEMBER-OF L))))
```

```
(IN-CONTEXT ((LET-BE L BOUNDED-LATTICE)
             (LET-BE T (TOP L)))
  (NOTE
    (EXISTS-SOME
      (NON-BOTTOM-MEMBER-OF L))))
```

```
(DEFTERM (FILTER-GENERATED-BY
           (X (NON-BOTTOM-MEMBER-OF L))
           (L BOUNDED-LATTICE))
  (THE-SET-OF-ALL
    (GREATER-OR-EQUAL-TO X L)))
```

```
(LEMMA
  (FORALL
     ((L BOUNDED-LATTICE)
      (X (NON-BOTTOM-MEMBER-OF L)))
    (IS (FILTER-GENERATED-BY X L)
        (FILTER-OF L))))
```

```
(IN-CONTEXT
    ((LET-BE L BOUNDED-LATTICE)
     (LET-BE X (NON-BOTTOM-MEMBER-OF L))
     (LET-BE F (FILTER-GENERATED-BY X L))
     (PUSH-GOAL (IS F (FILTER-OF L))))

  (IN-CONTEXT ((LET-BE S (U-SET L))
               (LET-BE Y (MEMBER-OF F)))
    (NOTE
      (IS F (NON-EMPTY-SUBSET-OF (U-SET L)))))

  (IN-CONTEXT ((LET-BE BOT (BOTTOM L)))
    (NOTE
      (NOT
        (IS (BOTTOM L) (MEMBER-OF F)))))

  (IN-CONTEXT
      ((LET-BE Y (MEMBER-OF F))
       (LET-BE Z (GREATER-OR-EQUAL-TO Y L)))
    (NOTE (FORALL ((Y (MEMBER-OF F)))
            (IS-EVERY
              (GREATER-OR-EQUAL-TO Y L)
              (MEMBER-OF F)))))

  (IN-CONTEXT ((LET-BE Y (MEMBER-OF F))
               (LET-BE Z (MEMBER-OF F))
               (LET-BE M (MEET X Y L)))
    (NOTE (FORALL ((Y (MEMBER-OF F))
                   (Z (MEMBER-OF F)))
            (IS (MEET Y Z L)
                (MEMBER-OF F)))))

  (NOTE-GOAL))
```

```
(LEMMA
  (FORALL ((L BOUNDED-LATTICE)
           (F (FILTER-OF L)))
    (IS (TOP L)
        (MEMBER-OF F))))
```

```
(IN-CONTEXT
    ((LET-BE L BOUNDED-LATTICE)
     (LET-BE F (FILTER-OF L))
     (PUSH-GOAL
       (IS (TOP L) (MEMBER-OF F))))

  (IN-CONTEXT ((LET-BE X (MEMBER-OF F))
               (LET-BE T (TOP L)))
    (NOTE-GOAL)))
```

```
(LEMMA
  (FORALL ((B BOOLEAN-LATTICE)
           (F (FILTER-OF B))
           (X (MEMBER-OF F)))
    (NOT (IS (COMPLEMENT X B)
             (MEMBER-OF F)))))
```

```
(IN-CONTEXT ((LET-BE B BOOLEAN-LATTICE)
             (LET-BE F (FILTER-OF B))
             (LET-BE X (MEMBER-OF F))
             (LET-BE CX (COMPLEMENT X B)))
  (NOTE (NOT (IS CX (MEMBER-OF B)))))
```

```
(DEFTYPE (ULTRAFILTER-OF (L BOUNDED-LATTICE))
  (MAXIMAL-ELEMENT-OF
    (INCLUSION-ORDER
      (THE-SET-OF-ALL (FILTER-OF L)))))
```

```
(LEMMA
  (FORALL ((L BOUNDED-LATTICE)
           (F (ULTRAFILTER-OF L)))
    (IS F (FILTER-OF L))))
```

```
(IN-CONTEXT ((LET-BE L BOUNDED-LATTICE)
             (PUSH-GOAL
               (IS-EVERY (ULTRAFILTER-OF L)
                         (FILTER-OF L))))
  (IN-CONTEXT
      ((SUPPOSE
         (EXISTS-SOME (ULTRAFILTER-OF L)))
       (LET-BE F (ULTRAFILTER-OF L))
       (LET-BE FILTER-SET
         (THE-SET-OF-ALL (FILTER-OF L)))
       (LET-BE FILTER-POSET
         (INCLUSION-ORDER FILTER-SET)))
    (NOTE-GOAL))
  (NOTE-GOAL))
```

```
(LEMMA
  (FORALL ((L BOUNDED-LATTICE)
           (F (ULTRAFILTER-OF L)))
    (NOT
      (EXISTS-SOME
        (AND-TYPE
          (FILTER-OF L)
          (PROPER-SUPERSET-OF F))))))
```

```
(IN-CONTEXT
    ((LET-BE L BOUNDED-LATTICE)
     (PUSH-GOAL
       (FORALL ((F (ULTRAFILTER-OF L)))
         (NOT
           (EXISTS-SOME
             (AND-TYPE
               (FILTER-OF L)
               (PROPER-SUPERSET-OF F)))))))
  (IN-CONTEXT
      ((SUPPOSE
         (EXISTS-SOME (ULTRAFILTER-OF L)))
       (LET-BE F (ULTRAFILTER-OF L)))
    (IN-CONTEXT
        ((SUPPOSE
           (EXISTS-SOME
               (AND-TYPE
                  (FILTER-OF L)
                  (PROPER-SUPERSET-OF F))))
         (LET-BE F2
           (AND-TYPE (FILTER-OF L)
                     (PROPER-SUPERSET-OF F)))
         (LET-BE FILTER-SET
           (THE-SET-OF-ALL (FILTER-OF L)))
         (LET-BE FILTER-POSET
           (INCLUSION-ORDER FILTER-SET)))
      (NOTE-CONTRADICTION))
    (NOTE-GOAL))
  (NOTE-GOAL))
```

```
(LEMMA
  (FORALL ((L BOUNDED-LATTICE))
    (IS (THE-SET-OF-ALL (FILTER-OF L))
        FAMILY-OF-SETS)))
```

```
(IN-CONTEXT
    ((LET-BE L BOUNDED-LATTICE)
     (LET-BE F
       (THE-SET-OF-ALL (FILTER-OF L)))
     (LET-BE S (MEMBER-OF F)))
  (NOTE (IS F FAMILY-OF-SETS)))
```

We now come to the proof that every filter is contained in some ultrafilter. The following natural argument is taken from [Bell & Machover 77] page 136.

Let F be the set of all filters in a Boolean algebra B; F can be partially ordered by inclusion. We will show that, with respect to this ordering, chains in F have upper bounds in F.

Let Γ be a chain in F, and let $C = \cup\Gamma$. If $x, y \in C$, then for some $D, E \in \Gamma$, $x \in D$ and $y \in E$. Since Γ is a chain, either $D \subseteq E$ or $E \subseteq D$; suppose the latter case obtains. Then $x, y \in D$ and because D is a filter we have $x \wedge y \in D \subseteq C$. If $z \in D$ and $x \leq z$ then *ipso facto* $z \in D \subseteq C$. Since $0 \notin D$ for all $D \in \Gamma$, it follows that $0 \notin C$. Therefore C is a filter and is the required upper bound for Γ in F.

We may accordingly invoke Zorn's Lemma to conclude that, for every filter D in B, F contains a maximal member, i.e. an ultrafilter, which includes D.

A comparison of the above English proof with the Ontic proof given below yields a predicater count loss factor of 1.3 and a word count loss factor of 1.2.

