Purdue University
Purdue e-Pubs

Open Access Dissertations | Theses and Dissertations

8-2016

Interactive logical analysis of planning domains

Rajesh Kalyanam
Purdue University

Follow this and additional works at: https://docs.lib.purdue.edu/open_access_dissertations

 Part of the Artificial Intelligence and Robotics Commons, and the Computer Engineering Commons

Recommended Citation
Kalyanam, Rajesh, "Interactive logical analysis of planning domains" (2016). *Open Access Dissertations*. 778.
https://docs.lib.purdue.edu/open_access_dissertations/778

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.

Graduate School Form 30
(Revised 08/14)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Rajesh Kalyanam

Entitled: Interactive Logical Analysis of Planning Domains

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

ROBERT L. GIVAN

AVINASH C. KAK

JEFFREY SISKIND

SURESH JAGANNATHAN

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification/Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

ROBERT L. GIVAN

Approved by Major Professor(s): _____

_____

Approved by: V. Balakrishnan 07/15/2016

Head of the Department Graduate Program Date

INTERACTIVE LOGICAL ANALYSIS OF PLANNING DOMAINS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Rajesh Kalyanam

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2016

Purdue University

West Lafayette, Indiana

Dedicated to my loving family

ACKNOWLEDGMENTS

First and foremost I must express my sincerest gratitude to my advisor Professor Robert Givan for his sustained support and guidance through this research endeavor. His enthusiasm, clarity of thought and vision, and singular focus are qualities that I will constantly strive to achieve. I have to also thank him for introducing me to the wonderful world of automated reasoning in general and Ontic in particular which provides endless possibilities for future exploration. This has been an incredibly satisfying learning experience which was enriched by his involvement. I would also like to take this opportunity to thank my committee members, Professors Jeff Siskind, Avi Kak and Suresh Jagannathan for their valuable comments and insightful questions that I hope to apply to any future work.

This enterprise would not have been possible without the tireless support and encouragement from my parents and brother, Suresh. They selflessly helped me every step of the way through graduate school, never once asking for anything in return. I am truly blessed to have such a caring and loving family and could not have wished for better parents or a better brother. Suresh's wife, Aparna and their kids always made me feel welcome in their home and made every holiday a memorable break away from graduate school.

I must also thank my labmate, Tanji Hu who helped get Ontic up and running again and provided me with benchmark results for planning verifications in the Coq proof assistant. My colleague and close friend Jungha Woo was great company during my graduate assistantship days and has helped me out of a bind several times through the years. My close friend and roommate from my undergraduate days, Vivek Keerthy, was a constant sounding board for my ideas and kept encouraging me to see this through. Last but certainly not the least, I must acknowledge the ever reliable support of my managers, Carol Song and Lan Zhao in Research Computing at Purdue

who always found the funding to sustain me through most of my graduate studies. My work with them was a welcome change of pace from my research and gave me an opportunity to attend and present at several conferences through the years. Without their support, none of this would have been possible.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Kalyanam, Rajesh PhD, Purdue University, August 2016. Interactive Logical Analysis of Planning Domains. Major Professor: Robert L. Givan.

Humans exhibit a significant ability to answer a wide range of questions about previously unencountered planning domains, and leverage this ability to construct "general-purpose" solution plans for the domain.

The long term vision of this research is to automate this ability, constructing a system that utilizes reasoning to automatically verify claims about a planning domain. The system would use this ability to automatically construct and verify a generalized plan to solve any planning problem in the domain. The goal of this thesis is to start with baseline results from the interactive verification of claims about planning domains and develop the necessary knowledge representation and reasoning methods to progressively reduce the amount of human interaction required.

To achieve this goal, a representation of planning domains in a class-based logic syntax was developed. A novel proof assistant was then used to perform semi-automatic machine analysis of two benchmark planning domains: Blocksworld and Logistics. This analysis was organized around the interactive formal verification of state invariants and specifications of the state-change effects of handwritten recursive program-like generalized plans.

The human interaction required for these verifications was metered and qualitatively characterized. This characterization motivated several algorithmic changes to the proof assistant resulting in significant savings in the interactions required. A strict limit was enforced on the time spent by the base reasoner in response to user queries; interactions taking longer were studied to direct improvements to the inference engine's efficiency. A complete account of these changes is provided.

1. INTRODUCTION

Proof assistants are computer programs that aid the development of machine verified formal proofs. A natural analogy can be drawn between the human-computer interaction in a proof assistant and the student-teacher dialogue in the Socratic teaching method. The teacher presents a logical argument to a student as a sequence of statements with each statement either being accepted by the student or requiring further explication on the part of the teacher. As part of this explication, the teacher poses simpler queries to the student which then lead back to the original statement. Proof assistants often differ in the nature of such interaction; some may simply respond with a yes, no or don't know to user queries while others may suggest simpler arguments as stepping stones towards the complete proof. The representation language underlying the interaction is usually rich enough to support the definition of relevant concepts in diverse domains. In addition, knowledge is cumulative, allowing for the recollection of previously verified definitions and statements in subsequent proofs. This raises the important question of relevance; which previously verified concepts are pertinent to the current proof? For instance, the fact that the length of a list is one plus the length of the sublist excluding the first element is vital in the proof that the length of a list is equal to the length of its reverse. It is however, not relevant to a proof of the fact that a function from characters to integers mapped over the elements of a list of characters produces a corresponding list of integers. Proof assistants differ in their approach to relevance; some proof assistants require the user to explicitly mention the concept or statement to employ, some allow the user to denote a set of statements or definitions as automatically applicable in all proof efforts, while others attempt to automatically detect relevance.

The central role in any proof assistant is played by an inference engine that verifies that the user's statement logically follows from previous knowledge about the domain.

An efficient inference engine is important to ensure timely responses from the proof assistant, allowing the user to formulate alternate or more detailed arguments where required. Ideally we would like to ensure that each reasoning question posed to the proof assistant is answered (or left unanswered) after only polynomially many reasoning steps. While this may be achieved by severely restricting the representation language, we would like to support an expressive and general-purpose language that can naturally represent a wide range of statements about different domains. Proof assistants thus have to strike a balance between these various conflicting requirements: have a rich enough representation language, a strong inference engine that can verify user claims and enable better proof automation; while being efficient enough to respond quickly to user queries. In the first part of this thesis I describe my work tackling each of these requirements in a general-purpose verification system, Ontic.

In the latter part of this thesis, I present a novel application of Ontic to the task of automated planning, the branch of artificial intelligence concerning decision-making problems. Humans are generally able to "understand" and answer questions about hypothetical planning scenarios. For instance, consider a stack of blocks on a table. Questions such as "Can we get to every block in the stack?" Or, "Can we get every block on the table?" Or, "Can a cycle of blocks be created?" are readily answerable. Such knowledge also appears to enable the checking (and sometimes, even the construction) of general-purpose solutions for abstract problems in these scenarios. For example, "Put all the blocks on the table and then build the desired stack from the bottom up" can be seen upon examination to be a correct solution to rearrange the blocks on the table into any arbitrary stack. Furthermore, no apparent consideration of a particular stack of blocks seems required to answer these questions. In addition to being able to construct such general-purpose solutions, humans are also able to identify invariants that can aid in the detection of unsolvable problems. For instance, consider a hypothetical package delivery scenario where there is no path between two packages. It is immediately apparent that no single truck can deliver both packages; any solution to this problem requires atleast two trucks. One plausible

perspective (and the position taken in this thesis) is that deduction plays a key role in these human abilities.

The long term vision of automated planning is to develop a system that is capable of effectively constructing and verifying a "generalized plan", a general-purpose solution for any arbitrary planning problem from a planning domain. An important step towards this goal is the building of a system that can quickly verify the correctness of a provided generalized plan. This thesis illustrates how Ontic is used to analyze and understand benchmark planning domains with specific examples of two domains: Blocksworld and Logistics. The analysis of these domains is organized around the verification of state invariants and the program analysis of macros achieving single atomic formulas. State invariants are properties which if true to begin with, are preserved by any sequence of actions performed. They are frequently used as integrity constraints in automated planning [1, 2] and often necessary for the verification of generalized plan correctness. As an example, verifying the correctness of a generalized plan for the Blocksworld depends on understanding the invariant that no cycle of blocks can be created. Macros that achieve single atomic formulas can be composed using program constructs such as conditionals and loops into program-like generalized plans for achieving complex conjunctive formulas. The complete analysis of such macros provides a strong foundation for analyzing the resulting generalized plans.

The human interaction required for these verifications is taken to be a measure of how effective the system is on its own. Such interaction can be qualitatively characterized to determine plausible approaches to eliminating them by informing changes to the inference and proof engine. In particular, the improvements to Ontic described in the first part of this thesis were in part motivated by this characterization.

The following chapters include a description of the Ontic verification system, in particular the representation language, the inference engine used to verify claims, a description of the quantifier instantiation methods that introduce new expressions into the reasoning, a description of Ontic's support for proof automation, a novel

application of Ontic to the logical analysis of planning domains, an overview of related work and finally a summary of the contributions of this thesis.

2. ONTIC

2.1 Introduction

The Ontic verification system described in this thesis is descended from a system by the same name created around 1990 by David McAllester. Knowledge representation in Ontic is organized around the notion of a "class", a collection or set of objects. Classes can also be thought of as the set of domain objects that satisfy a predicate in first order logic. Classes along with taxonomic relations between classes form the basis of taxonomic syntax. Given the interpretation of classes as objects satisfying a predicate, a taxonomic relation involving classes has a semantically equivalent set of first order logic formulas. However taxonomic syntax has certain computational advantages over standard predicate logic syntax.

Taxonomic syntax gains its advantages from its quantifier-free fragment being more expressive than the corresponding quantifier-free fragment of predicate logic. There are atomic formulas in taxonomic syntax that require quantifiers to be expressed in first order logic. For instance, there is a taxonomic relation "is" that represents a subtype relationship between its arguments. So, `(is P Q)` denotes the fact that "every P is a Q" where P and Q are class expressions representing the class of objects for which the unary predicates P and Q are true respectively. In first order logic, the equivalent formula would be quantified: $\forall(x)(P(x) \implies Q(x))$. Satisfiability of a set of quantifier-free taxonomic formulas has been previously shown to be polynomial-time decidable [3]. This increased expressiveness along with the enriched polynomial-time decidable fragment of taxonomic syntax plays a crucial role in Ontic's reasoning efficiency.

The rest of this chapter contains an overview of the Ontic language, in particular the syntax of class and taxonomic formula expressions, the concept of an Ontic

definition that is used to introduce new concepts into the reasoner's knowledge and finally an account of interactive theorem proving in Ontic.

2.2 Ontic Expressions

Ontic expressions can be broadly divided into two categories: class expressions and taxonomic formulas. The Ontic language provides a nondeterministic Lisp-like syntax for representing these expressions that closely resembles mathematical English. For instance, `(a person)`, `(an edge-on (a vertex (a graph)))` and `(an (the inverse f) x)` are all valid class expressions in the Ontic language that are easily interpretable. Taxonomic relations on class expressions are written in prefix notation similar to function application in functional languages, but read naturally as English statements in infix syntax. For instance, `(is (a person) (a mammal))` is read naturally as `(a person) is (a mammal)`, and `(is-never (a chair) (a book))` as `(a chair) is-never (a book)`. Having a natural representation language close to English helps the user translate their informal reasoning into syntactically close formal proof steps when interacting with Ontic.

2.2.1 Class Expressions

Ontic class expressions generalize both logical terms (class expressions with exactly one value) and the programming language notion of a "type". Starting from a small set of primitives built into Ontic, class expressions can be constructed to represent all of ordinary Mathematics. Variables, quoted symbols and the numbers 0 and 1 are the basic terms in Ontic. The Ontic language is typed and hence variables have an associated class as their type. The variable itself corresponds to exactly one value from this nondeterministic class. In the special case where the class is itself singleton (i.e. has exactly one value), the variable and the class are equivalent. There is also a class with no values `(fail)` that is equivalent to natural class expressions that have no values; for instance "the integer that is the square root of two".

Quoted symbols and the numbers 0 and 1 are members of the built-in classes `(a symbol)` and `(an integer)` respectively. There are four other built-in classes that account for most of the terms in ordinary Mathematics: `(a set)`, `(a type)`, `(a cons-cell)` and `(an operator)`. `(a set)` is a class corresponding to the ZFC theory notion of a set. `(a type)` is a class that is isomorphic to `(a set)`, but is included in Ontic as "syntactic sugar" for cases where a "type" is more intuitive; for instance as the domain of an operator. `(a cons-cell)` has its origins in Lisp and is the class all of whose members are pairs of class expressions. This naturally generalizes to all sequences since a sequence can be represented as a pair whose first element is the first member of the sequence and whose second element is itself a sequence corresponding to the rest of the sequence. The class `(an operator)` is the collection of operator spaces whose domains are regular sets. These four built-in classes also have corresponding constructive and destructive operators. For instance `the-set` and `the-type` are unary operators that collect the members of their class argument into a set and type respectively[1]. The `member` operator when applied to a set or type, produces a class whose elements are members of the set or type. The `cons` operator is used to combine two class arguments into a `cons-cell`. Just as in Lisp, the `car` and `cdr` operators when applied to a `cons-cell` return the first and second elements of the pair respectively. An operator space can be constructed using the `operator, total-operator, partial-function, function` constructors with the domain and range type class based on the desired cardinality. For instance, `(an operator from integer to integer)` represents the class of operators that take an integer as an argument and return zero or more integers when applied. The negation operator in Mathematics and the increment operator in programming languages are both members of this class. It is to be noted that `integer` in this example is the type class corresponding to the built-in class of all integers `(an integer)` and is equivalent to and a valid shorthand for the class expression `(the-type (an integer))`. There is a universe of objects in these built-in

[1]While this is usually true, there are some valid class expressions where the application of `the-set` or `the-type` does not produce a ZFC set.

classes, called `(a thing)` which can be considered to be the non-deterministic union of these six classes. This universe is termed the "predicative universe" and all of normal Mathematics can be perfomed within this universe. However, Ontic does not prevent writing class expressions that do not fall within the predicative universe. For instance `(the-set (a thing))` is a valid class expression, but is not a member of the class `(a set)` since this would lead to Russell's Paradox. Classes constructed using `the-set` and `the-type` operators are more generally termed "set-class" and "type-class" expressions on which set operators such as `member` and `subset` can still be applied. Similarly, while all members of the `(an operator)` class are operators whose domain and range types are in the class `(a type)`, operator classes constructed with "type-class" types for the range and domain are more generally termed "operator-class".

Ontic class expressions are closed under the application of non-deterministic union, intersection and difference operations. They can be combined with formulas into a conditional expression using the programming language constructs "when" and "if" to filter objects that satisfy a formula. The true expressive power of class expressions is derived from the ability to create arbitrary operators using dependently typed lambda expressions where the types themselves are arbitrary class expressions. Class expressions are closed under the application of such operators giving rise to interesting types. Application of a lambda to a non-deterministic class produces a new class collecting the image of that operator applied to each member of the argument class. For instance consider these diverse class expressions, "(a brother (a policeman))", "(the reverse (a list))" and "(the make-clear s b)". The articles "a", "an" and "the" are used to construct applicative Ontic expressions. The application of the `brother` relation to the class of policeman, returns a class all of whose members are brothers of policemen; the `reverse` operator can be defined recursively using a lambda expression and when applied to the class of lists (or sequences), produces the class all of whose members are the reverse of some particular sequence from the universe of objects. The `make-clear` operator can be defined recursively to operate on a planning state

`s` and a block `b` in the state to produce a resulting state where `b` does not have any blocks above it. The exact definition of this operator is provided in a later chapter on planning verifications using Ontic and illustrates the ability to compose operator applications, conditionals and recursion to define complex non-deterministic functions similar to Lisp programming. The complete grammar of Ontic class expressions is provided in the appendix.

2.2.2 Taxonomic Formulas

The primary Ontic atomic formula is the binary "is" formula, testing the subset relation between two classes. An "is" formula makes a typing or sub-typing assertion that one class is contained within another, generalizing the predicate logic equality test to classes. If both classes in the assertion are singleton (single-valued), then the "is" typing test reduces to the standard predicate-logic equality test. This enrichment of predicate logic to typing is vital to Ontic's performance.

Other atomic formulas are provided to assert that a class is non-empty, or that a class is deterministic (has at most one value). Some Ontic formulas are provided as "syntactic sugar" to improve the efficiency of the inference engine via dedicated rules and to improve user interaction. For instance, there is a binary atomic formula "is-never" testing disjointedness of two classes which could have been represented as `(not (there-exists (both C1 C2)))` where "both" is the intersection operation and "there-exists" is the atomic formula testing that a class is non-empty. Boolean operators combining atomic formulas are provided, as well as universal and existential quantification (for convenience, as quantification is already representable using the atomic formulas together with the rich class expression language.) Taxonomic formulas gain their expressiveness from the underlying class expressions. For instance, the fact that there are no cycles in a graph "g" can be represented as $(\textit{is-never}\ v\ (a\ (\textit{the edge-relation}\ g)^*\ (a\ (\textit{the edge-relation}\ g)\ v)))$ quantified over vertices "v" in the graph. The "edge-relation" is an Ontic operator that defines the

edges of graph "g" as an operator from vertices to vertices in lieu of the traditional ordered vertex pairs representation. A complete grammar of Ontic formulas is provided in the appendix.

2.3 Ontic Definitions

The primary means of introducing new concepts into Ontic's reasoning is via Ontic definitions. An Ontic definition defines a symbol to denote an Ontic class expression. Since symbols are assumed to assign a name to a singleton class (i.e. correspond to a single object in the domain), the class expression being defined must be singleton. The Ontic definition syntax is similar to Lisp's symbol and function definition syntax. Ontic has a "define-constant" keyword that is used to define a symbol to represent a particular arbitrary member of its argument class. All other definitions use the "define" keyword. Lambda expressions are always singleton since they denote exactly one way of mapping the arguments to a non-deterministic output; hence defined symbols corresponding to multi-argument operators do not need the "define-constant" keyword. Multi-argument operator definitions allow for dependently typed arguments where types can be complex class expressions. There is a special case of an operator definition: one that does not take any arguments. Such definitions are used to define a new type class. The defined symbol in this case is taken to be the type built from the non-deterministic class in the definition. For instance, consider these following definitions:

```
(define-constant x
 (an integer))

(define-constant y
 (an integer))

(define (a list-of-symbols)
  (either 'nil (the cons (a symbol) (a list-of-symbols))))

(define (the length (l (a list-of-symbols)))
  (if (= l 'nil)
     0
     (the sum 1 (the length (the rest l)))))
```

While `x` and `y` have the same definition body, they may or may not correspond to the same arbitrary integer object. If the class `(an integer)` were singleton, `x` and `y` would necessarily be the same object. The symbol `list-of-symbols` can be taken to denote an Ontic type, one whose members are all possible lists of symbols. The symbol `'nil` is taken to denote the empty list. The symbol `length` denotes a lambda expression whose domain type is the Ontic type `list-of-symbols` defined before.