```
(LEMMA
  (FORALL ((L BOUNDED-LATTICE))
    (IS (INCLUSION-ORDER
          (THE-SET-OF-ALL
            (FILTER-OF L)))
        INDUCTIVE-ORDER)))
```

```
(IN-CONTEXT
    ((LET-BE L BOUNDED-LATTICE)
     (LET-BE FILTER-FAMILY
       (THE-SET-OF-ALL (FILTER-OF L)))
     (LET-BE FILTER-POSET
       (INCLUSION-ORDER FILTER-FAMILY))
     (PUSH-GOAL (IS FILTER-POSET INDUCTIVE-ORDER)))

  (IN-CONTEXT ((LET-BE C (CHAIN-IN FILTER-POSET)))

    (IN-CONTEXT ((LET-BE S (MEMBER-OF C)))
      (NOTE (IS C FAMILY-OF-SETS)))

    (IN-CONTEXT
        ((PUSH-GOAL
           (EXISTS-SOME
             (UPPER-BOUND-OF C FILTER-POSET)))
         (LET-BE UC (FAMILY-UNION C)))

      (IN-CONTEXT
          ((PUSH-GOAL (IS UC (FILTER-OF L))))

        (IN-CONTEXT ((LET-BE USET (U-SET L))
                     (LET-BE S (MEMBER-OF C)))
          (NOTE
            (IS UC (NON-EMPTY-SUBSET-OF USET))))

        (IN-CONTEXT
            ((LET-BE BOT (BOTTOM L))
             (SUPPOSE (IS BOT (MEMBER-OF UC)))
             (WRITE-AS BOT (MEMBER-OF S)
               (S (MEMBER-OF C))))
          (NOTE-CONTRADICTION))

        (IN-CONTEXT
            ((PUSH-GOAL
               (FORALL ((X (MEMBER-OF UC)))
                 (IS-EVERY
                   (GREATER-OR-EQUAL-TO X L)
                   (MEMBER-OF UC))))
             (LET-BE X (MEMBER-OF UC))
             (LET-BE Y (GREATER-OR-EQUAL-TO X L)))
          (IN-CONTEXT ((WRITE-AS X (MEMBER-OF S)
                         (S (MEMBER-OF C))))
            (NOTE-GOAL)))
;
;  continued on next page
```

;continued from previous page

```
        (IN-CONTEXT
            ((PUSH-GOAL
               (FORALL ((X (MEMBER-OF UC))
                        (Y (MEMBER-OF UC)))
                 (IS (MEET X Y L) (MEMBER-OF UC))))
             (LET-BE X (MEMBER-OF UC))
             (LET-BE Y (MEMBER-OF UC))
             (LET-BE M (MEET X Y L)))
          (IN-CONTEXT
              ((PUSH-GOAL (IS M (MEMBER-OF UC)))
               (WRITE-AS X (MEMBER-OF S1)
                 (S1 (MEMBER-OF C)))
               (WRITE-AS Y (MEMBER-OF S2)
                 (S2 (MEMBER-OF C))))
            (IN-CONTEXT
                ((SUPPOSE (IS S1 (SUBSET-OF S2))))
              (NOTE-GOAL))
            (NOTE-GOAL))
          (NOTE-GOAL))

        (NOTE-GOAL))

      (IN-CONTEXT ((LET-BE S (MEMBER-OF C)))
        (NOTE
          (IS UC
            (UPPER-BOUND-OF C FILTER-POSET))))
      (NOTE-GOAL))

    (NOTE-GOAL)))
```

```
(LEMMA
  (FORALL ((L BOUNDED-LATTICE)
           (F (FILTER-OF L)))
    (EXISTS-SOME
      (AND-TYPE
        (ULTRAFILTER-OF L)
        (SUPERSET-OF F)))))
```

```
(IN-CONTEXT
    ((LET-BE L BOUNDED-LATTICE)
     (LET-BE F (FILTER-OF L))
     (PUSH-GOAL
       (EXISTS-SOME
         (AND-TYPE (ULTRAFILTER-OF L)
                   (SUPERSET-OF F)))))
  (IN-CONTEXT
    ((LET-BE FILTER-SET
        (THE-SET-OF-ALL (FILTER-OF L)))
     (LET-BE FILTER-POSET
        (INCLUSION-ORDER FILTER-SET))
     (LET-BE F2
       (AND-TYPE
         (MAXIMAL-ELEMENT-OF FILTER-POSET)
         (GREATER-OR-EQUAL-TO F FILTER-POSET))))
    (NOTE-GOAL)))
```

```
(DEFTYPE (ULTRAFILTER-CONTAINING
           (X (IN-U-SET L))
           (L BOUNDED-LATTICE))
  (LAMBDA ((F (ULTRAFILTER-OF L)))
    (IS X (MEMBER-OF F))))
```

```
(LEMMA
  (FORALL
      ((L BOUNDED-LATTICE)
       (X (NON-BOTTOM-MEMBER-OF L)))
    (EXISTS-SOME
      (ULTRAFILTER-CONTAINING X L))))
```

```
(IN-CONTEXT
    ((LET-BE L BOUNDED-LATTICE)
     (LET-BE X (NON-BOTTOM-MEMBER-OF L))
     (PUSH-GOAL
       (EXISTS-SOME
         (ULTRAFILTER-CONTAINING X L))))
  (IN-CONTEXT
      ((LET-BE G1 (FILTER-GENERATED-BY X L))
       (LET-BE G2 (AND-TYPE (ULTRAFILTER-OF L)
                            (SUPERSET-OF G1))))
    (NOTE-GOAL)))
```