The semantics of the application of defined symbols are implemented using beta reduction and eta abstraction rules. Application in Ontic is "curried"; an application of a multi-argument operator to an argument vector is interpreted as a sequence of nested applications starting from the inside out. Each intermediate partial application corresponds to the lambda expression resulting from substituting in the applied argument for the corresponding formal parameter.

Ontic definition forms allow proof statements to be included as part of the definition. This serves a number of purposes: first it enables natural proofs about the newly defined symbol (for instance proofs about the output types of the defined operator) to be included close to the definition; secondly Ontic treats proofs inside a definition differently; beta reduction of applications of the defined symbol is carried out automatically (with suitable restrictions to handle infinite expansion of recursive definitions), allowing for proofs to "see" the fully expanded form; finally proofs about the definition may be necessary for Ontic to accept the definition. In a subsequent chapter, Ontic's "typechecking" process will be described that validates Ontic expressions to ensure that all function applications are well-typed, i.e. the arguments to a function are in the corresponding domain type. To ensure that a Ontic definition typechecks, the user may need to aid the proof of typechecking claims about expressions in the definition. Such proofs can be included in the body of a definition to be pre-processed before the definition is verified and accepted.

There is a special definition form in Ontic that defines a "structure". Similar to the typical programming language notion of a structure, Ontic structures defined using the "defstruct" form group together fields to form a structure object. Processing

the definition defines a structure class, and the corresponding constructor and slot accessor functions that can be applied to objects in the structure class. The fields of the structure class can have arbitrary dependent types and are restricted to be singleton values for a particular structure object. Ontic structures provide a natural way to represent structures and classes from programming languages in addition to mathematical constructs such as graphs and automated planning constructs such as planning domains or states. For instance, a natural way to represent a graph is using a structure with two fields, the vertices and the edges. Similarly, a natural way to represent a planning domain is with a structure grouping the domain's object types, predicates and actions.

```
(defstruct directed-graph
 (nodes (a type))
 (edges (an operator from nodes to nodes)))
```

2.4 Recursion in Ontic

The Ontic language allows users to type recursive definitions which can subsequently be used in induction-based proofs. Such recursive definitions are only allowed for operator (and type class) definitions. All recursive definitions need to satisfy a semantic fixed point condition. Any recursively defined symbol in Ontic is assigned a well defined meaning; the transfinite limit of a series of approximations of the defined class. In short, an approximation of the definition can be defined for each ordinal α as follows: the approximation corresponding to the least ordinal is a version of the same definition except that the body does not contain any recursive calls, the approximation for an ordinal ω is a version of the same definition except that all recursive calls use the approximation corresponding to the preceding ordinal. The semantic fixed point condition requires Ontic to be able to automatically verify that this transfinite limit is a fixed point of the definition. Any such "valid" recursive definition can be used to carry out proofs via computational induction. Suppose we have an recursive operator foo, and an arbitrary member x from the class expression $(foo\ args)$ where $args$ are appropriate arguments to foo. The existence of such

an x implies that the computation of $(foo\ args)$ terminates and hence there is a well-founded order on the sequence of recursive computations used in obtaining x. Induction can be carried out on this order when trying to prove facts about x. In the special case of a recursively defined type *bar*, and an arbitrary member y from the class expression `(a member bar)`, y must be introduced at some approximation of *bar*, enabling transfinite induction.

Ontic provides a "show-by-induction-on" proof form that automates the setting up of the induction proof. This proof form enables the proof of a fact by induction on a recursive operator or type class. The desired fact needs to be universally quantified over the output of the recursive operator (or all members of the recursive type class). When executed, the proof form automatically sets up an induction proof by defining a version of the recursive operator (or type class) termed a "wishful-version" for which the universal fact is true. The user is then responsible for conducting the induction i.e. proving that the fact holds for all outputs of the class where all recursive calls have been replaced with the "wishful-version". For instance, returning to the previous recursive definition of a list of symbols; the proof of a fact about all such lists typically involves induction. A "show-by-induction-on" proof would set up an "wishful-version" of this definition and the user would need to prove the desired fact about an arbitrary member of the class:

```
(show-by-induction-on ((l (a list-of-symbols)))
 (is (the length l) (an integer))
 (suppose (= l 'nil))
 (suppose (not (= l 'nil))
   (show (is (the sum 1 (the length (the rest l))) (an integer))
      (show (is (the length (the rest l)) (an integer))))))
```

The proof above represents an induction proof of the fact that the length of any list of symbols is an integer. It should be noted that Ontic typechecking of the recursive definition of `length` can automatically infer this fact without any user proof. This proof is provided simply to illustrate the structure of a "show-by-induction-on" proof. When Ontic processes, this "show-by-induction-on" proof form, it sets up a "wishful-version" of the `list-of-symbols` definition such that the induction hypothesis is asserted about all its members; in particular that they have integer lengths. The

induction variable `l` is then assumed to be under a version of the `list-of-symbols` definition body where all recursive calls to `list-of-symbols` have been replaced with the "wishful-version":

```
(either 'nil
    (the cons (a symbol) (a (the wishful-version list-of-symbols))))
```

In the case where `l` is not empty, the sublist consisting of all but the first element of `l` is in this wishful type and hence has an integer length. The rest of the proof is then trivial.

2.5 Interactive verifications in Ontic

Ontic supports the interactive development of verifications by providing a sound polynomial-time procedure for checking specific entailment claims. This *base reasoning procedure* can be thought of as modeling the human notion of what is "obviously entailed" [4]. The procedure is sound but not complete; i.e., a possible response is "I don't know."

Ontic provides a very simple "Socratic proof system" [3] that enables a human user to prove any entailed claim by verifying a sequence of entailment claims with the base reasoner. Previously verified claims join the premise set, thus enabling the verification of complex claims in a sequence of verification steps. The Socratic proof system also enables the user to specify case analyses, combining previously checked entailments, as well as universal generalization.

The number of base-reasoner proof steps (i.e., the number of human interaction steps) needed to verify a claim can be considered as a metric on the effectiveness of the verification system to verify claims automatically. Qualitative characterizations of the necessary human interaction steps can be used to identify types of interactions that can be potentially eliminated with changes to the base reasoner and sequent system. Ontic also provides a user interface for converting natural-style mathematical proofs into sequent proofs automatically. As a result, human-written Ontic proofs are ex-

ceptionally readable and similar in style to the informal proofs found in mathematics textbooks:

```
(suppose-there-is ((l (a list-of integer)))
 (suppose (not (= l 'nil))
   (show (is (the first l) (an integer)))))
```

The base reasoner is implemented by a set of forward-chaining inference rules representing the basic properties of the Ontic language constructs. For example, the rules implement the transitivity of subset, the simplification of conditionals when the test is known, the relationship between union and subset, the distribution of application across union, etc. However, these rules are restricted to fire only when no new expressions can be introduced (all expressions need to either appear in the premises or the proposed conclusion, i.e., in the "query"); this restriction ensures a polynomial worst-case runtime [3].

This basic forward-chaining reasoning with the given expressions is enriched by multiple carefully-controlled mechanisms for limited introduction of new expressions into the forward-chaining reasoning. There are two primary methods for introducing new expressions, automatic quantifier instantiation (including beta reduction) and Skolemization of applicative class expressions. The sequent proof system can also introduce new expressions by automatically employing alternate strategies (termed "tactics") for proving the current claim. For instance, in a proof of an "is" fact, where the class being typed is an application of a defined symbol, Ontic may automatically beta reduce the application, introducing the result into Ontic's reasoning. Also, other tactics may be attempted on this resulting expression; in case the result is a "if" expression, case analysis may be employed to prove the subtyping relation for each branch of the "if" separately. A description of these tactics follows in a later chapter.

2.5.1 Proof Language

The Ontic proof language provides several proof constructs that enable users to type natural proofs in a style close to the typical development in a Mathematics

textbook. When a proof is evaluated, Ontic semantically transforms these constructs into verifiable sequent proofs. New Ontic proof constructs are defined using a proof macro definition language for composing existing proof constructs and proof sequents. A proof macro definition consists of a unique proof macro name, a parameterized proof form, a condition and a proof body. The proof form can contain variables that can be referred to in the condition and body. When a user proof statement is evaluated, an applicable proof macro is determined by matching against the defined proof forms. A successful match binds the variables in the proof form, following which the condition is tested. If it evaluates to true, the user's proof statement is replaced with the instantiated proof body from the definition and evaluated. Thus, the proof macro definition language can be thought of as defining rewrite rules for proof constructs that transform higher level proof constructs into low-level sequent proofs. Different proof macros can be defined for the same proof form with varying conditions; the first matching proof form with a condition that is true is chosen for evaluation. The ability to combine proof constructs in interesting ways is similar in scope to the notion of "tactics"in other proof assistants. The result of evaluating a proof is a sequence of "context extensions" (typically theorems) that now join Ontic's premise set.

show

The central proof construct in Ontic is "show"; which instructs Ontic to verify a claim (provided as an Ontic formula argument to "show") using its base reasoner. All proof evaluation occurs in a context which has two components: the "lemma library" that contains all the definitions, axioms and results of proofs evaluated so far, and the "local context" that is constructed from local assumptions, variable introductions and the semantic transformation of the current proof construct. Ontic uses forward chaining reasoning in this context to verify a "show" statement's "goal". However, in addition to using the base reasoner, Ontic also automatically employs additional "tactics" to attempt to verify the claim. For instance, when verifying a

universal fact, Ontic automatically introduces new variables for each of the quantified types and attempts to verify the body of the universal instantiated on these new variables. Similarly, when verifying a disjointedness fact (i.e. that two classes don't have any members in common), Ontic employs refutation by assuming the existence of common members, introducing a new variable from the intersection of the two classes and checking for a contradiction. Another major component of Ontic's automatic verification of a "show" goal is "case analysis". When trying to prove a claim, it often helps to split the task into several "cases"; for instance when trying to prove that for every natural number either one plus the natural number or two plus the natural number is even, it helps to consider two possibilities and prove the claim separately for each of these possibilities: either the natural number in question is even or it is odd. More details on Ontic's automatic case analysis can be found in a subsequent chapter.

If none of these attempts succeed, Ontic returns to the user saying that it cannot verify the claim. It is then the user's responsibility to setup a different proof that may involve instructing Ontic to verify supporting claims. These supporting proofs are typically placed inside the body of the "show" proof. When Ontic evaluates a "show" proof, it first attempts to use the base reasoner to verify the claim, following which the user's proof body is evaluated and finally the additional proof tactics described in the previous paragraph are employed. Proof evaluation also employs short-circuiting; if a surrounding "show" goal can be verified in the current proof context (which does not contain any new assumptions or variables that were not present in the "show" proof's context) that "show" is assumed to have succeeded.

suppose

"suppose" is an Ontic proof construct that introduces an assumption. Users often have to employ their own case analysis in proofs, and thus need to introduce assumptions. As in the case of "show", the body of the "suppose" can contain Ontic

proofs. The result of evaluating a "suppose" is a set of implication formulas where the formula being assumed is added as an implicant for any claims verified in body of the "suppose".

Some proofs forms retain the immediately surrounding goal (i.e. the claim currently being verified in the closest surrounding "show" form); "suppose" is one such form. After the body of the "suppose" is evaluated, Ontic automatically attempts to verify the surrounding goal, thus the user doesn't have to add another explicit "show" to this effect. Similarly, the user never has to explicitly include "suppose" forms for both a formula and its negation if one case can be verified automatically by Ontic. Recall that Ontic attempts to verify the "show" goal after evaluating the user provided proof body. All uses of Ontic's base reasoner to verify a claim employ proof by refutation, i.e. the negation of the claim to be verified is assumed and a contradiction is sought. In a context where Ontic has an implication statement about a goal (the result of evaluating a "suppose" inside a "show"), refutation of the goal makes the negation of the assumed formula true. If Ontic's base reasoner can verify the claim in this refutation context, a separate assumption of the negated formula is not needed.

suppose-there-is

A claim that is a universally quantified formula can often be verified by universal generalization, i.e., an arbitrary member from the quantified class is chosen and the claim is proven about that member and later generalized to all members of the class. The Ontic proof construct that introduces a new variable belonging to a particular class is "suppose-there-is". The arguments to a suppose-there-is form is a list of possibly-dependent bindings and a proof body. Every valid suppose-there-is proof must contain a "show" form in its body. Any variable that is introduced by a suppose-there-is has to be previously undefined.

It is possible to rely on Ontic's tactics for proving universal facts via generalization, however if Ontic cannot verify the claim, there is no way to refer to the variables chosen for the generalization. The tactics automatically employed by Ontic on a show form are all backtrackable; when any of them fail, Ontic automatically rolls back the context to the show form. In such cases, users would rather introduce the variables intended for generalization themselves to be able to refer to those variables in any subproof. Any theorems returned from the proof body are all universally quantified over the types of all the variables mentioned in the theorem. The argument types in the suppose-there-is bindings are not tested for existence; the proof form assumes the existence of these types when evaluating the proof body. As a result, even if some variable is not mentioned in a theorem result from the proof body, an implication on the existence of the corresponding type is added to the theorem when returned from the suppose-there-is form.

There is a special form of suppose-there-is that is used for Skolemization. Any applicative class expression in Ontic can be Skolemized by picking new variables for the class arguments in the application. Suppose there is an applicative expression $(a\ foo\ type_1\ type_2\ ...\ type_n)$ and a member of this applicative class z. From the existence of z, we know that there exist members x_1 through x_n from the classes $type_1$ through $type_n$ such that z is a member of the class $(a\ foo\ x_1\ x_2\ ...\ x_n)$. This is the basic principle behind the "suppose-there-is such-that" form that introduces such variables x_1 through x_n via the suppose-there-is binding list, but also restricts them with a "such-that" formula that specifies the relation between z and these variables: $(is\ z\ (a\ foo\ x_1\ ...\ x_n))$. When processing this proof form, Ontic verifies that such members do exist in the binding types $type_1$ through $type_n$ before evaluating the proof body.

Non-semantic proof constructs

In addition to the commonly used proof forms described above, there are some non-semantic proof forms that exist only to introduce non-semantic extensions into the context when evaluating a proof. For instance, proofs requiring special consideration of some expression in either quantifier instantiation or Skolemization, can contain proof forms termed "control predicates" or "control extenders". Such proof forms, typically have a corresponding non-semantic predicate or context extension whose only task is to denote additional "focus" on certain class expressions. For instance a "user-typed" proof form is used to identify expressions that need to be considered typed by the user and hence processed aggressively in quantifier instantiation, and Skolemization.

It is important to note that such non-semantic proof forms do not affect the validity of the proof; they exist simply to introduce new expressions into the inference. Other proof forms exist simply to control the current goal in a "show" proof. Recall that both the "suppose" and "suppose-there-is" proof forms attempt to prove the goal from a surrounding "show" form at the end of the evaluation of their bodies. Sometimes this may not be desired if the user expects that the goal is not yet provable without further sub-proofs. The "lemma" proof form allows disabling such surrounding goals.

3. INFERENCE ENGINE

3.1 Introduction

The base reasoner in Ontic carries out forward-chaining reasoning driven by inference rules that implement the semantics of the Ontic language. A typical Ontic inference rule consists of a set of parameterized antecent formulas and a parameterized conclusion formula where the parameters are variables standing in for Ontic classes and formulas. The variables in the conclusion are a subset of the variables in the antecedent set. Each parameterized formula can contain complex Ontic expressions built from these variables as subexpressions. In a canonical rule, no new expressions can be created when the rule fires and hence the rule can fire only if an assignment to the variables exists satisfying both the requirement that the class expressions in the instantiated (with the variable assignment) antecedent and conclusion formulas are present in Ontic's context and that all the antecedent formulas are known true. When both these requirements are satisfied, the (instantiated) conclusion formula is made true. For the rest of this thesis the presence of an expression (class or formula) in Ontic's inference context will be referred to as the "internedness" of that expression in the context. Any new expression created by Ontic via inference or that is entered into Ontic's context by user input is then said to be "interned". As mentioned previously, the class expressions in Ontic's context can be expanded by carefully controlled mechanisms typically involving quantifier instantiation and Skolemization. Such "interning" inference is implemented using a special kind of rule, a "ruleset" that allows for new expressions to be interned in the conclusion. The amount of interning allowed is controlled by "control predicates" tested in the antecedents of such rules and results in a layered interning approach. Control predicates are asserted on expressions

that need aggressive processing, but are not transferred to newly interned expressions created by interning rules; thus preventing the rule from firing on the new expression.

The primary expense in implementing an inference rule is in responding to every fact that becomes true that matches any one antecedent of the rule. All other antecedents of the rule need to be "checked" at this point to determine if the rule can fire with this new information. It is important to note that "checking" of an antecedent may involve finding all possible variable substitutions that make the antecedent true. Ontic requires rules to satisfy a "threading" requirement, it must be possible to bind every variable in a rule starting from the partial substitution corresponding to any one true antecedent. There are two approaches to implementing this that represent a space versus time tradeoff. In the first approach, a closure can be created from a partially instantiated rule (where some antecedents have been checked) and attached to the next antecedent fact that we are waiting on. When that fact becomes true, the closure is executed possibly creating a new closure that waits on the next antecedent to become true and so on. Each such closure represents a partial instantiation of a rule where previously verified antecedents no longer need to be re-checked. In the second approach, all the rule antecedents are checked whenever any one of the antecedents become true in a order determined by possible threading orderings. Since the antecedents could become true in any order and no state is maintained, all the antecedents need to be checked every time any one of them becomes true. This approach currently used in Ontic, while more time-intensive has the advantage that there is no long list of closures awaiting execution on a fact to become true. Later sections in this chapter describe approaches employed to improve the performance of the inference engine.

In addition to enabling the definition of new rules and rulesets, the Ontic rule language supports the definition of "orcfuns". In some cases, it is more natural to represent an inference concept as a recursive function being computed on classes, returning a set of values. Such functions can represent computations ranging from the set of classes that are singleton and have certain control predicates asserted about

them, to the set of substitutions for instantiating a quantified formula. It is straightforward to convert such functions into rulesets; the computation of the function on class arguments can be represented as a formula assertion on those classes and the return value of a function computed on class arguments can be represented using a relation between the function call and the return value. An Ontic orcfun definition is automatically processed to create the necessary formulas and rulesets implementing the semantics of the definition. A special kind of antecedent can be used in rulesets to bind the output of a computation of an orcfun on parameterized expressions to a rule variable. The output variable need not be mentioned in any of the other antecedents, however all the variables in the arguments to the orcfun need to occur in other antecedents. A later section describes some of the performance improvements afforded by the orcfun model.

Unlike most other reasoning, equality reasoning in Ontic is implemented outside of rules. An efficient implementation of congruence closure is used to propagate newly discovered equalities through the currently interned expressions. This is described later in this chapter in conjunction with a description of Ontic's internal representation of classes and formulas. This internal representation is central to the implementation of predicate, relation and internedness checking in rules. Changes to the internal representation that enabled improvements to the rule implementation are also discussed below.

3.2 Ontic Objects

Ontic expressions (classes and formulas) are managed in a production grammar where the non-terminals are the Ontic objects representing classes and formulas. A production can be built from a constructor applied to either zero, one or two arguments. Ontic expressions that have more than two arguments (for instance an `if` expression) are coerced into one of these forms by combining the arguments beyond the second one into a list. A non-terminal can have one or more productions associated

with it, corresponding to the various expressions that can be used to denote the same class. For instance the class expressions `(+ 0 0)` and `(+ (the negation 1) 1)` are both equivalent expressions for the class `0`. An Ontic object is taken to be the equivalence class of all the (possibly infinite) expressions that can be constructed using its productions from the grammar. The process of "interning" an expression can then be considered to be the process of adding this expression to the production grammar and returning the non-terminal corresponding to the Ontic object that the expression denotes.

A crucial part of maintaining the production grammar is to update it in response to newly discovered equalities. First a new equivalence class needs to be constructed by combining the expressions denoting the two classes being equated, and second, this equality needs to be propagated up productions that mention the two classes to discover other equalities that might result. This can be seen to be exactly the "congruence closure" procedure for extending the equality relation. Infact, the decision problem for the quantifier-free theory of equality with uninterpreted function symbols has been shown to reduce to the congruence closure problem [5].

3.2.1 Congruence Closure

Congruence closure is the process of extending a relation such that the resulting extension is an equivalence relation "closed under congruence". While congruence closure is defined in terms of the vertices in a graph, it can be extended to a production grammar by considering a graph formulation of the grammar. The terminals and the non-terminals of the grammar can be considered to be the vertices of the graph while the productions define the edges; an edge exists between each non-terminal and the non-terminals and terminals involved in one of its productions. The non-terminals are labeled by the constructor of the production on the right-hand-side. Given a relation R on the vertices, two vertices u and v are said to be congruent under R if their out-degrees and labels are the same and every pair of corresponding (according

to an ordering on the edges) descendants is in the relation R. R is said to be "closed under congruence" if for any such pair of vertices, the pair (u, v) is in R as well. The congruence closure of R is then a minimal extension R' of R such that R' is an equivalence relation that is "closed under congruence". The process of computing the congruence closure of a relation can then be defined by defining a procedure for computing the congruence closure of a relation R that was previously closed under congruence but has been extended with a new pair (u, v).

Considering R to be equality relation on Ontic objects, congruence closure can be used to maintain the invariant that given two congruent productions (i.e. with the same constructor and pairwise congruent arguments), the corresponding left hand sides of the production are inferred congruent (equal). Congruence closure has been proven to have a polynomial time worst-case and hence equality reasoning in Ontic can be performed in time that is polynomial in the number of productions.

3.2.2 Predicates and Relations

While equality reasoning in Ontic is performed outside of rules, the inference rule implementation depends on the ability to check the truth of predicates and relations on Ontic objects. Due to Ontic constructors being restricted to zero, one or two arguments, true facts are internally represented using constructors with an arity of atmost two. In addition to checking the truth of a fully instantiated antecedent, it is also necessary to support partially instantiated antecedents that may bind free (upto one) variables with objects satisfying a relation. Ontic objects are internally represented using a structure that has slots for storing the truth value of monadic predicates as well as a list of related objects for each binary Ontic relation. Since either one of the two arguments to the relation could be a free variable in an antecedent, Ontic maintains both "forward" and "backward" lists of related objects depending on whether the object in question is the first or the second argument to the relation respectively. Checking an antecedent then involves either testing the truth value of

a predicate, or looping through the list of objects related by a relation to bind a free variable.

It is obvious that not all predicates or relations are applicable to all Ontic objects. In particular, most predicates and relations only apply to classes. Ontic "categories" can be defined to restrict the argument types and to specify an output type for each Ontic constructor. Each Ontic object then has only the predicate and relation slots for its category. Similarly, typechecking can be introduced into the rule implementation to restrict rule variables substitutions by the categories that can be determined based on the antecedents they occur in.

3.3 Rule Implementation

As mentioned previously, Ontic's rule implementation involves checking every antecedent in the rule whenever any one antecedent becomes true for some set of Ontic objects. If a variable substitution (of Ontic objects) can be found such that every antecedent is true (after the substitution), then the conclusion can be made true. An extendable post-assertion trigger function is defined for each Ontic predicate and relation such that a sequence of attached procedures can be executed whenever a new assertion of that predicate or relation is made. Processing a new Ontic rule then involves processing each antecedent to define such a function that can be attached to the corresponding predicate or relation tested in the antecedent. The steps involved in this processing are detailed below. It is important to note that since the Ontic grammar only allows for constructors with upto two arguments, every antecedent in a rule is either a predicate or relation applied to Ontic expressions. Some rule antecedents are equality antecedents checking for equality between two parameterized Ontic expressions. Such antecedents can be interpreted as checking for an Ontic object that has productions which can provide variable substitutions for the variables on both sides of the equality. In the simplest case, the left hand side of the equality is a single variable that can be substituted with any Ontic object.

Congruence conversion It is important to note that while antecedents can contain complex expressions built out of variables, all predicates and relations are tested on Ontic objects. The task of checking an antecedent containing non-trivial expressions is simplified into two steps. First the top level predicate (or relation) of the antecedent is checked against Ontic objects. Next, a variable substitution for the antecedent expressions is determined by looking at the expressions denoting the succeeding objects from the first step. In order to accomplish this, all rules are preprocessed in a step termed "congruence conversion". Each complex expression in the antecedents and conclusion is replaced with a new variable and a separate equality antecedent is added to the rule for each such expression, equating the new variable to the expression. The process is then recursively applied to the updated rule, until each antecedent is either a predicate or relation symbol applied to rule variables or an equality antecedent where the left hand side is a rule variable and the right hand side is an Ontic constructor applied to rule variables.

3.3.1 Rule Processing

Once a rule has been congruence converted, each antecedent is processed in turn to construct a procedure that can be executed when the antecedent is made true. The antecedent being processed is denoted as the "head" and the bound variables are initially set to be the variables in the antecedent. The bound variables determine an ordering through the rest of the antecedents. The process first attempts to pick fully bound or non-branching antecedents that can be checked next. If no such antecedent can be found, partially bound relations are chosen where the (single) free variable can be bound to any Ontic object in the corresponding forward or backward list for the relation. The remaining antecedents need to be checked for each possible choice for this free variable. It is clear that this can give rise to nested loops in the procedure. The processing is non-deterministic; if more then one choice is detected at a certain

point, then all possible choices are tried with backtracking until an ordering can be found that binds all rule variables.

Recall that in addition to predicates and relation-based antecedents, some rule antecedents are equality antecedents. Just like relations, equality antecedents can be checked when partially bound. However, there are several choices for what variables can be left unbound. In particular, the antecedent can be checked if any of the variables on the right hand side are bound or if the variable on the left hand side is bound. If the variable on the left hand side is bound, then the variables on the right hand side can be bound by looking for a production on the object chosen to bind the variable. Similarly, if a variable on the right hand side is bound, then all productions of the appropriate constructor that include the bound object are used to bind the free variables.

Once an ordering of the antecedents can be determined that can bind all the remaining rule variables, a Lisp function can be constructed that eventually instantiates and asserts the rule conclusion. This function can then be attached to run whenever the predicate or relation corresponding to the "head" is asserted. Since every rule antecedent is processed as a "head", the equality antecedents added during congruence conversion are processed in a similar manner as well. The corresponding functions are then attached to be triggered whenever a new production is asserted on an Ontic object. In addition to checking the antecedents, several checks are introduced at different points in the function to improve the efficiency of rule execution. All such checks are prompted to occur as soon as possible, when the necessary variables have been bound. In particular, for each predicate and relation, the categories of the corresponding arguments are also checked for the variables used in that argument position. Similarly, any loop over objects in the forward or backward relation lists is protected by a test verifying that the lists are non-empty. Changes are also made to the Ontic object structure to enable more efficient checking of productions. Since an object can have several productions using various constructors, rather than have a single list, the productions are organized in an associative list keyed by the constructor.

3.4 Rulesets and Orcfuns

Ontic rules are restricted to have non-interning conclusions, however there are several inference principles that require new expressions to be interned. For instance, both quantifier instantiation and Skolemization when implemented using rules would require interning conclusions.

3.4.1 Rulesets

An Ontic "ruleset" is designed to allow for interning conclusions, while simplifying the task of representing certain inference principles. A ruleset as the name suggests can be considered to be a set of rules that are defined together. This is accomplished by allowing for several parallel sets of antecedents with their own conclusions to be a part of the same ruleset. Such antecedent sets can occur nested inside other antecedent sets, which are taken to be antecedents common to every set element. The ruleset language also provides special macros that enable writing rulesets in a Lisp-like syntax using conditionals and matching. For instance, the "if" ruleset macro allows a natural branch point to be setup in a ruleset depending on whether an antecedent fact is true or false. The "selectmatch" ruleset macro can be used to construct a case split-like structure based on the productions of an Ontic object. The selectmatch macro takes a rule variable as an argument and a list of cases where the case body is a ruleset (possibly containing other antecedents) and the case formula is a production pattern consisting of a constructor applied to rule variables. When processed, each case branch in effect introduces an equality antecedent between the selectmatch variable and the production expression from the case formula.

Ontic rulesets can also introduce non-monotonicity into the inference. The "nonmon-if" ruleset macro functions similar to an "if" except that the else branch uses the "not-proved" construct rather than the "not" negation construct. A ruleset containing a "nonmon-if" construct applied to a formula can be thought of as producing two rules. The first rule corresponding to the then branch, includes the formula being

tested as an antecedent. The second rule corresponding to the else branch, includes a "not-proved" antecedent that does not generate a head. The "not-proved" construct takes a formula as an argument and delays the firing of the rule by queuing the conclusion and also protecting it with a test formula. The intention is to wait for the fact being tested to possibly become true, but then pick elements off the delayed queue to fire when the inference has reached a quiescent state. If the fact is still not known true at this point, then the conclusion fires. This is different from testing for the fact to be known false in the else branch.

In general, Ontic rulesets are used whenever there are several distinct but related inference principles used to derive a particular fact. Rather than separate these into individual rules, they can be combined together into a single ruleset for ease in representation. This is exemplified in defining the inference rules for a matching principle in quantifier instantiation. A matching relation that asserts that a pattern expression can be matched to an Ontic class can be defined recursively by first tackling the base cases and then defining the relation recursively based on matches on subexpressions. Rather than define three separate rules, one for the base case, one for patterns using a unary constructor and another for patterns using a binary constructor; it is easier to combine them all into a single ruleset that uses the "selectmatch" macro to split these three cases.

3.4.2 Orcfuns

While most inference is naturally represented using a set of antecedents deriving a conclusion; some inference is better represented as a function being computed on zero or more class arguments and returning one ore more values. For instance, the substitutions that can be derived from matching a pattern to a class expression, or the Ontic classes that are both singleton and have a control predicate asserted about them, or the quantifier type class at a certain index in a universally quantified formula. In each of these cases, it is natural to think of the inference as a function being

computed on class objects; especially when there is a natural recursive definition for the function. Ontic "orcfuns" are ruleset macros that make it possible to represent such inference as a function definition that is then automatically converted into a ruleset.

Orcfuns are defined using the "deforcfun" keyword by specifying an orcfun name, a list of rule variables as arguments to the orcfun and a ruleset body. The body is similar to other rulesets except for the fact that conclusions use the "return" keyword to denote one of the values of the orcfun. The body in effect represents the "computation" of the values for the orcfun applied to the argument variables. The "computation" of an orcfun on some set of Ontic objects is triggered by the "compute-value" ruleset macro. This macro with syntax similar to "let*" in Lisp, allows a sequence of variable bindings to be specified as variable and orcfun application pairs and has a ruleset body. A variable bound in an earlier binding can be used as an orcfun argument in a later binding. compute-value and return go hand-in-hand; the variable bound to an orcfun application in compute-value can be bound to any of the Ontic objects in the "return" conclusion for that orcfun application.

Ontic orcfuns are implemented using two Ontic formulas: "compute!" and "has-value". compute! is used to assert a computation request for an orcfun on some Ontic objects. It is a unary predicate applied to the special Ontic category, "thunk". Each new orcfun defines a possible production for an Ontic thunk object where the constructor is the orcfun name and the arguments are assumed to be of the top-level Ontic category "anything". The "has-value" formula as the name suggests, is a relation that asserts that an Ontic object is a value of a particular orcfun thunk. The "return" ruleset macro in effect, turns into a "has-value" conclusion. A compute-value in effect splits a ruleset into two rules; a rule with all the prior antecedents upto that point, asserting the compute! conclusion and a separate rule that includes all of the prior antecedents upto the compute-value, a has-value antecedent (checking for a has-value fact on the variable and orcfun application from the compute-value binding) and the rest of the ruleset body of the compute-value. It is important to

note that a compute-value with a list of bindings is processed as a sequence of nested compute-values, one binding at a time. It may be apparent that this implementation is inefficient; the antecedents prior to the compute-value are checked twice; once in the rule concluding the compute! and again in the rule with the has-value antecedent. A solution addressing this is described in the next section along with other rule implementation changes targeted toward improving the efficiency of the inference engine.

3.5 Inference Engine Improvements

As the number of Ontic objects in a context increases, so does the expense of rule execution. In particular, antecedents that are partially bound need to be checked by looping through possible bindings of the free variable, the number of which can be directly proportional to the number of objects in the context. An `is` antecedent where the right hand side variable is bound to a "large" class like `(a thing)` can bind the left hand side variable to any of the Ontic objects that are `is`-related to `(a thing)`. Some pathological cases can contain nested loops over similar long forward or backward relation lists. The long lists need not be restricted to the `is` relation, for instance, a `is-never` antecedent where one variable is bound to the empty class `(fail)` can bind its other variable to any non-empty class.

The expense of a rule can be considered to be the cumulative expense of each of the procedures it attaches to the predicate and relation noticers. The primary expense of these procedures is in the nested loops over the relation lists. Ontic maintains a running total of the number of cycles through these loops, which is taken to the expense of any such procedure.

Another source of rule expense is the number of antecedents in a rule. As each antecedent is processed as a rule "head", the expense from some subset of the antecedents is incurred on all the other antecedents. In such cases, it might be more efficient to factor the rule into two separate rules using a new formula that represents

a partial state of the rule execution. This new formula can be asserted as a conclusion in one rule and tested as an antecedent in the second rule. For instance, consider the following rule that represents the inference principle that for all object x that have $pred_1$ true, and for all objects y and z such that x and y are related by rel_1 and y and z are related by rel_2, z has $pred_2$ true:

```
(rule sample
  ((pred1 ?x)
   (rel1 ?x ?y)
   (rel2 ?y ?z))
  (pred2 ?z))
```

This rule has three possible heads, one for each of the antecedents. We can assign an expense to each of the heads by counting the number of Ontic objects looped through when firing the rule. Now consider a factoring of this rule into two separate rules:

```
(rule sample-cut-1
  ((pred1 ?x)
   (rel1 ?x ?y))
  (predcut ?y))


(rule sample-cut-2
  ((predcut ?y)
   (rel2 ?y ?z))
  (pred2 ?z))
```

It it clear that due to this factoring, the head for $(rel_2\ ?y\ ?z)$ no longer needs to bind $?x$ by looping through the rel_1 facts on $?y$. The cost of the $(pred_{cut}\ ?y)$ head along with the cost of rule *sample-cut*-1 is the same as the cost of the $(pred_1\ ?x))$ and $(rel_1\ ?x\ ?y)$ heads from the original rule. While this factoring was performed by hand, a method for automatically identifying rule variables that can be used to factor a rule is described later in this section. A similar factoring principle applied specifically to orcfuns is also described in this section.

3.5.1 Antecedent Trees

As described before, Ontic's rule implementation involves determining an ordering of the antecedents that can bind all rule variables starting from the variables bound

by the head. In most cases, there are more than one antecedent orderings that can fully bind all rule variables. More importantly, there can be a broad variance in the expense incurred by these different antecedent orderings. For instance, if there are two partially bound antecedents that can both bind the same free variable, it is more efficient to promote the antecedent that checks a formula that is true of fewer Ontic objects. However, it is not always possible to decide which formula would be true of fewer objects at rule compilation time. For instance, consider the two Ontic relations `is` and `is-never`. The `is-never` formula is true of any pair of classes where one of them is the empty class `(fail)`. However, there are not many `is` facts where the right hand side (or the superclass) is `(fail)`. On the flip side, when considering the class `(a thing)`, there are a lot more `is` facts involving `(a thing)` compared to `is-never` facts. In effect, the antecedent ordering depends on the current partial rule substitution.

In order to take the current partial substitution into account, the linear rule antecedent ordering is extended to a tree structure where each antecedent can have one or more possible successor antecedents that could be checked next. The choice of which antecedent among them to check is made at runtime by wrapping this choice in a case structure. At runtime, the choice is made based on which among the antecedent choices involves the shortest lists to run down. It is to be noted that the choices are only drawn from the partially bound antecedents; any fully bound antecedent is promoted to be checked as soon as possible. The branching width and depth of the tree is controlled by two configurable parameters. Increasing the width and depth while decreasing rule execution expense, increases the size of each of the noticer procedures and the overall compile time for a rule. In practice, a branching width of 3 and a depth of 2 has resulted in significant savings in rule execution time.

3.5.2 Orcfun Continuations

One of the observations about the implementation of "compute-value" was that the resulting rules do repetitive work in checking the antecedents that infer the "compute!" conclusion. This is necessary since the body of the compute-value may refer to rule variables that were not present in the arguments to the compute!, but occur in the antecedents inferring the compute! conclusion. A rule that just contains the has-value antecedent and the compute-value body cannot ensure appropriate bindings for such rule variables. However, this raises an interesting point: we only need to ensure that we maintain state for the rule variables that are common to the antecedents of the compute-value and its body.

One way to maintain state is to assert a formula about the objects that need to be retained across the compute-value; such formulas can be termed "continuations". The compute-value body can then replace the antecedents of the compute-value by checking this one continuation formula instead. In order to determine the rule variables that need to be in the continuation formula, we first need to determine the shared rule variables between the antecedents and the body of the compute-value. A configurable parameter is provided that represents the minimum required difference between the number of rule variables from the antecedents of the compute-value and these shared variables. If this difference is small enough, it may not be advantageous to create a new continuation formula. Once the shared variables have been determined, a further optimization is performed to identify any rule variables that can functionally bind more than one shared variable. A variable $?x$ is said to functionally bind another rule variable $?y$, if there is a rule antecedent including these two variables such that if $?x$ is bound, then there is exactly one possible binding for $?y$. Functional binding essentially avoids the need for loops to check partially bound antecedents. If such a variable can be found, then it can be used in the continuation instead of the other rule variables that it can bind functionally. For instance, the variable corresponding to the thunk being computed can functionally bind the argu-

ments of the thunk's call (there can only be one production on a thunk object). The antecedents from the compute! rule that enable this functional binding have to then be checked in addition to the continuation formula. However, no heads are created for these additional antecedents since they are only present to bind the shared variables; they have already been checked in the rule concluding the compute!. The algorithm to determine the variables to include in the continuation termed "functional kernel variables", and the list of functional antecedents from the compute! rule to include is detailed below:

1. Let $V_{required}$ be the set of shared variables, V_{bound} be the set containing just the thunk variable and $ants$ the antecedents of the compute! rule.