```
(LEMMA
 (FORALL ((B BOOLEAN-LATTICE)
          (X (IN-U-SET B))
          (Y (IN-U-SET B)))
   (=>
     (NOT (IS X (LESS-OR-EQUAL-TO Y B)))
     (EXISTS-SOME
         ((F (ULTRAFILTER-CONTAINING X
                                      B)))
       (NOT (IS Y (MEMBER-OF F)))))))
```

```
(IN-CONTEXT
   ((LET-BE B BOOLEAN-LATTICE)
    (LET-BE X (IN-U-SET B))
    (LET-BE Y (IN-U-SET B))
    (SUPPOSE
      (NOT (IS X (LESS-OR-EQUAL-TO Y B))))
    (PUSH-GOAL
     (EXISTS ((F (ULTRAFILTER-CONTAINING X B)))
        (NOT (IS Y (MEMBER-OF F))))))
  (IN-CONTEXT
      ((LET-BE CY (COMPLEMENT Y B))
       (LET-BE M (MEET X CY B))
       (LET-BE F (ULTRAFILTER-CONTAINING M B)))
    (NOTE-GOAL)))
```

We now come to the proof that if F is an ultrafilter and if $x \vee y \in F$ then $x \in F$ or $y \in F$. The following natural argument is taken from [Bell & Machover 77] top of page 136, case (iii)⇒(iv).

Suppose F is an ultrafilter of a bounded distributive lattice L and that $x \vee y \in F$. To show that $x \in F$ or $y \in F$ suppose that $x \notin F$. It is easy to see that $\{z : x \vee z \in F\}$ is a filter which includes F, and so, since F is an ultrafilter, $F = G$. But since $x \vee y \in F$ it follows that $y \in G$ and hence $y \in F$.

A comparison of the above natural argument with the Ontic proof yields a predicate count loss factor of 2.1 and a word count loss factor of 2.7.

```
(LEMMA
  (FORALL
      ((L (AND-TYPE
            DISTRIBUTIVE-LATTICE
            BOUNDED-LATTICE))
       (F (ULTRAFILTER-OF L))
       (X (IN-U-SET L))
       (Y (IN-U-SET L)))
    (=> (IS (JOIN X Y L)
            (MEMBER-OF F))
        (OR (IS X (MEMBER-OF F))
            (IS Y (MEMBER-OF F))))))
```

```
(IN-CONTEXT
    ((LET-BE L (AND-TYPE
                 DISTRIBUTIVE-LATTICE
                 BOUNDED-LATTICE))
     (LET-BE F (ULTRAFILTER-OF L))
     (LET-BE X (IN-U-SET L))
     (LET-BE Y (IN-U-SET L))
     (SUPPOSE (IS (JOIN X Y L)
                  (MEMBER-OF F)))
     (PUSH-GOAL (OR (IS X (MEMBER-OF F))
                    (IS Y (MEMBER-OF F)))))

  (IN-CONTEXT
      ((SUPPOSE (NOT (IS X (MEMBER-OF F))))
       (PUSH-GOAL (IS Y (MEMBER-OF F))))

    (IN-CONTEXT
        ((LET-BE G
            (THE-SET-OF-ALL (Z (IN-U-SET L))
              (IS (JOIN X Z L) (MEMBER-OF F)))))
      ;clearly y is in g
      (IN-CONTEXT ((PUSH-GOAL (= F G)))
        ;this will complete the proof that
        ;y is in f

        (IN-CONTEXT
            ((PUSH-GOAL (IS G (SUPERSET-OF F)))
             (LET-BE Z (MEMBER-OF F))
             (LET-BE J (JOIN X Z L)))
          (NOTE-GOAL))

        (IN-CONTEXT
            ((PUSH-GOAL (IS G (FILTER-OF L))))
          ;since f is a maximal filter this
          ;completes the proof
          (IN-CONTEXT
              ((PUSH-GOAL
                 (IS G
                   (NON-EMPTY-SUBSET-OF
                     (U-SET L))))
               (LET-BE S (U-SET L))
               (LET-BE Z (MEMBER-OF G)))
            (NOTE-GOAL))

          (IN-CONTEXT ((LET-BE BOT (BOTTOM L)))
            (NOTE
              (NOT (IS (BOTTOM L)
                       (MEMBER-OF G)))))
;
;   continued on next page
```

;continued from previous page

```
          (IN-CONTEXT
              ((PUSH-GOAL
                 (FORALL
                     ((Z1 (MEMBER-OF G))
                      (Z2 (GREATER-OR-EQUAL-TO Z1
                             L)))
                   (IS Z2 (MEMBER-OF G))))
               (LET-BE Z1 (MEMBER-OF G))
               (LET-BE Z2
                 (GREATER-OR-EQUAL-TO Z1 L))
               (LET-BE J1 (JOIN X Z1 L))
               (LET-BE J2 (JOIN X Z2 L)))
             ;j2 is greater or equal to j1
            (NOTE-GOAL))

          (IN-CONTEXT
              ((PUSH-GOAL
                 (FORALL ((Z1 (MEMBER-OF G))
                          (Z2 (MEMBER-OF G)))
                   (IS (MEET Z1 Z2 L)
                       (MEMBER-OF G))))
               (LET-BE Z1 (MEMBER-OF G))
               (LET-BE Z2 (MEMBER-OF G)))
             (IN-CONTEXT
                 ((LET-BE J1 (JOIN X Z1 L))
                  (LET-BE J2 (JOIN X Z2 L)))
               (NOTE (IS (JOIN X (MEET Z1 Z2 L) L)
                         (MEMBER-OF F))))
             (IN-CONTEXT
                 ((LET-BE M (MEET Z1 Z2 L)))
               (NOTE-GOAL)))

          (NOTE-GOAL))
        (NOTE-GOAL))
      (NOTE-GOAL)))
  (NOTE-GOAL))
```

```
(LEMMA
  (FORALL ((B BOOLEAN-LATTICE)
           (F (ULTRAFILTER-OF B))
           (X (IN-U-SET B)))
    (OR (IS X (MEMBER-OF F))
        (IS (COMPLEMENT X B)
            (MEMBER-OF F)))))
```