2. Determine the set of functional edges in $ants$:

   (a) If an antecedent ant can bind a variable $depvar$ functionally from another variable var, then add the pair $(depvar, ant)$ to the $fk\_dependant\_vars$ of var.

   (b) If an antecedent ant can bind a variable $depvar$ functionally from a set of other variables $vars$, then add the antecedent to the set of hyper-edges.

3. Compute the transitive closure of the the $fk\_dependant\_vars$ relation. If all the variables except the dependant variable in a hyper-edge can be reached, then the dependant var $depvar$ is added to the transitive closure as well. Store this transitive closure in a new relation $fk\_derivable\_from$.

4. Determine the required variables that can be bound starting from V_{bound} and the antecedents required to bind them.

   (a) Search forward from each variable in V_{bound}, following the $fk\_dependant\_vars$ links. For every variable reached, set the fk_marked relation on that variable to the antecedent path followed to get there.

5. Determine the required variables that can be bound starting from the $V_{required}$ variables that have not yet been "marked" via fk_marked, following the same process as the previous step.

6. Determine if any of the variables of $ants$ can bind more than one of the $V_{required}$ variables that are not yet marked, using the $fk\_derivable\_from$ lists. If so, search forward from such a variable, setting the fk_marked antecedent paths and add any such variables to $V_{required}$.

7. Process the hyper edges:

   (a) For any hyper edge where the source variables are all either marked or required, set the dependant variable to be marked with the hyper edge's antecedent.

   (b) Search forward from the dependant variable, and add it to $V_{required}$ if any unmarked required variable can be reached.

   (c) If the dependant variable is added to $V_{required}$, also add all of the source variables in the hyper edge to $V_{required}$.

8. Finally, determine the set of continuation variables as all variables from $V_{required}$ that don't have an fk_marked value. The set of continuation antecedents are all the antecedents from the fk_marked paths for variables in $V_{required}$.

The algorithm above maintains the invariant that if a variable is the source of a binding of some variable in $V_{required}$, then it is added to $V_{required}$ as well. Any variables that have a fk_marked value can be derived from other variables via the antecedent path in fk_marked. It follows that only the $V_{required}$ variables that don't have a fk_marked value need to be included in the continuation formula. All other $V_{required}$ variables can be bound from either the V_{bound} or the continuation variables using their corresponding fk_marked antecedent paths. It is clear that the number of continuation variables is at most the number of variables in $V_{required}$ to begin with.

Any variable that is added to $V_{required}$ covers more than one existing $V_{required}$ variable, which are then no longer continuation variables.

3.5.3 Articulation Points

An example of rule factoring was illustrated previously where splitting a rule into two rules can reduce the overall inference expense. An approach to determining such potential rule factorings is briefly described here. By exploiting the principle of articulation points (or cut vertices) of a graph, a set of rule variables that can function as links between two separate rules (or blocks in the graph) can be determined. A cut vertex of a connected graph is a vertex removing which, causes the graph to be disconnected. In the case of a disconnected graph, a cut vertex is a vertex removing which, increases the number of connected components of the graph. Of course, this holds true for a connected graph as well, the disconnected components of the original connected graph are still connected, and hence the cut vertex when removed increases the number of connected components in a connected graph as well.

A graph formulation of a rule can be constructed by considering the rule variables as vertices and adding an undirected edge between every pair of variables in an antecedent. The standard algorithm for computing the articulation points can be run on this (unconnected) graph to identify the rule variables that are cut vertices of the rule graph. The connected components produced by removing a cut vertex identify sub-rules of the original rule that are still fully threaded. A continuation relation can be defined as before to be asserted about the cut vertex in one sub rule and tested in the other to link the sub rules together.

3.5.4 Declarative Rule Control

While the prior approaches described in this section are automatically applied to all rules, Ontic also provides the ability to declaratively control the implementation of certain antecedent formulas. An example usage of such declarative control was

described previously when identifying "functionally" derivable variables. For each Ontic relation, either one or both arguments can be specified to allow "functional-threading", i.e., that argument can be functionally derived from the other argument when it has been bound. Equality antecedents can be similarly functionally threaded by asserting certain categories to be "Herbrand", i.e., objects in these categories typically have just one production. In such cases, binding the variable on the left hand side of the antecedent can immediately bind the variables from the right hand side using the typically only production.

While functional threading is a means of preferring certain antecedents or binding choices, other declarative control causes certain antecedents or binding choices to be dispreferred. In particular the "ban-production-threading" control allows users to specify that certain argument positions for some constructor should not be used to determine bindings for the other argument positions. In general, if a variable at one of the argument positions in a production has been bound, the variables at the other argument position can be bound as well by going through the list of productions incident on the first object. In some cases, the number of such productions can be quite large making this inefficient. For instance, the number of productions incident on the built-in `(member-operator)` object can be quite large in a context that contains a large number of sets. Applications of the `(member-operator)` use the `iapply` constructor rather than the regular application constructor `apply` to distinguish applications to impredicative objects (sets, types). The `iapply` constructor can be set to disallow production threading at its first position, thus disallowing its second argument to be bound by looping through all possible productions incident on the first argument.

While this chapter has touched upon techniques for improving the efficiency of rules when handling pathological contexts with several thousands of Ontic objects; the next chapter will describe the primary inference methods that add new Ontic objects to the context and ways to control the number of objects created.

4. INTERNING INFERENCE

4.1 Introduction

Ontic rules by default implement non-interning inference, i.e. no new Ontic objects can be created when a rule fires. While this has some attractive properties including polynomial time decidability, it is also very restrictive. In particular there are two inference principles involving quantifiers, Skolemization and universal instantiation that are designed to introduce new expressions into the inferential context. Quantified facts are central to any natural development of substantial proofs as is evidenced by the increased expressiveness they provide over propositional logic.

Any inference mechanism that can introduce new expressions, needs to be carefully controlled to prevent unbounded runaway, especially in an interactive proof session. In Ontic such control is accomplished by using a layered interning approach; interning inference can only be triggered by Ontic objects with sufficient "focus", the resulting newly interned objects however decrease in "focus" and cannot lead to as much interning inference as the next higher layer. The notion of "focus" is implemented using control predicates, non-semantic facts that are asserted about Ontic objects. Different control predicates are used to correspond to each of the interning layers and checked as necessary in the various Ontic rules. The level of focus on an Ontic object depends on how far removed it is from the user's input; expressions directly typed by the user are heavily focused on. However, in addition to user input, there are other ways for an object to derive higher focus; in particular in service of certain inference invariants. For instance, applications of a defined symbol in proofs inside the body of that symbol's definition are assigned the highest level of focus. This enables the symbol's definition to be substituted in for all its occurrences in the proof. Similarly, for every defined operator, a generic application applying the operator to a new

variable from its domain is created. Since Ontic application is curried, for operators that accept more than one arguments, this process needs to be repeated until the operator has been applied to the required number of generic variables from the appropriate domain types. In order to accomplish this, the resulting generic application at each step is sufficiently focused on to enable it to come back through the rule. Generic applications, as the name suggests are important in generalization inference; for instance any typing information about a generic application can be generalized to applications of that operator to any subclass of its domain under certain conditions.

It should be pointed out that rather than rely on focus assigned to Ontic objects, it is individual expressions that need to be distinguished as having more focus than others. For instance, a user may mention the number `0` in their proof; however during the course of the proof `0` may have several productions assigned to it. It is clearly necessary to distinguish between the actual expression typed by the user from the other equivalent expressions interned during inference. However, the Ontic production grammar is not designed to enable such distinctions. Even if a particular production on `0` was marked as "special", it requires the same marking to be applied to a particular production on the objects that are arguments of this special production. At that point, it is impossible to distinguish between productions marked special for different surrounding expressions. This chapter first presents an alternative approach to managing user-typed expressions that preserves this distinction, enabling control of interning inference. The rest of the chapter describes the two primary interning inference principles in Ontic: Skolemization and universal quantifier instantiation.

4.2 Ontic Text Expressions

As discussed in the introduction, the Ontic production grammar managing class and formula expressions cannot maintain the distinction between expressions typed by the user and those interned during inference. This is in effect because congruence inference abstracts an equivalence class of expressions into an Ontic object. Instead

each expression needs to be treated as a separate Ontic object that cannot be equated; in effect a Herbrand interpretation.

This is accomplished by defining a new Ontic category,"text", whose objects cannot be equated. The Ontic production grammar is extended to this new category where a text object has exactly one production. A new relation is added to the Ontic formulas that relates a text object to its corresponding non-text Ontic object. It should be noted, that a particular Ontic object may have more than one text expression denoting it but a text expression has exactly one non-text counterpart. Each user typed expression in addition to being interned into the production grammar, is also translated into a corresponding text expression that is interned as well. The presence of a text version of an Ontic object is often required to allow interning of new expressions during inference. Various control predicates on text objects are used to implement a layered interning approach; text objects with the highest "focus" may enable rules to intern new text with lower "focus".

4.2.1 Managing Text Expressions

Each Ontic constructor has a text version with a "text-" prefix that enables a straightforward translation of user typed expressions to their text version. Each of these text constructors are applied to zero, one or two text Ontic objects. There is a separate set of constructors that are applied to the basic Ontic objects to produce their corresponding text versions; so for instance there is a `text-defined-sym` constructor that when applied to a defined symbol, produces its text counterpart (i.e. a text object related by the `text-of`) relation. There are several control predicates that can be asserted about a text object enabling a distinction between texts created by user typing, text created for the generic application of a user defined operator and finally texts interned via rules.

Most rules interning new text need to choose exactly one of the text counterparts of an Ontic object. Having a `text-of` antecedent binding the text object is undesirable

since that would cause each of the text counterparts to be chosen in turn, causing several text objects to be interned (recall that text expressions are Herbrand). In order to avoid this, Ontic maintains a hash table that contains the "smallest" known text counterpart for an Ontic object. The size of an Ontic object termed the "print-size" is computed as the minimum size among all its productions; each constructor is assumed to be of size one and the basic Ontic objects (including defined symbols) are assumed to be of zero size. When a new text counterpart for an object is discovered, the hash table is updated as necessary.

4.2.2 Role of Text Expressions

As described before, the primary role of text expressions is to help identify expressions to "focus" on. Interning rules in Ontic can now require text expressions to be present for a few "kernel" rule variables with varying control predicates asserted about them. For instance; a rule that interns the application of the "member" and "subset" operators to a set object requires sufficient marking on a text counterpart of the set object. Similarly, the presence of text expressions for the application of the inverse of an operator `f` to a class argument `x` and another class `c` that is known to be under this application, causes the application of `f` to `c` to be interned.

Text expressions enable the maintenance of certain invariants for different inference principles. Similar to the notion of user typed expressions having higher "focus", it is necessary to ensure that user typed instances of quantified facts are inferred true. Extending this principle to `some-such-that` classes, a user typed `is` fact between a singleton class and a `some-such-that` should result in the `some-such-that` formula being asserted true about the singleton object. While this latter example is straightforward to implement, detecting a user typed instance of a quantified fact requires expression matching. While the details of such matching and other quantifier instantiation principles are described later in this chapter, it should be pointed out that the Herbrand interpretation of text objects enables a straightforward matching of the

quantified formula's open body against user typed text. A substitution list is maintained to bind the variables from the open body and to enable typechecking against the quantifier types. Quantifier instantiation reasoning in Ontic is in general more powerful than that afforded by this simple matching principle; however having axioms and theorems in text form enables straightforward matching against user text.

4.3 Skolemization

Skolemization is the process of eliminating existential quantifiers by replacing the quantified variable with a new constant termed a "Skolem constant". While Ontic provides an "exists" quantifier, it is internally translated away to a quantifier-free existence assertion on the natural, corresponding some-such-that class. However, there are other implicit existential assertions inherent in taxonomic syntax that benefit from Skolemization.

Consider a context about colored graphs where we know the followings facts: `g` is a connected graph, `(is red (the color (a vertex g)))` and the axiom that `forall (v (a vertex g)) (= (the color v) (the color (a neighbor v)))`. So, at-least one of the vertices of g is red and for every vertex of g, it has the same color as its neighbor. It is obvious that these facts entail that every vertex of g is red. However, in order to employ the universal fact, we need to have a particular vertex of g to instantiate it on. Reconsidering the fact, `(is red (the color (a vertex g)))`, we can see that since `red` is a singleton object, there has to exist a vertex of `g` such that it has the color red. So, we can pick one such vertex as a new object in Ontic and instantiate the universal fact on that vertex. The desired fact is then obvious since the connectedness of the graph entails that every vertex of `g` is a neighbor of this newly picked vertex.

In general, the monotonicity of the `is` relation entails the existence of certain classes in a supertype if the subtype exists; Ontic Skolemization is based on this principle. The most obvious example is an applicative supertype; both the operator

being applied and the application argument need to exist. More generally, such "well-behaved" positions in an expression can be syntactically defined. The well-behaved positions are those that are both monotone and continuous and identify existential quantifiers that can be added to convert an `is` fact to an equivalent existential fact. For instance, the fact `(is red (the color (a vertex g)))` is equivalent to the existential fact `∃((v (a vertex g))) (is red (the color v))`. Similarly, consider another `is` fact, `(is Fred (except (a father (a policeman)) (a fireman)))`, i.e. `Fred` is someone who is a father of a policeman but is not a fireman. As before, there is an equivalent existential fact:
`∃((p (a policeman))) (is Fred (except (a father p) (a fireman)))`. For singleton classes of sufficient focus, any supertype that is of sufficient focus as well is Skolemized at its well-behaved positions, producing a new supertype for the singleton class. The presence of text expressions on classes of sufficient focus and a syntactic definition of well-behavedness, enables Ontic Skolemization to be carried out on text expressions. More significantly, Ontic Skolemization attempts to reuse constants instead of generating a new Skolem constant for each (implicit) existential quantifier.

4.3.1 Skolem constant reuse

The principle of Skolem constant reuse has been applied in the past to semantic tableaux proofs [6], where specializations of the Skolemization rule allow for Skolem constants or functions to be reused on distinct branches of the proof or in the case of formulas that are identical upto renaming of the free and bound variables. Ontic uses a semantic notion of reusability by encoding the "reason" for a constant's creation into its representation. Skolem constants are Ontic classes that have both a type and a "reason" that can represent concepts such as, the constant created as an argument to a certain function for a particular singleton target, or the constant created for an operator class in an applicative supertype of a particular target. Creation of a Skolem constant is then implemented as an Ontic orcfun computation on the class being Skolem-

ized and the "reason" for the Skolem constant's creation. If a Skolem constant already exists for the same reason and of the type that is the class being Skolemized, it can be reused. Checking for the existence of a Skolem constant of a type for a particular reason, is a semantic concept; infact this underlies the strength of Ontic Skolemization. Inference rules can be written to represent this semantic principle. For instance, if a singleton Ontic object is the argument to a function application and that application is known to be a supertype of a singleton class, then the application argument satisfies the obvious "reason". For instance, in the example about `Fred` above, if we also knew the facts, `(is Fred (the father Bob))` and `(is Bob (a policeman))`, then `Bob` is inferred to satisfy the reason "argument for the operator father, for the target Fred". No new Skolem constant needs to be created in this case; `Bob` can be used to Skolemize `(except (a father (a policeman)) (a fireman))`. Infact, if it was also known that `(is Fred (both (a father (a policeman)) (a painter)))`, then `Bob` can still be used to Skolemize `(both (a father (a policeman)) (a painter))` since the Skolem constant desired in both examples is semantically, "some policeman that Fred is the father of". In general, Skolem constant reuse reduces the number of distinct Ontic objects in a context, improving the efficiency of inference.

4.4 Universal Quantifier Instantiation

The primary interning Ontic inference is universal quantifier instantiation; henceforth simply referred to as quantifier instantiation. Both user-typed axioms and theorems resulting from proof attempts are typically universally quantified formulas. Quantifier instantiation inference needs to balance the efficiency of reasoning while ensuring that the expected instantiations are performed. Specifically, any theorems "relevant" to user-typed expressions or queries need to be instantiated. The various ways that Ontic determines such relevance while balancing the efficiency of inference is described later in this section.

While quantifier instantiation is a central inference principle in any proof assistant, its implementation varies based on the level of automation desired. Some proof assistants require the user to explicitly indicate the quantified fact to be instantiated, while others attempt to automatically determine the most relevant facts for the current proof attempt. Most proof assistants however, strike a balance in between these two extremes. Ontic attempts to automatically determine the most relevant theorems for instantiation by matching expressions in the current context to the universal facts in the context. The expressions selected for such matching are usually those with sufficiently high focus. As mentioned before, the presence of text expressions on both theorems and focus objects, enables straightforward expression matching. In addition to purely syntactic matching, certain facts may be deemed relevant for semantic reasons. For instance, a fact that can be instantiated to assert a new supertype for a particular class object may be identified as relevant even though there may not be an expression directly matching it.

The goal of matching expressions to relevant facts is to determine a substitution for the quantified variables for subsequent instantiation. This affords certain simplifications to the matching process. In general, rather than require the whole quantified formula to match an expression in the context, minimal subexpressions that contain all the quantified variables can be extracted as match targets. A resulting substitution from matching such subexpressions can bind all the quantified variables. Further simplifications are possible by realizing that the quantifier types can be dependent; i.e. an inner quantifier type can refer to variables from the outer quantifiers. In such cases, it suffices to only choose those subexpressions from the formula body that bind variables that do not occur in the dependant types. The dependant types can then be separately matched to bind the remaining variables. Any substitution derived from such matching needs to be typechecked to ensure that the objects being substituted in for the quantifier variables are of the corresponding quantifier types, and are singleton.

4.4.1 Quantifier Instantiation Implementation

In Ontic, quantified formulas are represented internally using the De Bruijn notation. The innermost quantifier corresponds to the De Bruijn index one, and the indices increase as we move to a surrounding quantifier. Using De Bruijn indices enables congruence closure to be applied to quantified formulas that are congruent upto variable renaming. A substitution can then be represented as a list; the substitution for a particular quantifier being the element at the corresponding De Bruijn index.

Each quantified theorem or axiom is processed to identify the Ontic class subexpressions termed "handles", that mention all the De Bruijn indices that are not covered by the dependant types. Substitutions are extracted from possible matches to the handles and used to instantiate the corresponding quantified fact. Not all quantified formulas have valid handles, for instance consider the formula: `∀((x (an integer-< 1)) (y (an integer-> 1))) (is x (an integer-< y))`. Neither `x` nor `(an integer-< y)` mention all the quantified variables. However, we still need to maintain the invariant that if the user types an instance of a quantified theorem, then that instance needs to be inferred true. In Ontic this is accomplished using a separate quantifier instantiation mechanism termed the "instance invariant". It should be noted that the formula above can be simplified to get rid of a quantifier. Due to the monotonicity of `is`, the following formula is equivalent: `∀((y (an integer-> 1))) (is (an integer-< 1) (an integer-< y))`. Each quantified theorem is simplified in this manner in a pre-processing step termed "quantifier elimination". Getting rid of excess quantifiers improves the efficiency of the inference process since there are fewer quantified variables to bind, thus reducing the complexity of the matching process. It should however be noted that the instance invariant is implemented on the original theorem; a user-typed instance of a theorem may not match the quantifier eliminated form. Quantifier elimination and substitution derivation from matching are described next.

Quantifier Elimination

Quantifier elimination in Ontic attempts to get rid of quantified variables that occur exactly once in a well-behaved position in either an inner quantifier's type or in certain class expressions in the formula body by replacing the variable with its type. The class expressions in a formula that are considered for such processing depend on the formula's constructor. For instance, only the left hand side of an `is` formula is considered, while both class expressions from an `is-never` formula are processed. Well-behaved positions in a class expression are defined syntactically for the different class expression constructors; for instance both the operator and argument positions in an `apply` expression are well-behaved but only the type in a `some-such-that` expression is well-behaved. Of course, since quantified facts are text expressions, all such processing occurs on text expressions. It is straightforward to see why the variable is required to occur exactly once. Consider the following formula:

```
(forall ((x (a number))
         (y (the sum 1 x))
         (z (the sum (the negation 1) x)))
  (is (the sum y z) (an even-number)))
```

Even though the variable `x` doesn't occur in the body, eliminating it from the types for `y` and `z` produces this untrue formula:

```
(forall ((y (the sum 1 (a number)))
         (z (the sum (the negation 1) (a number))))
  (is (the sum y z) (an even-number)))
```

Similarly,

∀`((x (a number))) (is (the sum x x) (an even-number))`

cannot be simplified to the unquantified formula

`(is (the sum (a number) (a number)) (an even-number))`.

Quantifier elimination from both the quantifier types and the formula body is conducted together by processing a list of class expressions that includes the quantifier types and the allowed class expressions from the body (for instance the left hand side of an `is` formula). Processing starts from the end where each quantifier is tested to determine if it occurs exactly once in a well-behaved position in the rest of the list. It

is important to note that eliminating quantifers reduces the De Bruijn index of outer quantifiers, both inner quantifier types and the body need to be updated in response to this change.

Formulas that are cardinality assertions about operator applications go through additional post-processing to derive possible classification facts about the operator. For instance, the fact:

∀((x (a number)) (y (a number))) (singleton (the foo x y))

can be inferred to entail:

```
(is foo
 (a function from number and number
             to (a foo (a number) (a number))))
```

if `(a number)` happens to be the domain type of `foo` and the partially curried operator `(a foo (a number))`.

Subsumption Instantiation

The primary quantifier instantiation principle in Ontic is "subsumption"; instantiations that infer additional information about an Ontic class via monotonicity of `is` are preferred. Both the handle extraction process and the matching process are designed with this principle in mind.

Handle Selection Handles are subexpressions of the quantified formula body that serve to determine potential instantiations. Class expressions in the current context are matched against handles to obtain a substitution for the quantifier variables. The handle selection is designed such that the resulting instantiated formula aids in taxonomic reasoning about these matching class expressions. Handle selection passes through conjunctions and only considers the implicand in an implication formula. Unary predicates such as the cardinality assertions have exactly one possible handle, the argument to the formula. The left hand side of an `is` formula is considered a potential handle, while `=` formulas are processed as a conjunction of two `is` formulas. Similarly, handles can be drawn for both arguments to an `is-never` formula.

Since matches to handles need to determine a substitution for all quantified variables, handles are required to mention all quantified variables that do not occur in any dependant types.

Once the handles for a quantified formula are obtained, the subsumption instantiation process is triggered by asserting a formula for each handle on the quantifier type list, the handle and the formula body. The next step in the instantiation process is to determine a substitution via matching.

Substitution Derivation The possible substitutions for a quantified formula are derived in two phases. First the type list is processed from the outside in, to obtain partial substitution lists that bind the De Bruijn indices in the dependant types. Processing the type list first enables the dependant types to be instantiated, allowing for substitutions to be typechecked during the matching process. Once all the dependant types have been instantiated, the partial substitution is completed by processing the handle and potential matching class expressions. The substitution extraction is itself conducted in two phases. First potential matches to a handle (or dependant type containing De Bruijn numbers) are determined using two separate matching principles; equality matching and subsumption matching. Next, the matches to a handle are processed to extract substitutions for the De Bruijn indices in the handle. The reason a substitution needs to be extracted separately after the matching process is because the matching process does not keep track of the substitutions to ensure that repeated occurrences of the same De Bruijn index are matched to the same singleton Ontic object. Also, the matching process does not carry the quantifier type list around to typecheck objects that are assumed to potentially match the De Bruijn indices. Once potential matches to a handle have been determined by a cheaper matching process, the actual substitution extraction process can ensure that these additional criteria are satisfied. It bears mentioning that while both matching and substitution extraction is organized around text expressions, substitutions themselves are lists of

Ontic class objects. Incorporating text expressions into the matching process helps restrict quantifier instantiation to the focus objects.

Equality Matching Equality matching can be defined recursively as follows:

- A De Bruijn index matches any text expression on a singleton class expression as well as itself.
- A text expression with no De Bruijn index subexpressions matches any text expression of the non-text counterpart.
- A text expression (*constructor* $arg1$) matches text expression (*constructor* $arg2$) if $arg1$ equality matches $arg2$.
- A text expression (*constructor* $arg1$ $arg2$) matches text expression (*constructor* $arg3$ $arg4$) if $arg1$ matches $arg3$ and $arg2$ matches $arg4$.
- If text expression $t1$ matches $t2$ and $t2$ is a text on a defined symbol's definition, then $t1$ matches the text version of the defined symbol.
- For a text intersection expression ($text - intersection$ $t1$ $t2$), if $t2$ does not contain any De Bruijn indices and $e1$ matches $t1$ and the non-text counterpart of $e1$ is known to be a subclass of the non-text counterpart of $t2$, then $e1$ matches ($text - intersection$ $t1$ $t2$). Similarly, for the case where $t1$ does not have any De Bruijn indices.

Equality matching is implemented with a binary matchability relation between a handle text and a matching text. An Ontic rule for inferring this relation can be defined based on the rules above for concluding matchability. Recall that substitution derivation is implemented as a sequence of updates to a partial substitution (starting with an empty substitution) based on possible matches to the dependant types in the type list and the extracted handle. An Ontic orcfun is defined to operate on these matched pairs (of handle and matching text), computing all possible extensions of a

partial substitution that are derivable from these matches. A possible return value of this orcfun is an extension of the incoming partial substitution that binds all the De Bruijn indices in the handle text.

The substitution extraction process can be defined recursively as a computation on the handle text *pattern*, a matching text identified by the matchability relation *exp* and a partial substitution *subst* as follows:

- If *pattern* does not contain any De Bruijn numbers, then return *subst* as long as *exp* is an alternate text on the non-text counterpart of *pattern*.
- If *pattern* is a text version of a De Bruijn number, then return an updated substitution using these rules as long as *exp* is a text on a singleton class and that class is a subclass of the quantifier type at the De Bruijn index corresponding to *pattern* :
  - If *subst* does not have a value at the De Bruijn index corresponding to *pattern* then return an updated substitution where the value at De Bruijn index is now the singleton class.
  - If there is already a value at the De Bruijn index corresponding to *pattern* in *subst*, then return *subst* if the value at that index is equal to the singleton class.
- If the defining production of *pattern* is $(constructor_p\ arg_{p1})$ and the defining production of *exp* is $(constructor_e\ arg_{e1})$, then return the result of recursing on arg_{p1}, arg_{e1} and *subst* as long as $constructor_p$ and $constructor_e$ are the same.
- If the defining production of *pattern* is $(constructor_p\ arg_{p1}\ arg_{p2})$ and the defining production of *exp* is $(constructor_e\ arg_{e1}\ arg_{e2})$, then if $constructor_p$ and $constructor_e$ are equal, let $subst_1$ be the result of a recursive computation on arg_{p1}, arg_{e1} and *subst*. Return the result of a recursive computation on arg_{p2}, arg_{e2} and $subst_1$. In the special case where the constructor is `lambda`, we need to account for the additional quantifier from the lambda's type. $subst_1$

is augmented with a new value `(self-binding)` at the front of the list corresponding to the De Bruijn index one. If the substitution at a De Bruijn index is `(self-binding)`, then any computation of this orcfun on a handle text corresponding to that De Bruijn index can only succeed if the matching text is the same De Bruijn number text as well.

In order to take advantage of cached results (via orcfun has-value facts) on common subexpressions, the partial substitution is first limited to only contain the De Bruijn indices that are present in the handle text. This gets rid of the variance in the values at the other indices that are irrelevant to this matching request.

Subsumption Matching While equality matching is based on a matchability relation between two texts, subsumption matching is based on a matchability relation between a text and a class object. A text handle is subsume matchable to a class object as long as there exists some substitution which when applied to the non-text counterpart of the text handle, produces a supertype of the class. In conjunction with the way handle selection is designed, any resulting instantiation of the fact corresponding to this handle provides additional taxonomic knowledge about the subsume matchable class. As with equality matching, the subsume matchability relation can be defined recursively as follows:

- If a handle *pattern* is subsume matchable to class exp_1, then it is also subsume matchable to any subclass exp_2 of exp_1.
- If a handle *pattern* is equality matchable to some text *exp*, then *pattern* is subsume matchable to the non-text counterpart of *exp*.
- If $p1$ is subsume matchable to $e1$ and $p2$ is subsume matchable to $e2$, then the application `(text-apply p1 p2)` is subsume matchable to `(apply e1 e2)` as long as $e1$ and $e2$ are singleton.
- If $p1$ is subsume matchable to $e1$, then the text expression corresponding to the application of a unary impredictive operator's text to $p1$ is subsume matchable

to the class expression corresponding to the application of the unary impredicative operator to $e1$.

- If $p1$ is subsume matchable to $e1$ and $p2$ is subsume matchable to $e2$, then the text expression corresponding to the application of a binary impredicative operator to $p1$ and $p2$ is subsume matchable to the class expression corresponding to the application of the binary impredicative operator to $e1$ and $e2$.

- If $p1$ and $p2$ are subsume matchable to e, then the text `(text-intersection p1 p2)` is subsume matchable to e.

- If $p1$ is subsume matchable to $e1$ and $p2$ is equality matchable to the text $e2_{text}$, then `(text-except p1 p2)` is subsume matchable to the class `(except e1 e2)` where $e2$ is the non-text counterpart of $e2_{text}$.

An inference rule can be written based on these rules to conclude the subsume matchability relation between a text and Ontic class. The classes under consideration in this rule are restricted by requiring a separate control predicate to be known about them. The control predicate essentially tests for the presence of a text expression with sufficient focus on these classes.

Based on the subsume matchability relation, a substitution can be derived from a triple of handle text *pattern*, subsume matchable class *exp* and partial substitution *subst* using the rules below:

- If *pattern* does not contain any De Bruijn numbers, then return *subst* as long as *exp* is a subclass of the non-text counterpart of *pattern*.

- If *pattern* is a text version of a De Bruijn number, then return an updated substitution using these rules as long as *exp* is a singleton class and also a subclass of the quantifier type at the De Bruijn index corresponding to *pattern* :

  - If *subst* does not have a value at the De Bruijn index corresponding to *pattern* then return an updated substitution where the value at De Bruijn index is now *exp*.
  - If there is already a value at the De Bruijn index corresponding to *pattern* in *subst*, then return *subst* if the value at that index is equal to the *exp*.

- If the defining production of *pattern* is (*text-apply* arg_{p1} arg_{p2}), then for every supertype of *exp* of the form (*apply* arg_{e1} arg_{e2}) such that arg_{p1} is subsume matchable to arg_{e1} and arg_{p2} is subsume matchable to arg_{e2} compute an updated substitution as follows:
  - Let $subst_1$ be the result of recursive computation on arg_{p1}, arg_{e1} and *subst*.
  - Return the result of recursive computation on arg_{p2}, arg_{e2} and $subst_1$.
- Repeat the same process for the case where the defining production of *pattern* is the application of an impredicative operator.
- If the defining production of *pattern* is (*text-intersection* p_1 p_2), then first recurse on p_1, *exp* and *subst* and use the resulting substitution to recurse on p_2 and *exp* as long as *exp* is subsume matchable to both p_1 and p_2.
- If the defining production of *pattern* is (*text-except* p_1 p_2), then for each supertype of *exp* of the form (*except* e_1 e_2) such that e_1 is subsume matchable to p_1 and e_2 is equality matchable to p_2, first recurse on p_1 and e_1. The resulting substitution is then used to compute a substitution by equality matching p_2 and e_2.

Just like equality matching, the partial substitution at each recursion is limited to only the De Bruijn indices in the pattern being recursed on.

Both equality and subsumption matching maintain an aggressiveness relation about the orcfun thunk being computed and the returned substitution. This relation is asserted whenever the matching text or class is of sufficiently high focus. The

purpose of this relation is to instruct the quantifier instantiation process to generate text from the instantiation.

Once the dependant types and the handle text have been matched to derive a complete substitution that binds all the De Bruijn indices, the next step is to instantiate the corresponding quantified fact.

Substitution Application

Applying a substitution to a quantified formula body is straightforward, each De Bruijn number in the body is replaced with the substitution range object at the corresponding index. The resulting formula is then interned and asserted true. However, the layered interning approach built into Ontic requires text to be created as a result of certain instantiations. The idea is that for objects of sufficiently high focus, theorems matching them should produce text to enable inference chaining via subsequent matching of the resulting text and instantiation of other theorems. As mentioned previously, a substitution resulting from a match can be asserted as aggressive, causing text to be created using the smallest text on each substitution object.

Even if the substitution was not marked aggressive, certain other conditions can cause new text to be created during instantiation. For instance, if the instantiation produces a universally quantified formula, then text is created for that formula to enable handles to be derived from it. This is especially relevant in the case of `some-such-that` classes where the formula is universally quantified. If an object is known under this class, then the universally quantified formula resulting from instantiating it on this object needs to be asserted and have text on it to implement the semantics of the `some-such-that` class.

In some other cases, text is simply produced during instantiation if it can improve on some previous text on the class subexpressions in the result. The potential print size of any resulting text is maintained during the instantiation process. If the print

size of the current smallest text on any class expression is larger than the estimated print size of the new text, then the resulting text is created as well.

The Instance Invariant

As mentioned previously, we require some guarantees of the quantifier instantiation process. In particular, a user typed instance of an quantified theorem needs to be asserted true; the instance invariant implements this guarantee. The instance invariant follows the general principle of equality matching, except that instead of a single handle for the theorem, there is now a set of handles corresponding to all the top-level class expressions from the formula body. If any top-level class expression is just a De Bruijn number, then the closest formula surrounding it is used as a handle.

As before, substitutions are extracted by processing the type list and then the list of handles from the formula body. However, equality matching (but not subsume matching) is used for each of the types and the handles; the rationale being that we are attempting to identify exact matches to user typed instances of this theorem.

Monadic Lemmas

Some universally quantified theorems have just one quantifier, for instance: `∀((x (an integer))) (= x (the sum 0 x))`. There is a trade-off in such cases, since such theorems are simple enough for the user to expect that they are instantiated on every possible singleton class under the quantifier type. However, this can quickly turn expensive in the presence of other similar thereoms with a single quantifier.

Thus, the instantiation of such theorems is heavily controlled, only allowing singleton classes from user typed show statements, variables introduced by let-be statements and variables on which induction is being conducted. Certain kinds of monadic lemmas are allowed more leeway. If the formula body is an `is` or `=` fact such that one side does not mention the quantified variable; then other singleton classes of slightly lower focus are also allowed as instantiation targets.

5. PROOF AUTOMATION

Most proof assistants include support for proof tactics that combine several low-level proof steps or try alternate proof strategies in a backtrackable fashion. Proof tactics foster shorter proofs by enabling proof reuse, thus allowing users to focus on the high-level details of the proof. For instance, the *auto* proof tactic in Coq attempts to prove the current goal by repeated use of the $reflexivity$, *intros* and *apply* tactics. The hypotheses in the current context of a proof are checked for applicability to the current goal and if so, automatically applied. This process is then repeated with the resulting subgoals. Additional lemmas can be added to the hypotheses set of *auto* by using the Coq hints database. Similarly, definitions can be set to unroll automatically. In order to control the expense of this process, *auto* has a fixed depth of nesting proof attempts. The Coq tactics language allows users to define their own tactics that can generate proofs based on syntactic matching of the current proof context and goal.

Proof tactics in Ontic follow a similar principle; syntactic (and some semantic) matching of the current goal is used to select from a set of built-in tactics to be automatically employed. The Ontic proof macro language can also be used to define new tactics that can be used in proofs. The Ontic `show` proof form can be thought of as performing a combination of the *auto* and other proof tactics from Coq. When a `show` proof is evaluated, Ontic first uses the inference engine to attempt to verify the goal by refutation. This involves executing the inference rules in a new context that has been augmented with the goal text and the refutation assertion. Any quantified theorems in the current context and the surrounding global context (resulting from proofs that have been evaluated so far and libraries that have been included) are automatically applied via the quantifier instantiation process described previously. Since beta reduction is implemented using quantifiers, applications of defined symbols in the goal formula are automatically beta reduced as well. If Ontic fails to verify the

goal, proof tactics (collectively termed "second-try") applicable to the current goal are automatically employed in a backtrackable manner. This may generate new proof goals and nested second-try attempts on those goals, which are controlled with a configurable nesting depth parameter. If none of the proof tactics succeed in proving the original goal, the user is returned to the context of the original `show` proof, requiring the user to provide their own proof body.

In addition to automatically employing proof tactics based on the current goal, Ontic also maintains a list of formulas that can be used to conduct automatic case analysis. Rather than conduct the obvious depth-bounded exponential case analysis, Ontic case analysis is designed to be done in quadratic time in the number of cases. The case formulas are split into priority and non-priority cases, with priority cases being employed in case analysis across second-try proof attempts. Case analysis is usually performed before the `show` proof form falls back to second-try tactics or when no specific second-try tactics exist for the current goal.

While proof tactics and case analysis are intended for the current goal, an orthogonal but useful tactic is automatically applied to every user typed proof statement. This typechecking tactic processes user typed expressions to ensure that every application is well-typed; i.e. each argument to an operator is of the right domain type. In addition to the primary typechecking task, the tactic can also be configured to verify that the expression being typechecked has a particular supertype or certain cardinality properties. For instance, when typechecking a user typed `show` statement where the goal is an `is`, `there-exists` or `at-most-one` formula, the typechecking tactic can be instructed to check for these properties when processing the goal formula. Similarly, when processing the definition of an operator, the typechecker attempts to infer cardinality properties of the operator's application to generic arguments from the domain types. Typechecking recursive definitions in particular can provide useful theorems about the newly defined symbol. For instance, consider the following recursive definition of whole numbers:

```
(define (a number)
  (either 0 (the sum (a number) 1)))
```

Typechecking this definition requires Ontic to verify that `(a number)` is in the domain of the `sum` operator, the built-in type `(an integer)`. Ontic automatically adds the resulting theorem that `(is (a number) (an integer))` to the lemma library. However, the primary purpose of the typechecker is still user input validation. Unless a typechecking goal is set to be optional, any typechecking failures are reported to the user, requiring them to add a proof of the fact before proceeding.