```
(IN-CONTEXT ((LET-BE B BOOLEAN-LATTICE)
             (LET-BE F (ULTRAFILTER-OF B))
             (LET-BE X (IN-U-SET B))
             (LET-BE CX (COMPLEMENT X B)))
  (NOTE (OR (IS X (MEMBER-OF F))
            (IS CX (MEMBER-OF F)))))
```

A.11 The Stone Representation Theorem

Finally we come to the Stone representation theorem for Boolean algebras. The following natural definitions and natural arguments are taken from [Bell & Machover 77] pages 141 and 142.

> Let us define a *field of sets* to be a subalgebra of a power set algebra. In particular, a *field of subsets* of a set X is a subalgebra of the power set of X.
>
> If B is a Boolean algebra, we denote by SB the set of all ultrafilters in B.
>
> Theorem. Each Boolean algebra is isomorphic to a field of subsets of SB.
>
> Proof. Let B be a Boolean algebra. Define a mapping $u : B \to PSB$ by putting:
>
> $$u(x) = \{F \in SB : x \in F\}$$
>
> for each $x \in B$. Thus $u(x)$ is the set of all ultrafilters containing x.
>
> We claim that u is a homomorphism of B into PSB. For suppose that $x, y \in B$; then, if $F \in SB$, we have
>
> $$F \in u(x \wedge y) \Leftrightarrow x \wedge y \in F \Leftrightarrow x \in F \& y \in F \Leftrightarrow F \in u(x) \cap u(y)$$
>
> Hence $u(x \wedge y) = u(x) \cap u(y)$. Also, we have
>
> $$F \in u(x*) \Leftrightarrow x* \in F \Leftrightarrow x \notin F (\text{by Thm. 3.5(iv)}) \Leftrightarrow F \in SB - u(x)$$
>
> Accordingly $u(x*) = SB - u(x)$, so that, by Prob 3.3, u is a homomorphism.
>
> We also note that u is one-one, for if $x \neq y$ then by Cor. 3.9 there is an ultrafilter F containing x, say, but not y. Then $F \in u(x)$ and $F \notin u(y)$, so that $u(x) \neq u(y)$.
>
> We have therefore shown that u is an isomorphism of B onto the subalgebra $u[B]$ of PSB, which proves the theorem.

A comparison of the above natural definitions and arguments with the remainder of this section yields a predicate count loss factor of 2.0 and a word count loss factor of 1.7.

```
                (DEFTYPE FIELD-OF-SETS
                  (WRITABLE-AS (BOOLEAN-SUBALGEBRA-OF
                                 (POWER-SET-LATTICE S))
                     (S SET)))

(LEMMA                                   (IN-CONTEXT ((LET-BE S SET)
  (EXISTS-SOME FIELD-OF-SETS))                        (LET-BE P (POWER-SET-LATTICE S)))
                                           (NOTE (EXISTS-SOME FIELD-OF-SETS)))

(LEMMA                                   (IN-CONTEXT
  (FORALL ((B FIELD-OF-SETS))                ((LET-BE B FIELD-OF-SETS)
    (IS B BOOLEAN-LATTICE)))                  (WRITE-AS B (BOOLEAN-SUBALGEBRA-OF
                                                            (POWER-SET-LATTICE S))
                                                (S SET))
                                              (LET-BE B2 (POWER-SET-LATTICE S)))
                                           (NOTE (IS B BOOLEAN-LATTICE)))

                (DEFTERM (ALL-STONE-MODELS (B BOOLEAN-LATTICE))
                  (THE-SET-OF-ALL (ULTRAFILTER-OF B)))

                (DEFTERM (THE-STONE-MODELS-OF
                           (X (IN-U-SET B))
                           (B BOOLEAN-LATTICE))
                  (THE-SET-OF-ALL
                    (ULTRAFILTER-CONTAINING X B)))

(LEMMA                                   (IN-CONTEXT
 (FORALL ((B BOOLEAN-LATTICE)                ((LET-BE B BOOLEAN-LATTICE)
          (X (IN-U-SET B)))                   (LET-BE S (ALL-STONE-MODELS B))
   (IS (THE-STONE-MODELS-OF X B)              (LET-BE X (IN-U-SET B))
       (SUBSET-OF                             (LET-BE SX (THE-STONE-MODELS-OF X B))
         (ALL-STONE-MODELS B)))))             (PUSH-GOAL (IS SX (SUBSET-OF S))))
                                           (IN-CONTEXT
                                               ((SUPPOSE
                                                   (EXISTS-SOME (MEMBER-OF SX)))
                                                (LET-BE F (MEMBER-OF SX)))
                                             (NOTE-GOAL))
                                           (NOTE-GOAL))

                (DEFTERM (STONE-MAP (B BOOLEAN-LATTICE))
                  (MAKE-MAP
                    B
                    (POWER-SET-LATTICE
                      (ALL-STONE-MODELS B))
                    (THE-RULE ((X (IN-U-SET B)))
                      (THE-STONE-MODELS-OF X B))))
```

```
(LEMMA
  (FORALL ((B BOOLEAN-LATTICE))
    (IS (POWER-SET-LATTICE
          (ALL-STONE-MODELS B))
        POWER-LATTICE)))

(LEMMA
  (FORALL ((B BOOLEAN-LATTICE))
    (= (U-SET (POWER-SET-LATTICE
                (ALL-STONE-MODELS B)))
       (POWER-SET
         (ALL-STONE-MODELS B)))))

(LEMMA
  (FORALL
      ((B BOOLEAN-LATTICE)
       (S2 (SUBSET-OF
             (ALL-STONE-MODELS B))))
    (IS S2
      (MEMBER-OF
        (U-SET
          (POWER-SET-LATTICE
            (ALL-STONE-MODELS B)))))))
```

```
(IN-CONTEXT ((LET-BE B BOOLEAN-LATTICE)
             (LET-BE S (ALL-STONE-MODELS B)))
  (NOTE (IS (POWER-SET-LATTICE S) POWER-LATTICE))
  (IN-CONTEXT ((LET-BE PS (POWER-SET S)))
    (NOTE (= (U-SET (POWER-SET-LATTICE S)) PS))
    (NOTE
      (IS-EVERY
        (SUBSET-OF S)
        (MEMBER-OF
          (U-SET
            (POWER-SET-LATTICE S)))))))
```

```
(LEMMA
  (FORALL ((B BOOLEAN-LATTICE))
    (IS (THE-RULE ((X (IN-U-SET B)))
          (THE-STONE-MODELS-OF X B))
        (RULE-BETWEEN
          (U-SET B)
          (U-SET
            (POWER-SET-LATTICE
              (ALL-STONE-MODELS B)))))))

(LEMMA
  (FORALL ((B BOOLEAN-LATTICE))
    (IS (STONE-MAP B)
        (MAP-BETWEEN
          B
          (POWER-SET-LATTICE
            (ALL-STONE-MODELS B))))))

(LEMMA
  (FORALL ((B BOOLEAN-LATTICE))
    (IS (STONE-MAP B) BOOLEAN-MAP)))

(LEMMA
  (FORALL ((B BOOLEAN-LATTICE))
    (= (DOMAIN (STONE-MAP B))
       B)))

(LEMMA
  (FORALL ((B BOOLEAN-LATTICE))
    (= (RANGE (STONE-MAP B))
       (POWER-SET-LATTICE
         (ALL-STONE-MODELS B)))))

(LEMMA
  (FORALL ((B BOOLEAN-LATTICE)
           (X (IN-U-SET B)))
    (= (APPLY-MAP (STONE-MAP B) X)
       (THE-STONE-MODELS-OF X B))))
```

```
(IN-CONTEXT
    ((LET-BE B BOOLEAN-LATTICE)
     (LET-BE SB
       (POWER-SET-LATTICE (ALL-STONE-MODELS B)))
     (LET-BE H (STONE-MAP B))
     (LET-BE R (THE-RULE ((X (IN-U-SET B)))
                 (THE-STONE-MODELS-OF X B)))
     (LET-BE X (IN-U-SET B)))
  (IN-CONTEXT ((LET-BE HX (APPLY-RULE R X))
               (LET-BE USET1 (U-SET B))
               (LET-BE USET2 (U-SET SB)))
    (NOTE (IS R (RULE-BETWEEN USET1 USET2))))
  (NOTE (IS H (MAP-BETWEEN B SB)))
  (NOTE (IS H BOOLEAN-MAP))
  (NOTE (= (DOMAIN H) B))
  (NOTE (= (RANGE H) SB))
  (NOTE (= (APPLY-MAP H X)
           (THE-STONE-MODELS-OF X B))))
```

```
(LEMMA
  (FORALL ((B BOOLEAN-LATTICE))
    (IS (STONE-MAP B)
      BOOLEAN-HOMOMORPHISM)))
```

```
(IN-CONTEXT
    ((LET-BE B BOOLEAN-LATTICE)
     (LET-BE H (STONE-MAP B))
     (LET-BE SB
       (POWER-SET-LATTICE
         (ALL-STONE-MODELS B)))
     (PUSH-GOAL
       (IS H BOOLEAN-HOMOMORPHISM)))

  (IN-CONTEXT
      ((PUSH-GOAL
         (IS H MAP-WHICH-RESPECTS-MEET))
       (LET-BE X (IN-U-SET B))
       (LET-BE Y (IN-U-SET B))
       (LET-BE X-MODELS (APPLY-MAP H X))
       (LET-BE Y-MODELS (APPLY-MAP H Y))
       (LET-BE M (MEET X Y B))
       (LET-BE M-MODELS (APPLY-MAP H M))
       (LET-BE MODEL-INTERSECTION
         (INTERSECTION X-MODELS Y-MODELS)))

    (IN-CONTEXT
        ((PUSH-GOAL
           (= M-MODELS MODEL-INTERSECTION)))

      (IN-CONTEXT
          ((PUSH-GOAL
             (IS MODEL-INTERSECTION
               (SUBSET-OF M-MODELS))))
        (IN-CONTEXT
            ((SUPPOSE
               (EXISTS-SOME
                 (MEMBER-OF MODEL-INTERSECTION)))
             (LET-BE F
               (MEMBER-OF MODEL-INTERSECTION)))
          (NOTE-GOAL))
        (NOTE-GOAL))

      (IN-CONTEXT
          ((PUSH-GOAL
             (IS M-MODELS
                 (SUBSET-OF MODEL-INTERSECTION))))
        (IN-CONTEXT
            ((SUPPOSE
               (EXISTS-SOME
                 (MEMBER-OF M-MODELS)))
             (LET-BE F (MEMBER-OF M-MODELS)))
          (NOTE-GOAL))
        (NOTE-GOAL))

      (NOTE-GOAL))
    (NOTE-GOAL))
;
;    continued on next page
```

;continued from previous page

```
(IN-CONTEXT
    ((PUSH-GOAL
       (IS H MAP-WHICH-RESPECTS-COMPLEMENT))
     (LET-BE X (IN-U-SET B))
     (LET-BE HX (APPLY-MAP H X))
     (LET-BE C (COMPLEMENT-OF X B))
     (LET-BE C-MODELS (APPLY-MAP H C))
     (LET-BE ALL-MODELS (ALL-STONE-MODELS B))
     (LET-BE MODEL-COMPLEMENT
             (SET-DIFFERENCE ALL-MODELS HX)))

  (IN-CONTEXT
      ((PUSH-GOAL (= C-MODELS MODEL-COMPLEMENT)))

    (IN-CONTEXT
        ((PUSH-GOAL
           (IS MODEL-COMPLEMENT
               (SUBSET-OF C-MODELS))))
      (IN-CONTEXT
          ((SUPPOSE
             (EXISTS-SOME
               (MEMBER-OF MODEL-COMPLEMENT)))
           (LET-BE F
             (MEMBER-OF MODEL-COMPLEMENT)))
        (NOTE-GOAL))
      (NOTE-GOAL))

    (IN-CONTEXT
        ((PUSH-GOAL
           (IS C-MODELS
               (SUBSET-OF MODEL-COMPLEMENT))))
      (IN-CONTEXT
          ((SUPPOSE
             (EXISTS-SOME
               (MEMBER-OF C-MODELS)))
           (LET-BE F (MEMBER-OF C-MODELS)))
        (NOTE-GOAL))
      (NOTE-GOAL))

    (NOTE-GOAL))
  (NOTE-GOAL))

(NOTE-GOAL))
```

```
(LEMMA
  (FORALL ((B BOOLEAN-LATTICE))
    (IS (STONE-MAP B) INJECTION)))
```

```
(IN-CONTEXT ((LET-BE B BOOLEAN-LATTICE)
             (LET-BE H (STONE-MAP B))
             (PUSH-GOAL (IS H INJECTION)))
  (IN-CONTEXT
      ((LET-BE MSET (MEMBER-OF (IMAGE H)))
       (LET-BE PRE-MSET
         (PREIMAGE H (MAKE-SET MSET))))
    (IN-CONTEXT
        ((PUSH-GOAL
           (EXACTLY-ONE (MEMBER-OF PRE-MSET)))
         (LET-BE X (MEMBER-OF PRE-MSET))
         (LET-BE Y (MEMBER-OF PRE-MSET)))
      (IN-CONTEXT
          ((PUSH-GOAL
             (IS X (LESS-OR-EQUAL-TO Y B))))
        (IN-CONTEXT
            ((SUPPOSE
               (NOT (IS X (LESS-OR-EQUAL-TO Y B))))
             (LET-BE F (ULTRAFILTER-CONTAINING X B)
               (NOT (IS Y (MEMBER-OF F)))))
          (NOTE-CONTRADICTION))
        (NOTE+GENERALIZE-GOAL))
      (NOTE-GOAL))
    (NOTE-GOAL)))
```