The rest of this chapter describes the various Ontic proof tactics applied to a `show` goal, case selection and the implementation of Ontic case analysis, and the typechecker.

5.1 Proof Tactics

The second-try proof tactics are implemented using a single proof form `second-try` that takes a formula argument. However, several proof macros can be defined for this form for the various formula constructors. Syntactic matching on the current goal is used to determine which of the second-try macros to employ. For goal formulas that do not have specific second-try macros, a catch-all macro is defined that simply employs case analysis to try to verify the goal. Each of the second-try proof tactics are described next.

second-try-is The second-try tactic for `is` goals employs semantic matching on the left and right hand side to determine a proof strategy to employ. For instance, if the left hand side matches an `either` expression, then the `is` fact is proven separately for each component of the `either`. Similarly, if the right hand side matches an `if` expression, then the `is` fact is proven by first assuming that the `if` condition is true and using the then branch as the right hand side and similarly assuming that the `if` condition is false, with the else branch being the new right hand side. However, before the semantic matching can be performed on the left or right hand side, a variable is created for the left hand side and given sufficient focus to enable quantifier inference on it. While the matching is still based on the original left hand side class,

the second-try proofs instead attempt to prove that the variable is under the right hand side class. In the special case where the left hand side class is an application of a recursive operator, an induction proof is first attempted to prove the goal. Semantic matching of an expression involves two pre-processing steps, first if the expression is an application of a defined symbol to singleton arguments, then beta reduction is carried out on the application. If the expression is a defined symbol, then it is replaced with its definition. Next, the resulting expression is simplified from the inside out using the following rules:

- When the truth of a condition is known for `if` or `when` expressions, the expression is replaced with the appropriate branch (in the case of `if`), and either the body or the empty class `(fail)` (in the case of `when`).
- Empty components of `either` expressions are removed.
- Chains of the impredicative `member` operator and its inverse are simplified. For instance, the expression `(a member (the-set C))` is simplified to `C` and similarly, `(the-set (a member S))` is simplified to `S`.
- Applications of `lambda` expressions to singleton arguments of the appropriate domain type are beta reduced.
- Applications of `car` and `cdr` to cons-cell classes are simplified to the appropriate element of the cons pair.
- Applications of the `domain` operator to operator class or lambda expressions are simplified to the domain type.

In order to avoid repeated definition expansion on nested second-try attempts, the resulting expression is tagged and the pre-processing step is protected against processing tagged expressions. Separate proof macros are used for processing the left and right hand sides. Processing the right hand side is relegated to occur after the left hand side has been processed by adding it to the body of the left hand

side processing proof macro. The proof macro processing the left hand side employs syntactic matching using the following rules:

- If the left hand side expression matches `(intersection X Y)`, the proof recurses to perform syntactic matching on `X` and then syntactic matching on `Y`.
- If the left hand side expression matches an `if` expression, two proofs are conducted, first assuming the if condition is true and recursing to process the then branch and second assuming the condition false and recursing to process the else branch.
- If the left hand side expression matches a `when` expression, the condition is assumed and the body is processed.
- If the left hand side matches an `either` expression, the proof recurses to process the first component of the either, followed by the second component (which could itself be an `either` expression).
- If the left hand side does not match any of these branches, the incoming body of the second-try proof is evaluated.

It should be noted that the incoming body when processing the left hand side is a proof that processes the right hand side expression. The right hand side expression is processed using the following rules (recall that the incoming left hand side is now a singleton variable):

- If the right hand side expression is an `if`, we conduct two proofs, one assuming that the `if` condition is true and proving that the left hand side is under the then branch and then assuming that the condition is false and proving that the left hand side is under the else branch.
- If the right hand side is a `when` expression, we need to prove that the condition is true, following which we need to prove that the left hand side is under the body of the `when`.

- If the right hand side is an `either` expression, we try to first prove that that left hand side is under the first component of the either. If that fails, we try to prove that it is under the second component of the either.

- If the right hand side is an `intersection`, we prove that the left hand side is under each of the components of the intersection.

- If the right hand side matches `(some-such-that (var type) formula)`, we first prove that the left hand side is under `type` and then prove the `formula` instantiated on the left hand side.

- If the right hand side is an operator class expression, we first prove that the left hand side has the same domain type as the operator class. Next, we create a generic application of the left hand side to a member of its domain and prove that the application is under the range class of the right hand side. Finally, if the operator class is functional, we prove the appropriate cardinality property about the generic application.

In contrast to the second-try-is macro, the tactics for the other formula types are fairly obvious and straightforward. A brief description of each is given below.

second-try-= Since an equality can be proven by two `is` facts, this macro simply transforms into two `show` proofs for the `is` facts.

second-try-forall A universally quantified fact is typically proven by introducing variables for the quantifiers and proving the formula body about those variables and then generalizing the result. Just like the *intros* tactic in Coq, the second-try-forall tactic introduces the necessary variables and a `show` proof for the formula body on those variables.

second-try-is-never In order to prove that two classes are disjoint, the second-try-is-never tactic employs proof by refutation. A new variable is introduced that is

under the two classes that need to be proven disjoint. A contradiction can then be sought in this new context by trying to show `(false)`.

second-try-at-most-one In order to prove that a class has at-most one member, we again use refutation. Two distinct members of the class are introduced and a contradiction is sought.

second-try-singleton Based on the singleton property being shorthand for `there-exists` and `at-most-one`, the second-try-singleton tactic conducts two proofs to prove that the class exists and has at-most-one member.

second-try-and When proving a conjuction formula, we simply prove each conjunct individually. The only important point here is to ensure that the second-try nesting depth does not prevent proving conjuctions with more than 5 conjuncts. This is accomplished by directly recursing to the `second-try` proof macro rather than the `show` macro to avoid increasing the nesting depth.

second-try-implies An implication can be proved by assuming the implicant (using the `suppose`) proof form and proving the implicand.

As mentioned before, if the goal formula does not syntactically match any of these formula constructors handled by second-try, the `show` proof simply falls back to case analysis as a final attempt at automatically proving the goal. The implementation of automatic case analysis in Ontic is described next.

5.2 Automatic Case Analysis

Ontic case analysis is organized around case formulas that are automatically identified in a proof context. Ontic rules are used to ensure that the necessary control predicates are tested to restrict the number of case formulas and to add these formulas to a cases list. Typical case analysis involves selecting one of the case formulas

and attempting to prove the goal under the two assumptions: the formula is true or false. If the goal cannot be proven in a branch, another case formula is chosen and the process is repeated until no case formulas remain or some predefined depth bound is reached. At any such step, inference may add new case formulas to the list. It is clear that this approach is exponential in the number of cases (or the predefined tree depth).

While a tight bound on the tree depth alleviates this issue, it raises the question of how to prioritize among the many case formulas to choose from. In addition, the choice of an appropriate depth bound is non-trivial. Ontic case analysis takes the position that rather than restrict the depth of the tree, only the most promising cases are considered. An additional goal is to restrict the branching width of the tree. Also, cases from failed case analysis attempts are marked to avoid repetition across second-try proof attempts.

The process of case formula selection and the implementation of case analysis is described next.

5.2.1 Case Formula Selection

The most obvious case splits originate from conditionals in class expressions. For instance, the condition in an `if` or `when` class expression can signal an important case split that needs to be performed. Another source of case formulas is `either` expressions. For instance, proofs about numbers can often benefit from considering whether the number in question is either even or odd. The supertypes of objects in focus are processed (using inference rules) to identify `if`, `when` and `either` class expressions that can provide useful case splits. In the case of an `either` supertype, a new case formula is constructed as an `is` relation between the focus object and one of the components of the `either` expression. This represents a case split between which of the `either` expression's components the focus object is under. It should be noted that such focus objects are necessarily singleton; for non-singleton classes there may

not be one particular component of the `either` that the class falls entirely under. In addition to these three conditionals, implicants from instantiation of quantified formulas are added to the cases list as well.

Some of these cases are marked as "priority" cases based on whether the focus object in question has the highest focus (i.e. typed by the user). In the case of implicants from quantifier instantiation, a substitution range of the highest focus causes the implicant to be added to the priority case list. The primary distinction between priority and non-priority cases is that priority cases from failed case analyses can be retried across second-try proof attempts.

The cases list is slightly more complex than just a list of formulas, instead it consists of a list of structure objects where each object contains the case formula, and a number indicating the number of times a case has been tried. The number of times a case has been tried is used to prioritize unused cases before retrying a previously considered case formula.

5.2.2 Case Analysis Implementation

Case analysis is implemented using a proof macro that is always evaluated in a refutation context, i.e. one where the current goal has been assumed false. The goal of case analysis is to then derive a contradiction on each of its branches. In order to control the branching width of the case analysis tree, a case formula is selected during case analysis only if a contradiction is immediately inferrable under the assumption that the formula is true. The only remaining task is to then derive a contradiction under the assumption that the formula is false, this results in a linear case tree. When a formula is selected for addition to the cases list, both the formula and its negation are added to the cases list, so it is sufficient to just check for a contradiction under the formula assumption.

If a contradiction is not immediately inferrable, the formula is considered as unpromising and dispreferred in subsequent searches (for the same goal) for a useful case

split. The reason such formulas need to be retained (and not immediately discarded) even if they don't immediately derive a contradiction is because a conjunction of case formulas may still entail false even if they don't individually. Once a formula is selected as a useful case split, a contradiction is sought under the assumption that the formula is false. If the contradiction is not immediate, a nested case analysis proof is generated. This new proof seeks a new promising case formula from the cases list with preference given to previously untried formulas. If no such formula can be found, case analysis has failed and the non-priority formulas on the cases list are discarded by disallowing the formulas from being tried again in the current proof context. Such formulas are termed as "committed" cases. It should be noted that the priority formulas are not always retried either; they are typically reconsidered in a last ditch proof attempt. If no useful case split can be discovered, the priority cases are committed as well before reporting a failure to the user. However, priority cases are prioritized before non-priority cases when searching for a promising case split.

In order to reduce the number of patently unpromising cases, additional bookkeeping is performed whenever a case formula assumption does not derive an immediate contradiction. Any case formulas that are entailed in the context of this assumption are discarded as well since they are in essence weaker than the current assumption and cannot derive a contradiction. While going down the list of case formulas to determine such unpromising formulas during each case attempt makes this process quadratic in the number of case formulas, it avoid the expense of going to the context where the formula is assumed to check for a contradiction when it is known to fail.

Tables 5.1 and 5.2 presents the pesudo-code for Ontic case analysis.

When the case analysis proof macro is evaluated, the context is extended with a $SELECT-CASE$ extender that takes an optional argument indicating whether to retry all priority cases and commit them if no promising case can be found. This extender in turn performs some initial filtering before invoking the $FIND-NEW-CASE$ function on a list of case structure objects. Pesudo-code for the $SELECT-CASE$ extender is presented below.

Table 5.1.: Pseudo-code for case selection from a list of cases.

```
FIND-NEW-CASE(cases)
//  cases is a list of case structure objects.
//  Each case structure object consists of a case formula,
    and a tried level indicating the maximum nesting depth the
    case has been tried at before.
//  Each case starts at a tried level of zero on addition to the cases list.
//  *case-selection-level* is the current case analysis depth, starting at one.
//  Return a case formula (if one exists) from cases such that a contradiction
    is inferred when assuming the formula.

1.  Delete any members of cases whose formulas are known true or false.
2.  Delete any members of cases that have been committed,
    i.e., those whose tried level is set to nil.
3.  Sort cases in increasing order of tried level.
4.  Repeat for each member case of cases :
5.     Let formula be the case formula of case
6.     If assuming formula true leads to a contradiction
7.        Return the negation of formula as the selected case formula.
8.     Else
9.        For each case case_other in cases,
10.          If the case formula for case_other is known true
    in the current context (with the formula assumption),
11.             Set the tried level for case_other to the maximum of its previous
    tried level and the current *case-selection-level*.
12.       Remove the formula assumption from the context.
```

Table 5.2.: Pseudo-code for case selection context extender.

```
SELECT-CASE(commit?)
//  commit? is an optional argument specifying whether to retry all priority cases
    and commit them all if no useful case can be found.
//  *priority-cases-list* is the list of priority cases
    *cases-list* is the list of non-priority cases.
//  sets a global *forced-case* variable to the newly selected case formula.

1.   If commit? is true
2.      Set priority-cases to *priority-cases-list*.
3.   Else
4.      Set priority-cases to *priority-cases-list* cases with tried level is zero,
    i.e. only the newly added priority cases.
5.   Set *forced-case* = FIND-NEW-CASE(priority-cases).
6.   If *forced-case* is null,
7.      Set *forced-case* = FIND-NEW-CASE(*cases-list*).
8.      If *forced-case* is null,
9.         If commit? is true,
10.           Set the tried level on every case in *priority-cases-list* to nil.
11.        Set the tried level on every case in *cases-list* to nil.
```

5.3 Typechecking

While the primary role of proof tactics and case analysis is the automation of certain proofs, typechecking is mainly intended to validate user typed expressions, informing the user of unsatisfied proof obligations. Due to the richness of the Ontic class expression syntax, it is possible for users to type expressions that while valid, may not accurately represent the semantics they have in mind. For instance, Ontic rules for range reasoning on operator applications simply conclude that an application of an operator to a member of its domain is under the range class. If the operator is incorrectly applied to an argument that isn't in its domain, no error is generated. This is still a valid class expression, but will just not be known to exist or be in the range class. In order to avoid such inadvertent mistakes, the typecheck constrains the set of expressions that the user can type to be well-typed. Every argument to an operator has to be in the domain class of that operator. This naturally extends to applications containing more than one argument; Currying is used to ensure that each argument is of the right domain type for the operator corresponding to the partial application upto that point.

The Ontic typechecker processes each user typed proof form, stepping through the expression and typechecking each application. The Ontic Emacs interface is designed to place the cursor on the exact expression that fails typechecking when an error is returned. In addition to verifying domain typing for application arguments, typechecking also checks that the arguments to impredicative arguments are of the desired Ontic category; for instance an argument to `member` needs to either be a set or type object. The typechecker can also be charged with verifying certain cardinality or typing properties on class expressions. In fact, verifying domain typing on application arguments is a special case of verifying that a class expression has a certain type. In addition, these checks can be designated as optional or required.

5.3.1 Definition Typechecking

A special case of user proof forms that are typechecked are Ontic definitions. Operator definitions are processed by introducing generic variables for the arguments and the body of the definition (instantiated on these generic variables) is typechecked after the argument types have themselves been typechecked. The typechecker is also asked to optionally verify cardinality properties of the resulting body. Such theorems are useful in classifying the operator being defined into an appropriate operator class (partial or total operator, function). Similarly, any typing theorems about the body are universally quantified and added to the base context. An Ontic definition is accepted and interned to the defined symbol only if the typechecker succeeds on it. In contrast, when typechecking a recursive definition, the definition needs to be processed beforehand since the body mentions the symbol being defined. However, the typechecker still needs to succeed on the definition before any other proofs can be written. When the typechecker fails on a definition, any supporting proofs need to be evaluated before the definition can be processed again. In the case of non-recursive definitions, such proofs can be added before the definition and evaluated before the definition is re-evaluated. In the case of recursive definitions, such proofs may need to mention the newly defined symbol. As a result, such proofs are added to the body of the recursive definition. Proofs in a recursive definition body are evaluated after the definition is processed, but before the typechecker runs on it.

Just like regular Ontic definitions, Ontic structure definitions are typechecked by first typechecking the types of the structure slots and then typechecking the optional filtering formula in the structure definition body.

5.3.2 Typechecker Implementation

The typechecker is implemented using an Ontic proof macro that uses a special form of the `show` proof form to verify any typing or cardinality claims. Theorems resulting from these verifications are then added to the context. In contrast to the

regular `show` form, only a single second-try proof attempt is made with no nested second-try attempts. This ensures that the typechecker can efficiently process the user typed expression, falling back on the user to provide any necessary proofs of typing facts. The proof form takes a single Ontic expression to be typechecked as an argument and optional cardinality and typing requirements. The typecheck proof form can then be inserted into the proof macro for the various external proof forms like `show`, `suppose` and `suppose-there-is`.

The argument to the typechecker proof form is syntactically processed to step through the various constructors, setting up the necessary context until an application expression is reached. The application is then processed to ensure that every argument is in the correct domain type of the operator applied to it. Similarly, when an application of an impredicative operator is reached, the arguments are typechecked recursively but also verified to be of the appropriate Ontic category. In the case of the `subtype` and operator class constructors such as `total-operator`, `partial-function`, etc., the argument is required to be `type-class`. Similarly, in the case of the `subset` operator, the argument is required to be `set-class`. The argument to the `member` operator can either be `type-class` or `set-class`. Since the `cons` operator allows pairs to be built from arbitrary classes, no specific category is required of its arguments.

Verification of the cardinality and typing claims on the top level expression are relegated to after the expression has been completed typechecked. However, these verifications may benefit from the typechecker stepping through the expression. For instance, in the case of a typing claim on an `if` expression, it may help to verify the claim separately on each of the branches while they are being typechecked rather than wait until the expression has been completely typechecked. This observation applies to `either` and `both` expressions as well. The typecheck proof macro is designed to either discard or pass along the optional verification requests along with the subexpressions based on the constructor of the expression being processed.

The overall typecheck proof form can then be described using the following rules. Unless otherwise mentioned, any optional cardinality or typing claim requests are discarded in recursive typecheck proofs.

- When typechecking an `if` expression, typecheck the then branch under the assumption that the condition is true, and typecheck the else branch under the assumption that the condition is false. Pass along any optional cardinality or typing claim requests.
- When typechecking a `when` expression, typecheck the body under the assumption that the condition is true. Pass along any optional cardinality or typing claim requests.
- When typechecking an `either` expression, recursively typecheck each component of the expression separately. Pass along any optional cardinality or typing claim requests.
- When typechecking a `both` expression, recursively typecheck each component of the expression separately. Pass along any optional cardinality or typing claim requests.
- When typechecking an `(except A B)` expression, recursively typecheck `A` passing along any cardinality or typing claim requests. Typecheck `B` separately.
- When typechecking a `(some-such-that ((var type)) formula)` expression, first typecheck `type`, then create a `suppose-there-is` proof using the `(var type)` binding and a typecheck of `formula` as its body.
- When typechecking a `lambda` expression, create a `suppose-there-is` proof that uses the `lambda` expression's argument list as its binding list and consists of the typecheck of the lambda body as its proof body. It is important to note that the typecheck of the `suppose-there-is` proof form will result in the types from the binding list being typechecked.

- When typechecking either `(the-set S)` or `(the-type T)`, recursively typecheck `S` or `T` respectively.

- When typechecking `(the-set x type such-that formula)`, first typecheck `type`, then evaluate a `suppose-there-is` proof with the `(x type)` binding and the typecheck of `formula` in the body.