```
(LEMMA
  (FORALL ((B BOOLEAN-LATTICE))
    (IS (BOOLEAN-IMAGE (STONE-MAP B))
        FIELD-OF-SETS)))
```

```
(LEMMA
  (FORALL ((B BOOLEAN-LATTICE))
    (IS (SET!-RANGE
          (STONE-MAP B)
          (BOOLEAN-IMAGE (STONE-MAP B)))
      (BOOLEAN-ISOMORPHISM-BETWEEN
        B
        (BOOLEAN-IMAGE
          (STONE-MAP B))))))
```

```
(LEMMA
 (FORALL ((B BOOLEAN-LATTICE))
  (EXISTS-SOME
   (AND-TYPE
    FIELD-OF-SETS
    (BOOLEAN-LATTICE-ISOMORPHIC-TO
      B)))))
```

```
(IN-CONTEXT ((LET-BE B BOOLEAN-LATTICE)
             (LET-BE H (STONE-MAP B))
             (LET-BE B2 (BOOLEAN-IMAGE H)))
  (IN-CONTEXT ((LET-BE S (ALL-STONE-MODELS B)))
    (NOTE (IS B2 FIELD-OF-SETS)))
  (IN-CONTEXT ((LET-BE H2 (SET!-RANGE H B2)))
    (NOTE
      (IS H2
        (BOOLEAN-ISOMORPHISM-BETWEEN B B2)))
    (NOTE
      (EXISTS-SOME
        (AND-TYPE
          FIELD-OF-SETS
          (BOOLEAN-LATTICE-ISOMORPHIC-TO B))))))
```

Bibliography

[Ait-Kaci & Nasr 86] Hassan Ait-Kaci, Roger Nasr, Logic and Inheretance, Thirteenth Annual Symposium on Principles of Programming Languages, January 1986, pp. 219-228.

[Andrews 81] Peter Andrews, Theorem Proving via General Matings, JACM, Vol 28, no. 2, April 1981, pp. 193-214.

[Bell & Machover 77] John Bell and Moshe Machover, A Course in Mathematical Logic, North-Holland, 1977.

[Bibel 79] W. Bibel, Tautology Testing with a Generalized Matrix Reduction Method, Theoretical Computer Science 8, 1979, pp. 31-44.

[Bibel 81] W. Bibel, On Matrices with Connections, JACM vol. 28, No. 4, October 1981, pp 633-645.

[Ballantyne & Bledsoe 77] A. M. Ballantyne and W. W. Bledsoe, Automatic Proofs of Theorems in Analysis Using Nonstandard Techniques, JACM vol. 24 no. 3, July 1977, pp. 353-374.

[Bledsoe et al. 72] W. W. Bledsoe, R. S. Boyer, W. H. Henneman, Computer Proofs of Limit Theorems, Artificial Intelligence 3, 1972, pp. 27-60.

[Bledsoe & Bruell 73] W. W. Bledsoe, Peter Bruel, A Man-Machine Theorem Proving System, Proc. of the 3rd IJCAI, 1973, pp. 56-65.

[Bledsoe 77] W. W. Bledsoe, Non-resolution theorem Proving, Artificial Intelligence 9, 1977, pp. 1-35.

[Boyer & Moore 79] Robert S. Boyer, J. Struther Moore, A Computational Logic, ACM Monograph Series, 1979.

[Boyer & Moore 84] Robert S. Boyer, J. Struther Moore, A Mechanical Proof of the Unsolvability of the Halting Problem, Journal of the Associateion for Computing Machinery, Vol. 31, No. 3, July 1984, pp. 441-485.

[Boyer & Moore 86] Robert S. Boyer, J. Struther Moore, Overview of A Theorem-Prover for A Computational Logic, 8th International Conference on Autoated Deduction, Lecture Notes in Computer Science, Springer-Verlag 1986, pp. 675-678.

[Bundy 73] Alan Bundy, Doing Arithmetic with Diagrams, Proc. of the 3rd IJCAI, 1973, pp. 130-138.

[Brachman 79] Ronald J. Brachman On the Epistemological Status of Semantic Networks, in Readings in Knowledge Representation, R. Brachman, H. Levesque eds., Morgan Kaufmann Publishers, 1985.

[Brachman, Fikes, & Levesque 82] R. Brachman, R. Fikes, H. Levesque, Krypton: A Functional Approach to Knowledge Representation, IEEE Computer 16, 1983, pp. 63-73

[Brachman & Schmolze 85] R. J. Brachman, J. Schmolze, An Overview of the KL-ONE Knowledge Representation System, Cognitive Science 9 (2), 1985, pp. 171-216.

[Cardelli 84] Luca Cardelli, The Semantics of Multiple Inheritance, Procedings of the Conference on the Semantics of Datatypes, Springer-Verlag Lecture Notes in Computer Sciece, June 1984, pp. 51-66.

[Chang & Lee 73] C. Chang and R. C. Lee, Symbolic Logic and Mechanical Theorem Proving, Academic Press, New York, 1973.

[Chou 84] Shang-ching Chou, Proving Elementary Geometry Theorems Using Wu's Algorithm, in Automated Theorem Proving after 25 Years, W. W. Bledsoe and D. Loveland eds., AMS Contemporary Mathematics Series 29 (1984), 243-286.

[Chou 85] Shang-ching Chou, Proving and Discovering Geometry Theorems using Wu's Method. PhD thesis, Department of Mathematics, University of Texas, Austin (1985).

[Chou & Schelter 86] Shang-ching Chou, William, F. Schelter, Proving Geometry Theorems with Rewrite Rules, Journal of Automated Reasoning 2, 1986, pp. 253-273.

[Constable et al. 82] R. L. Constable, S. D. Johnson, C. D. Eichenlaub, An Introduction to the PL/CV2 Programming Logic, Lecture Notes in Computer Science 135, Springer-Verlag, 1982

[Constable et al. 85] R. L. Constable, T.B Knoblock, J. L. Bates, Writing Programs that Construct Proofs. Journal of Automated Reasoning 1 (1985) pp. 285-326.

[Constable 85] Robert Constable, Constructive Mathematics as a programming Logic I: Some Principles of Theory, Annals of Discrete Mathematics 24, North Holland, 1985, pp. 21-38

[Constable et al. 86] R. L. Constable, S. F. Allen, H. M. Bromely, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. K. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, S. F. Smith, Implementing Mathematics with the Nuprl Development System, Prentice Hall, 1986.

[Davis 81] Martin Davis, Obvious Logical Inferences, Proc. of IJCAI-81, Vancouver, BC, August 1981, pp. 530-531.

[deBruijn 68] N. G. de Bruijn, The Mathematical Language Automath, its use and some of its extensions. Symposium on Automatic Demonstration (Versailles, December 1968), Lecture Notes in Mathematics, Vol 125, pp. 29-61, Springer-Verlag, Berlin, 1970.

[Dershowitz 79] Nachum Dershowitz, Orderings for Term Rewriting Systems, Proc. of the 20th Symposium on the Foundations of Computer Science, 1979, pp. 123-131.

[deBruijn 73] The AUTOMATH Checking Project. Procedings of the Symposium on APL (Paris, December 1973), ed. P. Braffort.

[deKleer et al. 77] J. de Kleer, J Doyle, G. Steele, G. Sussman, Explicit Control of Reasoning, MIT AI Lab. Memo 427, June 1977.

[Downey, Sethi & Tarjan 80] Peter J. Downey, Ravi Sethi, Robert E. Tarjan, Variations on the Common Subexpression Problem, JACM 27, No. 4, October 1980, pp. 758-771.

[Etherington & Reiter 83] David W. Etherington, Raymond Reiter, On Inheritance Hierarchies With Exceptions, AAAI-83, pp. 104-108.

[Ernst 73] G. W. Ernst, A Definition Driven Theorem Prover, Proc. of the 3rd IJCAI, 1973, pp. 51-55.

[Fahlman 79] Scott E. Fahlman, NETL: A System for Representing Real World Knowledge, MIT Press, Cambridge Mass, 1979.

[Gelernter 59] H. Gelernter, Realization of a Geometry theorem Proving Machine, in Automation of Reasoning 1, J. Siekmann and G Writson (eds.) Springer-Verlag 1983.

[Goldstein 73] I. Goldstein, Elementary Geometry theorem Proving, MIT-AI Lab Memo 280, (April 1973).

[Gordon, Milner & Wadsworth 79] M. Gordon, R. Milner, C. Wadsworth, Edinburgh LCF, Lecture Notes in Computer Science 78, Springer-Verlag 1980.

[Gratzer 78] George Gratzer, General Lattice Theory, Academic Press, 1978.

[Guttag & Horning 78] J. V. Guttag, J. J. Horning, The Algebraic Specification of Abstract Data Types, Acta Informatica 10, no. 1, 1978, pp. 1-26.

[Harper 85] Robert Harper Aspects of the Implementation of Type Theory, Ph.D dissertation, Department of Computer Science Cornell University, 1985.

[Hayes 85] Patrick Hayes, The Second Naive Physics Manifesto, in Formal Theories of the Commonsense World, J. Hobbs and R. Moore eds., Ablex Publishers, 1985.

[Howe 86] Douglas J. Howe, Implementing Number Theory: An Experiment with Nuprl, in 8th Internaltion Conference on Automated Deduction, Lecture Notes in Computer Science, Springer-Verlag, July 1986, pp. 404-415.

[Huet 75] G. Huet, A Unification Algorithm for Typed λ-Calculus, Theoretical Computer Science 1, 27-57, 1975.

[Huet & Hullot 83] Gerard Huet, Jean-Marie Hullot, Proofs by Induction in Equational Theories with Constructors, JCSS 25, 1982, pp. 239-366.

[Huet 86] Gerard Huet, Theorem Proving Systems of the Formel Project, Proc. of the 8th International Conference on Automated Deduction, Lecture Notes in Computer Science, Springer-Verlag, 1986, pp. 687-688.

[Ingalls 76] Daniel H. Ingalls, The Smalltalk-76 Programming System: Design and Implementation, 5th Annual ACM Symposium on Principles of Programming Languages, Jan. 1978, pp. 9-15.

[Jutting 79] Checking Landau's "Grundlagen" in the AUTOMATH system. Mathematical Centre Tracts 83, Mathematisch Centrum, Amsterdam 1979.

[Kapur et al. 86] D. Kapur, G. Sivakumar, H. Zhang, RRL: A Rewrite Rule Laboratory, Proc. of the 8th International Conference on Automated Deduction, Lecture Notes in Computer Science, Springer-Verlag, 1986, pp. 691-692.

[Ketonen 84] Jussi Ketonen, EKL - A Mathematically Oriented Proof Checker, Procedings of the 7th International Conference on Automated Deduction, Lecture Note in Computer Science, 1984, pp. 65-79.

[Knuth & Bendix 69] Donald E. Knuth, Peter B. Bendix, Simple Word Problems in Universal Algebras, in Computational Problems in Abstract Algebra, J. Leech (ed.), Pergamon Press, 1969.

[Kozen 77] Dexter C. Kozen, Complexity of Finitely Presented Algebras, Doctoral Dissertation, Computer Science Department, Cornell University, 1977.

[Lescanne 86] Pierre Lescanne, REVE a Rewrite Rule Laboratory, Proc. of the 8th International Conference on Automated Deduction, Lecture Notes in Computer Science, Springer-Verlag, 1986, pp. 695-696.

[Levesque & Brachman 85] Hector J. Levesque, Ronald J. Brachman, A fundamental Tradeoff in Knowledge Representation and Reasoning, in Readings in Knowledge Representation, R. J. Brachman, H. J. Levesque (Eds.), Morgan Kaufmann Publishers, 1985.

[Loveland 78] Donald Loveland, Automated Theorem Proving: A Logical Basis, North-Holland 1978.

[Lusk McCune & Overbeek 82] E. L. Lusk, W. McCune, R. A. Overbeek, Logic Machine Architecture: Kernel Functions, Proc. of the 6th International Conference on Automated Deduction, Lecture Notes in Computer Science 138 (Ed. D.W. Loveland) Sprnger-Verlag 1982, pp. 70-84.

[Lusk & Overbeek 84] E. L. Lusk, R. A. Overbeek, A Portable Environment for Research in Automated Reasoning, Proc. of the 7th International Conference on Automated Deduction, Lecture Notes In Computer Science, Springer-Verlag, 1984.

[Mackworth 77] A. K. Mackworth, Consistency in Networks of Relations, Artificial Intelligence 8, 1977, pp. 99-118.

[McAllester 83] David McAllester, Symmetric Set Theory, A General Theory of Isomorphism, Abstraction, and Representation, MIT AI Lab. Memo no. 710, August 1983.

[McDonald & Suppes 84] J. McDonald, P. Suppes, Student Use of an Interactive Theorem Prover, in Automated Theorem Proving After 25 Years (W. W. Bledsoe, D. W. Loveland Eds.), Vol 29 of Contemporary Mathematics, AMS, Providence R. I. 1984.