- When typechecking an operator class of the form `(an operator-class from` $\texttt{domain}_1$ `and` $\texttt{domain}_2$ `... and` $\texttt{domain}_k$ `to range)`, each of $domain_1$ through $domain_k$ and *range* are typechecked and verified to be `type-class`.

- When typechecking `(a member T)`, `T` is typechecked and verified to be either `set-class` or `type-class`.

- When typechecking an expression of the form `(a ?op X)` where `?op` is a unary impredicative operator, `X` is typechecked but also verified to be of the appropriate category. In the case when `?op` is `subtype`, `X` needs to be `type-class`, similarly for `subset`, it needs to be `set-class`. If `?op` is either `domain`, `range`, `inverse` or `transitive-closure`, `X` needs to be `operator-class`. Finally, if `?op` is `car` or `cdr`, `X` needs to be `cons-cell-class`.

- If the expression is an application $(a\ fun\ arg_1\ ...\ arg_n)$, then first typecheck fun and then for each arg_i, typecheck arg_i and verify that it is in the domain class of $(a\ fun\ arg1\ ...\ arg_{i-1})$.

- If the expression is a quantified formula $(quantifier\ bindings\ formula)$, then create a `suppose-there-is` proof with *bindings* and a proof body that typechecks $formula$.

- If the expression is an if and only if expression, `(iff formulas)`, typecheck each of `formulas` separately.

- If the expression is an implication, `(implies ?phi ?psi)`, then typecheck `?phi` first and then `?psi` under the assumption that `?phi` is true.

- If the expression is a conjunction, $(and\ formula_1\ ...\ formula_k)$, then typecheck each $formula_i$ under the assumption that each of $formula_1$ through $formula_{i-1}$ are false. The reasoning behind assuming the previous conjuncts false is to ensure that they do not cause the a later conjunct to succeed typechecking if they aren't true to begin with.

The resulting typechecker proof macro is used in the common user proof forms such as `show`, `let-be`, `suppose-there-is` and `suppose` to typecheck the user-typed expressions. An attempt is made to aid proof automation by using the typechecker to possibly verify the typing claim when a `show` goal is an `is` formula. Similarly, when a `show` goal is a cardinality formula such as `there-exists`, `at-most-one` or `singleton`, the typechecker is provided with an optional cardinality claim to verify. If the typechecker does not succeed in verifying the goal, the `show` proof form falls back to using forward chaining inference and second-try.

This concludes the first half of the thesis, a novel application of Ontic to the logical analysis of automated planning domains is described next.

6. REASONING ABOUT PLANNING DOMAINS

Planning problems are commonplace in the real world. For instance, the task of picking up packages from different locations and delivering them to various other locations in a city involves rudimentary planning. A route to the next package's source and from each package's source to its destination must be determined and followed. Additional considerations such as reducing the total distance traveled require the use of heuristics to determine an ordering of package delivery. Similarly, airport crew need to determine possible routes from runways to gates that avoid long queues of landing aircraft. Warehouse crew often need to unstack and move crates around to get to a desired crate, while leaving sufficient room for maneuverability.

More formally, planning can be thought of as the process of identifying a sequence of actions that can accomplish a desired objective. Automated planning is the computational approach to this process. Planning problems are abstracted into formal planning domains and automated planners are sought that when provided an initial and desired configuration can generate a sequence of actions to accomplish this desired configuration starting from the initial configuration. Humans are generally well-equipped to handle planning tasks and more specifically they are easily able to answer questions or make observations about them. For instance, in a city where some package's source and destination are on either side of a river, it is obvious that the river necessarily needs to be crossed to deliver all packages. Similarly, in order to fetch a crate from a stack, only the crates above it need to be moved.

A formalization of the stacking and unstacking problem is one of the most popular automated planning domains, "Blocksworld". Blocksworld involves stacks of blocks on a table where the desired goal configurations can require them to be stacked in towers in a particular order. There are several observations that are immediately obvious to humans about this domain. For instance, consider being provided with

blocks of two different colors, red and green. All the blocks need to be utilized to construct a tower of alternating colors. It is obvious that such a tower can be constructed only if the difference in the number of blocks of each color is atmost one. It is also obvious that the color with the greater number of blocks is necessarily at the bottom of the tower. Consider the more general problem of having to build a tower out of uniquely identifiable blocks. It is obvious that any tower can be built by placing all the blocks on the table and then constructing the tower from the bottom up. A key observation here is that constructing the tower from the bottom up preserves the structure from the table up to the block currently being operated on.

The position taken in this thesis is that deduction plays a key role in this human ability. A system that is capable of verifying such observations about a planning domain can aid in the construction of a provably correct solution for planning problems. For instance, verifying that every tower can be unstacked to get all the blocks on the table and that building the desired towers from the bottom up can produce any desired configuration is central to deriving a provably correct general purpose solution to any Blocksworld problem. The rest of this chapter presents an overview of the various automated planning methods, a description of how a proof assistant and more specifically, Ontic can be employed to represent and conduct useful reasoning about planning domains, a representation of a general purpose plan for the two benchmark planning domains, Blocksworld and Logistics in the Ontic language and results on verifications of a proof of correctness of these plans.

6.1 Automated Planning

Planning in general refers to the process of selecting and ordering actions based on their outcomes in order to achieve a particular objective. Automated planning [7] is the field concerned with a computational approach to this process. The "planning problem" can be represented conceptually as a state transition system (S, A, γ) where $S = \{s_0, s_1, ...s_n\}$ represents a set of states, $A = \{a_0, a_1, ...a_m\}$ a set of actions, and

$\gamma\colon S \times A \to 2^S$ the underlying state transition relation. A distinguished "goal" state (or set of states) s_g represents the objective of the planning problem. The task of the "planner" is to then produce a sequence of actions or more generally a mapping from states to actions that can reach the goal state from a provided "initial state", s_0. Restrictions on this state transition system differentiate the range of planning problems. For instance, we can define a relation η from states to observations that maps a state to a set of observations about the objects in the planning problem. The planner is then no longer aware of the current state of the transition system and must instead rely on the current set of observations when deciding on the action to be chosen. In a "fully observable" planning problem, each state produces a unique set of observations, making η functional. On the other hand, in a "partially observable" planning problem, multiple states may generate the same set of observations. Similarly, the properties of the transition relation distinguish between deterministic and stochastic planning problems based on whether $\gamma(s, a)$ for some arbitrary state s and action a maps to atmost one state s' or not. In the case of stochastic planning problems, the planner needs to take into account, the different possible states that taking action a in state s may lead to. This work is primarily concerned with the simplest form, "classical planning" where the state transition relation is deterministic and the states are fully observable and the set of states and actions is finite. The reasons for this are two-fold: first, most of the interesting issues in automated planning and the analysis of planning problems are already present and concisely describable in this simple form, and second, none of the current systems for classical planning can as yet approach the human ability to efficiently analyze and solve such planning problems, making the pursuit of the harder planning cases a non-immediate goal.

6.1.1 Planning Representations

There are several possible representations that can be chosen for the state and action spaces. In the simplest `STRIPS` representation, states are sets of logical propo-

sitions representing the set of propositions that are true in that state and actions are a triple, $a = (precond(a), effects^{+}(a), effects^{-}(a))$, where $precond(a)$, $effects^{+}(a)$ and $effects^{-}(a)$ are sets of propositions representing the preconditions, add and delete effects, respectively of the action a. The action a is said to be "applicable" in a state s if $precond(a) \subseteq s$, i.e., its set of preconditions is a subset of the propositions in s. Executing a in s produces the new state $s' = (s - effects^{-}(a) \cup effects^{+}(a))$, where the delete effects have first been removed and the add effects added to the resulting state. Inspite of this simplicity in representation, this complete propositionalization is obviously inefficient in planning problems with a large number of objects. A better approach involves "lifting" the actions to operator spaces where the preconditions, add and delete effects are no longer propositions but predicates applied to variables of appropriate object types. Individual actions applicable in states can then be generated by instantiating the operator with the appropriate objects. Lifting to a predicate logic representation also provides the ability to now concisely specify operator preconditions and effects using quantified formulas. This lifted representation forms the basis of the widely used standard planning domain definition language, `PDDL` [8]. A "planning domain" in this `PDDL` representation defines a language for representing a class of planning problems. This language specifies a set of object types and predicates over these object types along with an associated set of operator spaces defining the actions that can be performed on objects of these types. A "planning problem" for a particular domain specifies the set of objects (from the domain types) relevant to that problem and the initial and goal states represented as grounded formulas built from the planning domain predicates instantiated with these objects. The set of available actions in a planning problem are exactly those that can be produced by instantiating the operator spaces with the objects in this planning problem.

6.1.2 Existing Planning Methods

A brief description of the different existing planning methods follows. While most of the popular current planning systems still use some form of heuristic search to solve each individual planning problem, there exist approaches that seek general-purpose solutions for the whole planning domain. Such approaches involve either the generalization of solutions from multiple sample problems or the solution of interesting abstractions of the planning domain. The current goal of this research is not to field an automated planning system, but instead to develop a system that can represent and reason about the planning issues that arise when seeking a general-purpose plan that solves a planning domain. In particular, results are reported on the use of automated reasoning for interactively verifying statements about two benchmark planning domains, Blocksworld and Logistics whose `PDDL` domain definitions can be found in the appendix. Thus, while this work is not directly comparable to any of the planning methods below, such a reasoning system could be used in conjunction with a planning method to aid it in the search for a provably-correct solution plan.

State-space search The simplest and most obvious approach to solving a planning problem is to conduct a directed search in the state space from the initial state to the goal via a sequence of action transitions. This sequence of actions is then the desired "plan". Several heuristics have been proposed in the past that estimate the distance from the current state to the goal to aid this search. Some of the most popular such heuristics include the delete relaxation heuristic [9], where the distance to the goal is computed in a modified planning problem where actions do not have any delete effects, the additive heuristic [10] where the distance to the goal is computed as the sum of the cost of achieving each goal proposition separately and the more recent landmark cut heuristics [11] which identify the set of actions that must be taken in any successful plan, thus lower bounding the length of such plans.

Another significant development in heuristic search based planning was the notion of a "planning graph" [12] consisting of alternating layers of actions and propositions

starting with the propositions that are true in the initial state. The planning graph efficiently represents reachability information by representing the set of states that can *possibly* be reached from the initial state. The actions in a particular layer are all those actions whose preconditions are satisfied in the preceding proposition layer, while the succeeding proposition layer includes all action effects from the preceding layer in addition to the propositions from the previous proposition layer. Mutex information can be computed for the proposition and action layers by identifying inconsistent or competing propositions (due to mutexed actions adding them) and actions (due to mutexed preconditions in the preceding proposition layer) in each layer. Plan extraction proceeds by starting from a layer where all goal propositions are true and successively extracting a set of non-mutex actions at the preceding layer that add all propositions that are required true at the current layer with the preconditions of these actions forming the desired propositions at the previous layer and so on until the desired propositions are a subset of those that are true in the initial state. Failures can be backtracked from by choosing an alternate set of non-mutex actions to achieve the propositions at a particular layer or by extending the planning graph until a fixpoint is reached.

This planning graph approach forms the basis of the `Graphplan` [12] and the widely popular `FF` [9] planning system that has dominated the international planning competitions in past years. The `FF` planning system constructs the plan graph for a delete relaxed version of the planning problem and uses the length of the resulting plan as a heuristic estimate for the current state. It then performs a variant of the "hill-climbing" search procedure to identify a successor state with a strictly lower heuristic value, eventually leading to the goal. Brute force search is employed when this search procedure fails to reach the goal. Additional heuristic measures such as "added goal deletion" which disprefer actions that delete goal propositions that have been made true and "goal agenda" which specifies a preferred order for achieving the goal propositions are incorporated based on their usefulness in particular planning domains. While these search based approaches and `FF` in particular prove effective

on smaller planning problems, the need for propositionalization and the need to start from scratch for each newly encountered planning problem makes this an ineffective approach on larger planning problems with a considerable number of objects.

Domain-expert provided heuristics have also been used in the past to guide the state space search making it more efficient. For instance, `TLPlan` [13] uses domain-expert provided temporal logic formulas to specify constraints on the search tree to avoid the consideration of useless search paths that can be pruned without affecting the completeness of the search procedure. Examples of such constraints in the Blocksworld domain include: "no block should be picked up unless it is the next needed block or above a next needed block" (a next needed block is a block such that the block below it in the goal state has been placed in the correct position) and "a block that has been put down should not be picked up immediately".

Plan space search In contrast to the state space search method, plan space search-based planners [14–16] conduct a search in the space of partial plans. A "partial plan" consists of a set of actions with some ordering constraints enforced between them, "causal links" specifying the relationship between a fact and the action adding that fact and "threats" identifying actions deleting desired facts. A partial planning system starts by adding the goal propositions to the set of "open propositions" that need to be currently achieved. An open proposition can be removed from this set by the addition to the partial plan of an action that achieves this proposition. The preconditions of the action are in turn added to the set of open propositions. Threats are identified and resolved by a number of resolution methods including the addition of ordering constraints to order an action before another action whose required effect it deletes, or by choosing alternative actions in a backtrackable manner. When the current set of open propositions are covered by the initial state and no threats are present, planning is done and a solution plan can be extracted based on the ordering constraints present.

Deduction and Learning-based approaches Some of the earliest attempts at automated planning employed deductive inference methods to construct a solution plan [16–18]. The planning problem was posed as a desired existence theorem for a sequence of actions that achieves the goal. A proof of this fact produces the desired plan as a side-effect. For example, Manna and Waldinger [19] described the use of the deductive tableaux method to construct a recursive plan that clears an arbitrary block in a stack of blocks. However, human input was required to provide the necessary strengthened induction hypothesis for the system to be able to verify the correctness of the derived plan. Similarly, the deduction based refinement planning approach [20] uses human provided abstract solution plans using abstract operators that can be proven to achieve desired effects when certain human specified preconditions are met. This abstract solution is then "refined" by the addition of concrete actions to achieve the necessary preconditions of the abstract operators. In other work, the planning problem was encoded as a satisfiability problem that could be solved using traditional SAT-solvers [21]. The planning domain is modified such that each domain predicate has an additional argument to specify the step where it is true. Similarly, a new predicate with an action and step argument is added that holds true when the action is taken in that particular step. A possible plan length is chosen and the planning problem can then be encoded as a conjunctive formula where the initial state facts are true in step 1, the goal facts are true at the step equal to the plan length chosen before and a disjunctive formula is added at each step encoding the different possible action choices at that step. The action precondition and effect formulas at each step can be similarly encoded. A solution to this satisfiability problem provides an action choice at each step by making one of the new action-step propositions true for each step. If no solution is found for the current plan length chosen, then a new length is chosen and the process repeated. Machine learning methods have also been used to learn control rules and decision-list based action choices from sample plans generated by traditional planners [22–24]. However, in contrast to deduction based planning,

the learnt policy (a mapping from states to action choices) does not come with any plan correctness guarantees.

Abstraction-based and Generalized Planning The realization that solution plans for different planning problems can often be abstracted into a single general-purpose plan for solving multiple problems has led to development of planning methods seeking such program-like plans. The simplest such abstraction method, combines individual actions into "macros" that can be be used in place of the original actions for obtaining a solution plan. Macros can be constructed either from a simple sequence of actions that find repeated use in sample planning solutions, by filtering out the operators with the best utility from an exhaustively generated set of such macro operators, or from the solutions to abstractions of the planning problems where some features of the state space have been abstracted away [25, 26]. Other generalized planning approaches seek program-like plans containing program constructs such as loops, conditionals and recursion. LoopDISTILL [27] identifies and annotates repeated structure in sample plans and replaces them with looping constructs. KPLANNER [28] uses a unique "planning parameter" that when known for a particular problem precludes the need for a looping plan for that problem. Thus, looping plans are only required when the value of this parameter is not known. The user specifies two values for this parameter, N_1 and N_2 such that the planner searches for a plan that works for all problems with parameter values less than N_1 and then checks to see whether the same plan works for all values upto N_2 as well. In the process of generating a plan, the planner also checks to see if any part of the plan is the unwinding of a loop and if so, returns the loop as well. The hope is that any plan that works for all values upto N_2 will work for any possible value of the planning parameter for that domain. FSAPLANNER [29] is another loop generating planner that uses finite state automatons to abstract the plan search process in the presence of observations. Rather than provide a mapping from plan states to actions, FSAPLANNER seeks a mapping from the finite state automaton state-observation pairs to actions. Whenever a state tran-

sition occurs due to taking an action, FSAPLANNER chooses a state in the FSA to transition to instead (with the transition annotated with both the action taken and the observation observed). Repeated FSA state choices thus, automatically establish a potential loop in the plan. A backtracking search is used to choose either alternative actions or alternative or new FSA states when the planner fails to reach the goal. FSAPLANNER repeats this process for several planning problems in the domain to obtain a general-purpose looping plan for the domain. Another recent significant work in generalized planning utilizes three-valued approximation to aggregate planning states into abstract states by aggregating state objects into equivalence classes identified by roles [2]. A role is the set of unary domain predicates that are satisfied by an object. Binary domain predicates are collapsed to be asserted about these abstract objects (one for each equivalence class) with the truth value of these predicates ranging between $[0, 1/2, 1]$ based on whether the predicate is true for none of the objects in the equivalence class, some of them or all of them. Sample plans for the original domain can be traced through these abstract states identifying repeated structure giving rise to loops. Alternatively, simple forward search can be conducted from an abstract state corresponding to a class of initial states to an abstract state corresponding to the goal condition to obtain a plan in this abstract state space. Actions operate on arbitrary members chosen from roles with more than one objects, hence conditionals naturally arise in such plans depending on whether the role has any more objects or not following the choice. Preconditions in the form of simple constraints on role membership counts can be propagated back from the abstract goal state to the abstract initial state to identify conditions under which the plan under consideration is terminating and provably correct [30].

6.2 Representing Planning Domains in Ontic

Automatic translation of PDDL domain descriptions into the Ontic language is generally straightforward. It is important, however, to select a translation that ex-

ploits the class-based reasoning available in Ontic. To this end, the planning domain predicates are mechanically translated into Ontic operators as described below.

Single argument PDDL state predicates such as `ontable`(b) are translated into non-deterministic Ontic operators from states to blocks. So, for instance, the Ontic class expression `(an ontable-block s)`, denotes the blocks that are on the table in state s. Binary PDDL state predicates such as `on`(b, c) are also mechanically translated to non-deterministic operators. Given a block as input, along with the state, the operator yields a class collecting all related blocks under the binary predicate. So, the class `(a (the on-relation s) b)` denotes the class of all blocks on the block b in the state s.

In essence, each predicate is now translated into an operator that when given a state, returns the predicate's interpretation in that state. In the case of unary predicates, this interpretation is a class of domain objects, in the case of binary predicates, it is a relation (Ontic operator) of the appropriate domain and range types. For simplicity, only unary and binary predicates are currently considered, but it is straightforward to extend this translation process to predicates with more than two arguments.