[Miller et al. 82] D. A. Miller, E. L. Cohen, P. B. Andrews, A look at TPS, Proc. of the 6th International Conference on Automated Deduction, Lecture Notes in Computer Science 138, Springer-Verlag, 1982, pp. 50-68.

[Murray 82] N. V. Murray, Completely Non-Clausal Theorem Proving, Artificial Intelligence 18, 1982, pp. 67-85.

[Newell, Shaw & Simon 57] A. Newell, J. C. Shaw, H. A. Simon, Empirical Explorations with the Logic Theory Machine: a Case Study in Heuristics, in Automation of Reasoning 1, J. Siekmann and G Writson (eds.) Springer-Verlag 1983.

[Nelson & Oppen 79] Greg Nelson, Derek Oppen, Simplification by Cooperating Decision Procedures, ACM Trans. Prog. Lang. Syst. 1,2 Oct. 1979, pp. 245-257.

[Nelson & Oppen 80] Greg Nellson, Derek Oppen, Fast Decision Procedures based on Congruence Closure, JACM 27, No. 2, April 1980, pp 356-364.

[Nevins 74] A. J. Nevins, A human Oriented Logic for Automatic Theorem Proving, J. ACM 21, 1974, pp. 606-621.

[Nevins 75] A. J. Nevins, Plane Geometry Theorem Proving using Forward Chaining, Artificial Intelligence 6, 1975, pp. 1-23.

[O'Donnell 85] Michael J. O'Donnell, Equational Logic as a Programming Language, MIT Press, 1985.

[Reiter 73] R. Reiter, A Semantically Guided Deductive System for Automatic Theorem Proving, Proc. of the 3rd IJCAI, 1973, pp. 41-46.

[Robinson 65] J. A. Robinson, A Machine Oriented Logic based on the Resolution Principle, JACM 12, no. 1, 1965 pp. 23-41.

[Russinoff 85] David M. Russinoff, An Experiment with the Boyer-Moore Theorem Prover: A Proof of Wilson's Theorem, Journal of Automated Reasoning 1, 1985, pp. 121-139.

[Siekmann & Wrightson 83] J. Siekmann, G. Wrightson ed., Automation of Reasoning: Classical Papers on Computational Logic, Springer-Verlag 1978 (in two volumes).

[Siekmann 84] Jorg H. Siekmann, Universal Unification, Proc. of the 7th International Conference on Automated Deduction, Lecture Notes In Computer Science, Springer-Verlag, 1984.

[Shankar 85] N. Shankar, Towards Mechanical Metamathematics, Journal of Automated Reasoning 1, 1985, pp. 407-434.

[Shostak 82] Robert E. Shostak, Deciding Combinations of Theories, 6th International Conference on Automated Deduction, Lecture Notes in Computer Science, Springer-Verlag, 1982, pp. 1-12.

[Slagel 74] James Slagel, Automated Theorem-Proving for Theories with Simplifiers, Commutativity, and Associativity, JACM 21, No. 4, October 1974, pp. 622-642.

[Stallman & Sussman 77] Richard M. Stallman and Gerald J. Sussman, Forward Reasoning and Dependency-Directed Backtracking in a system for Compuer-Aided Circuit Analysis, Artificial Intelligence 9, 1977, pp. 135-196.

[Stickel 82] M. Stickel, A Non-clausal Connection Graph Theorem Prover, Proc. of AAAI-82 National Conference on Artificial Intelligence, Pittsburgh, Pennsylvania, 1982, pp. 229-233.

[Stickel 85] M. Stickel, Automated Deduction by Theory Resolution, Journal of Automated Reasoning 1, 1985, pp. 333-355.

[Siklossy et al. 73] L. Siklossy, A. Rich, V. Marinov, Breadth-First Search: Some Surprising Results, Artificial Intelligence 4, 1973, pp. 1-27.

[Sussman & Steele 80] Gerald J. Sussman, Guy Lewis Steele, CONSTRAITS — A language for Expression Almost-Hierarchical Descriptions, Artificial Intelligence 14, 1980 pp. 1-39.

[Turner 79] David A. Turner, Another Algorithm for Bracket Abstraction, The Journal of Symbolic Logic 44, 2, June 1979, pp. 267-270.

[Trybulec & Blair 85] A. Trybulec, H. Blair, Computer Assisted Reasoning with MIZAR, Proc. of IJCAI-85, Los Angeles, Ca., August 1985 pp. 26-28.

[Walther 84a] Christoph Walther, Unification in Many Sorted Theories, European Conference on Artificial Intelligence, 1984, pp. 593-602.

[Walther 84b] Christoph Walther, A Mechanical Solution of Schubert's Steamroller by Many-Sorted Resolution, Procedings of AAAI-84, pp. 330-334.

[Waltz 75] David L. Waltz, Understanding line drawings of scenes with shadows, in *The Psychology of Computer Vision*, Patrick H. Winston ed. McGraw-Hill, 1975.

[Weyhrauch 77] Richard Weyhrauch, Arthur Thomas, FOL: A Proof Checker for First Order Logic, Stanford Artificial Intelligence Laboratory Memo AIM-235.1, 1977.

[Weyhrauch 80] Richard Weyhrauch, Prolegomena to a Theory of Mechanized Formal Reasoning, Artificial Intelligence, vol. 13 no. 1,2, April 1980, pp. 133-170.

[Wos 82] L. Wos, Solving Open Questions with an Automated Theorem-Proving Program, Proc. 6th Conference on Automated Deduction, New York, Lecture Notes in Computer Science 138, Springer-Verlag 1982 pp. 1-13

[Wos & Winker 84] L. Wos, S. Winker, Open Questions Solved with the Assistance of Aura, in Automated Theorem Proving After 25 Years, Vol. 29 of Contemporary Mathematics (W.W. Bledsoe and D. W. Loveland Eds.), AMS, Providence, Rhode Island, 1984, pp. 73-88.

[Wos et al. 84] L. Wos, R. Overbeek, E. Lusk, J. Boyle, Automated Reaoning: Introduction and Applications, Prentice-Hall, Englewood Cliffs, 1984

[Wos et al. 85] L. Wos, F. Pereira, R. Hong, R. Boyer, J. S. Moore, W. W. Bledsoe, L. J. Henschen, B. G. Buchanan, G. Wrightson, C. Green, An Overview of Automated Reasoning and Related Fields, Journal of Automated Reasoning 1, 1985, pp. 5-48.

[Wu 86] Wu Wen-Tsun, Basic Principles of Mechanical Theorem Proving in Elementary Geometries, Journal of Automated Reasoning 2, 1986, pp. 221-252

Index

Page 90 is missing.