PDDL abstract action definitions are mechanically translated to Ontic operators mapping values for the action parameters to concrete actions. A concrete action is an Ontic structure that in addition to storing a list of its arguments, has slots for the action preconditions and effects. The action preconditions are abstracted into an Ontic type consisting of planning states that satisfy the precondition formulas, meanwhile the action effects are abstracted as a function from states that satisfy the preconditions to a new state that has the appropriately modified predicate interpretation. A new *result* operator is defined to be the effect of taking a concrete action in a state and abstracts the application of the action's effect slot to the input state. Thus, `(the result (the pickup b) s)` denotes the state that results from picking up block b in state s.

Action effects are defined in terms of four domain-independent operators that modify the predicate interpretations of a state in response to additions or deletions of unary and binary predicate facts, returning a new state with the modified predicate interpretation. The frame problem is handled by defining these operators to retain the predicate interpretations of all other predicates that aren't being modified. Thus, the *result* operator can be defined as a composition of the necessary predicate operations on the action arguments.

For instance, the `unstack` action applied to blocks $b1$ and $b2$ in state s can be defined as: `(the upred-add (the bpred-del s on b1 b2) holding b1)`

where *holding* and *on* are domain predicates in Blocksworld. The `upred-add` operator when applied to a state, a unary predicate and an object produces a new state with the object added to the predicate interpretation of the unary predicate in that state. The `bpred-del` operator when applied to a state, a binary predicate and two objects from the domain and range types of the binary predicate produces a state where the first object has been deleted from the class of objects that are related by the binary predicate to the second object. The interpretations of all other predicates remain the same across both states. In this case, the result of applying an `unstack` action to blocks $b1$ and $b2$ results in $b1$ being held and no longer on top of $b2$.

In addition to these definitions, the translation from PDDL includes verifications of the action effects. This has two purposes, first to ensure that the effect slot of each action is defined correctly, and second to generate theorems that are directly applicable to subsequent verifications about predicate interpretations of states resulting from executing a plan. These verifications result in theorems such as *the blocks on the table after executing an* `unstack` *action in state s are exactly the blocks on the table in the state s.*

```
(forall ((s (a blocksworld-state))
         (act (an unstack-action)))
  (= (an ontable-block (the result act s))
     (an ontable-block s)))
```

and a similar verification for the `on` predicate results in the theorem that *the blocks on a given block d after executing an action* `unstack`(b, c) *in state s are the blocks*

on d in s with b removed, if d and c are equal, and are exactly the blocks on d in s, otherwise:

```
(forall ((s (a blocksworld-state))
         (act (an unstack-action))
         (d (a block-in-state s)))
  (= (a (the on-relation (the result act s)) d)
     (if (= d (the second-arg act))
         (except (a (the on-relation s) d) (the first-arg act))
         (a (the on-relation s) d))))
```

The class `(a blocksworld-state)` is a specialization of the more general Ontic structure of planning states `(a state)`. The state structure has slots for the planning domain that it is a state of, and a predicate and type interpretation that abstract the predicate interpretations of the domain predicates in the state and the interpretation of the various domain object types. In the case of the *Blocksworld* domain, there is just one object type, *blocks*, so the type interpretation is the class of the blocks in that state.

6.3 Verification Tasks

One of the primary motivations of this research is insights drawn from the human reasoning processes involved in handling hypothetical planning scenarios. For example, consider a table with an arbitrary number of towers of blocks. Any block that is clear (i.e. does not have any blocks on top of it) can be picked up as long as no block is currently being held. Any block being held can either be placed on top of a clear block or on the table. This forms the basis of the familiar Blocksworld benchmark planning domain. However, while typical planners cannot function without being provided with a particular planning problem, humans can see several immediate conclusions and verify facts about this planning scenario without needing to consider any specific planning problem. For example, the following facts should be immediately obvious without ever having to consider a particular tower configuration or the number of blocks:

- any block in any of the towers can be picked up,

- any block can be put on the table by first picking it up and immediately putting it down
- any block a can be placed on top of any other block b without disturbing the towers that neither a nor b are a part of,
- in addition, the only blocks that need be affected are those that are possibly above a and b to begin with,
- any desired tower can be built bottom-up without having to disturb the partially constructed tower and,
- all blocks can eventually be put on the table.

Furthermore, these facts can be used to verify that the following general-purpose plan is executable and can correctly construct any arbitrary configuration of block towers: *"put all the blocks on the table and construct the desired towers bottom-up"*.

A description of the verifications carried out using the Ontic verification system for two benchmark planning domains, Blocksworld and Logistics follows.

6.3.1 State Invariants

A closer look at the reasoning steps involved in verifying the facts above identifies additional facts that need to (and are verified to) hold irrespective of the action taken. For instance, in order to verify that all the blocks can eventually be put on the table it is first necessary to verify that any block that is not yet on the table can be accessed and put on the table irrespective of all the actions that have been taken upto this point. A concise way of stating this fact is that each block should either be on the table, or held currently or in some block tower. Since we need to be able to reach any block in a tower to eventually pick it up and put it on the table, there cannot be any cycles in these block towers. In other words, each tower needs to contain a unique topmost block that can be cleared repeatedly until we get to the block in question.

Such statements are termed as "state invariants" in automated planning. They are facts about planning states that if true to begin with, remain true of all states that can be reached via an arbitrary sequence of action applications. A self-sustaining set of state invariants can be obtained using the following procedure:

1. Let $I_{in} = \{I_1, ...I_n\}$ be the set of candidate invariants, $I_{out} = \emptyset$
2. For each $I_i \in I_{in}$,
   (a) If $\cap_{1 \leq j \leq (length\ I_{in})}\ I_j\ (s) \implies I_i\ (the\ result\ \ a\ s)$ for some arbitrary action a and all states s that satisfy the preconditions of a, then
      i. $I_{out} = I_{out} \cup I_i$
3. If $I_{out} = I_{in}$, I_{out} is the desired set of state invariants, else repeat with $I_{in} = I_{out}$, $I_{out} = \emptyset$ unless I_{out} is empty.

The central verification task here is to verify whether each candidate invariant fact holds for the state (*the result act s*), for all actions *act* and state s that satisfies the preconditions of *act*, when all the candidate invariants are assumed to hold for state s. In the case of the Blocksworld domain, 9 such state invariants are verified to be preserved under application of 4 different action types, `(stack, unstack, pickup, putdown)`. In the Logistics domain, 5 such state invariants are verified to be preserved under application of 3 different action types, `(load, unload, drive)`. Each of these verifications can be carried out in Ontic with no human interaction and are thus fully automated. A complete set of the invariants verified for the Blocksworld and Logistics domains can be found in Figure 6.1.

Verifying the no cycle in block towers invariant The verification of the state invariant in Blocksworld that no cycles of blocks can be introduced requires the implementation of a new proof method in Ontic to reason about transitive closures. In this case, a proof is required of the fact:

1. 9 statespace invariants are preserved by 4 Blocksworld actions

   a. At most one block is held.
   b. Nothing is on the held block.
   c. At most one block is on a block.
   d. At most one block is under a block.
   e. No block on the table is held.
   f. No block on a block is held.
   g. No block on the table is on a block.
   h. No block is transitively on itself.
   i. Every block is either on the table, on a block or held.

2. 5 statespace invariants are preserved by 3 Logistics actions

   a. No package is `in` two trucks.
   b. No object is `at` two locations.
   c. Each truck is `at` a location.
   d. No package is `in` a truck and `at` a location.
   e. Each package is either `in` a truck or `at` a location.

Fig. 6.1.: Verifications conducted for Blocksworld and Logistics state invariants

```
(forall ((s (a blocksworld-state))
         (act (either (a stack-action) (an unstack-action)
                      (a pickup-action) (a putdown-action)))
         (b (a block-in-state s)))
  (is-never b (a (the on-relation (the result act s))^+ b)))
```

For conciseness, we use `s'` to denote `(the result act s)` as `s'` below.

Since (*the on-relation* s') is an operator from blocks to blocks in the state s', (*the on-relation* s')$^+$ is the transitive closure of this operator (relation).
Thus, $\big(a\ (\textit{the on-relation}\ s')^+\ b\big)$ represents the class of blocks that are above b in the state s'.[1] The above disjointedness fact states that b is not a block above itself, or equivalently, b is not a part of a cycle of blocks. For the case where a is an `unstack, pickup` or `putdown` action, the following reasoning using Ontic's forward-chaining inference rules about the subrelation property and transitive closures leads to the automated verification of the desired invariant property:

1. the `on` relation in the state s' is a subrelation of the `on` relation in state s based on the action effect axioms (no new "on" facts are added).

[1]The class of blocks in the state s is the same as the class of blocks in the state s' since there is no action that can add or delete blocks.

2. the $(\textit{the on-relation } s')^+$ relation is hence a subrelation of the $(\textit{the on-relation } s)^+$ relation.

3. $\big(a\ (\textit{the on-relation } s')^+\ b\big)$ is hence a subtype of the class $\big(a\ (\textit{the on-relation } s)^+\ b\big)$.

4. $\big(a\ (\textit{the on-relation } s')^+\ b\big)$ is disjoint from b follows from disjointedness reasoning about subtypes and the invariant assumption that b is disjoint from $\big(a\ (\textit{the on-relation } s)^+\ b\big)$.

For the case where *act* is a `stack` action, a new "on" fact has been added, potentially creating the cycle of blocks. For queries mentioning the application of the transitive closure of an operator, that cannot be verified using the base-reasoner, Ontic is instructed to attempt an alternate method to verifying the query. This alternate method instructs Ontic to attempt a progressive approximation of this applicative expression from below (starting with existing class expressions that are subtypes of this expression and introducing new class expressions in a controlled manner) until either an equivalent expression can be obtained or a predefined syntactic expression size threshold is reached.

If the size limit is reached, Ontic signals a failure to verify the query, requiring further human interaction. If an equivalent expression has been obtained, Ontic attempts to verify the original query by substituting in this equivalent expression. Introduction of this new equivalent class expression potentially enables further forward-chaining inference and the verification of the desired query. This process can be formalized as follows:

1. Let $(R^*\ C)$ be the transitive closure expression of interest and $seeds = \{S_1, ...S_n\}$ be a set of class expressions such that $(is\ S_i\ (R^*\ C))$ holds for each S_i.

2. For each seed S_i

   (a) Let $A = S_i$ be the current approximation, $C_D = S_i$ be the current candidate

(b) While $\mathit{expression\text{-}size}\,(A) < \mathit{size\text{-}threshold}$ repeat

i. $C_D = (R\ C_D)$

ii. If $(\mathit{is}\ C_D\ A)$ holds, break

iii. Else, $A = (\mathit{either}\ C_D\ A)$

(c) If $(\mathit{is}\ \ (R\ A)\ \ A)$ and $(\mathit{is}\ C\ A)$ hold, construct a new query by substiting A for $(R^*\ C)$ in the original query

(d) If this new query can be verified, break

(e) Else repeat

The algorithm above constructs a growing approximation A starting from each seed S_i by adding $(R\ S_i)$, $(R\ \ (R\ S_i))$ and so on to a growing union expression until a size limit is reached. It is straightforward to see that each of these expressions is a subtype of the class expression $(R^*\ C)$ and hence A bounds $(R^*\ C)$ from below. However, we can stop adding new candidate expressions to A when the candidate expression does not contribute any new members to A. At this point, $(\mathit{is}\ \ (R\ A)\ \ A)$ can be seen to hold. If $(\mathit{is}\ C\ A)$ is true as well, then A bounds $(R^*\ C)$ from above as well, making A equal to $(R^*\ C)$, providing a new equivalent class expression for $(R^*\ C)$ which can then be used in a new (equivalent) query.

When trying to prove the cycles invariant for some arbitrary `stack` action *act*, state s and block b, Ontic can infer that $\big(a\ \ (\mathit{the\ on\text{-}relation}\ s)^+\ \ b\big)$ is a seed for $\big(a\ \ (\mathit{the\ on\text{-}relation}\ s')^+\ \ b\big)$. For some block b that is not one of the arguments to this action, Ontic automatically obtains
$\big(\mathit{either}\ \ \big(a\ \ (\mathit{the\ on\text{-}relation}\ s')\ \ \big(a\ \ (\mathit{the\ on\text{-}relation}\ s)^+\ \ b\big)\big)\ \ \big(a\ \ (\mathit{the\ on\text{-}relation}\ s)^+\ \ b\big)\big)$
as an equivalent expression for
$\big(a\ \ (\mathit{the\ on\text{-}relation}\ s')^+\ \ b\big)$ using the procedure above and is subsequently able to verify the invariant automatically (verification of the invariant for the action arguments is trivial).

6.3.2 Predicate-achievement Macros and Generalized Plans

A natural approach to solving a complex goal is to break it down into subgoals, solving each of them separately and then combining the solutions. In the case of planning problems with conjunctive goals, this most obviously translates into a process of achieving each goal conjunct separately while preserving the ones that have already been achieved. Often, several goal conjuncts can be achieved using the same basic subplan and such repeated structure has been identified and exploited by planning systems in the past to convert a sequential plan into a looping plan or to extract useful macro actions. For example, in a package delivery domain such as `Logistics`, a delivery "subplan" needs to be repeatedly invoked on each undelivered package until no such packages remain. The basic structure of this subplan involves driving a truck to pickup the package at its current location, load the package onto the truck, drive the truck to the package's destination and unload the package.

One of the goals of this work is to illustrate how reasoning about a planning scenario can be organized around the development and analysis of macros that achieve individual domain predicates. Verification of the effects of such macros on the domain predicates provides useful information on ordering, composing and interleaving such macros to achieve other domain predicates or complex conjunctive goals. For example, consider a macro operator for the Blocksworld domain that clears a block by repeatedly clearing off all the blocks above this block and putting them on the table. This macro can in turn be used to construct a macro that places block a on block b by first invoking the "make-clear" macro to clear blocks a and b and then picks up a and stacks it on b. This new "make-on" macro can be verified to have the effect that all "on" facts between blocks are preserved except for those blocks that were either above a or b to begin with. Similarly, the "make-clear" macro can be analyzed to verify that any blocks that were above the block being cleared end up on the table after the macro is run. The fact that "make-on" preserves all "on" facts for blocks below b immediately suggests that any desired tower should be built bottom

up by repeatedly invoking the "make-on" macro on the desired "on" facts from the bottom up. This also suggests that any "ontable" goal facts need to be achieved first before constructing the goal towers.

Such macros can be represented in the Ontic language as (recursive) Ontic operators that are built from the basic *result* operator defined on state-action pairs that produces a state resulting from taking the action in the state along with constructs such as "if", "when", etc. Examples of such predicate achievement macros for the Blocksworld domain can be found in Figure 6.2. A complete set of definitions of predicate achievement macros for the Blocksworld and Logistics domains can be found in the appendix. Interactive verifications are carried out for the effects of such hand-written macros on each of the domain predicates. In addition verifications are carried out to ensure that each such macro produces a singleton result state (all actions are applied to states satisfying their preconditions, termination, etc.). The individual predicate achievement macros can be composed into an Ontic operator defining the generalized plan to achieve an input goal state from the input initial state. An example hand-written definition of such a generalized plan is provided in Figure 6.3. Similar verifications can be carried out to ensure the well-definedness of this generalized plan, essentially that the result is a well-defined state that is equivalent to the input goal state (i.e. has the same predicate and type interpretations). The complete list of verifications carried out for the Blocksworld and Logistics predicate achievement macros and generalized plans can be found in Figure 6.4 and Figure 6.5.

6.4 Verification Results

The planning-related verifications described in the previous section were first carried out using the original Ontic system from the 1990s. The number of interactions required were carefully metered and characterized to identify how far the system is from fully automatic verification of these claims. The metric used is the number

make-clear(s, b)— If the type `(a held-block s)` is occupied, then take `putdown` on its occupant. Then, if the type `(a (the on-relation s) b)` is occupied by block c, i.e., there is a block c on b in s, then recursively `make-clear`(c), take `unstack` c, and take `putdown` c.

```
(define (the make-clear (s (a blocksworld-state))
                        (b (a block-in-state s)))
 (if (there-exists (a held-block s))
     (the make-clear (the result (the putdown (the held-block s)) s)
                     b)
     (if (not (there-exists (a (the on-relation s) b)))
         s
         (the result (the putdown (a (the on-relation s) b))
           (the result (the unstack (a (the on-relation s) b) b)
              (the make-clear s (a (the on-relation s) b)))))))
```

make-on(s, b, c)— Make c clear, then make b held, then stack b on c.

```
(define (the make-on (s (a blocksworld-state))
                     (b (a block-in-state s))
                     (c (a block-in-state s) such-that (not (= b c))))
  (the result (the stack b c) (the make-held (the make-clear s c) b)))
```

Fig. 6.2.: Example predicate-achievement macros.

genplan(s, g)— First repeat until all blocks that are desired on the table in the goal are moved to the table, then repeat until there exists some block that is in its final position but has an incorrect block on it (or needs to be made clear), place the correct block on it (or clear the block), finally pickup any block that needs to be held in the goal.

```
(define (the genplan (s (a blocksworld-state)) (g (a valid-goal-for s)))
  (if (there-exists (except (an ontable-block g) (an ontable-block s)))
      (let ((b (except (an ontable-block g) (an ontable-block s))))
        (the genplan (the make-ontable s b) g))
    (if (there-exists (some-such-that b (a correctly-solved-block s g)
                          (not (= (a (the on-relation s) b)
                                  (a (the on-relation g) b)))))
        (let ((b (some-such-that b (a correctly-solved-block s g)
                    (not (= (a (the on-relation s) b)
                            (a (the on-relation g) b))))))
          (if (not (there-exists (a (the on-relation g) b)))
              (the genplan (the make-clear s b) g)
            (the genplan (the make-on s (a (the on-relation g) b) b) g)))
        (if (there-exists (a held-block g))
            (the make-held s (a held-block g))
          s))))
```

Fig. 6.3.: Generalized plan for the Blocksworld domain using predicate-achievement macros.

1. Exact effects of `make-clear`(s, b) in Blocksworld.

   a. Executing the macro results in a single reachable state.
   b. The type "blocks on the table" adds any held block and all blocks above b.
   c. The type "clear blocks" grows, adding b and all blocks above b in s.
   d. The type "held blocks" becomes empty.
   e. For each block d, the type "blocks on d" is unchanged, except when d is the block b or a block above b , when it becomes empty.

2. Exact effects of `make-held`(s, b) in Blocksworld.

   a. Executing the macro results in a single reachable state.
   b-e. Four claims characterizing the state predicates, similar to `make-clear`.

3. Exact effects, `make-ontable`(s, b), Blocksworld (5 claims).

4. Exact effects of `make-on`(s, a, b) in Blocksworld.

   a. Executing the macro results in a single reachable state.
   b. The type "held blocks" becomes empty.
   c. For each d, the type `on`(d) becomes empty if d is a, above a, or above b; the type becomes a if $d = b$; the type is otherwise unchanged.
   d. "Blocks on table" adds any held block, and all above a or b, but excludes a.
   e. The type "clear blocks" adds a, and the blocks above a or b, but excludes b .

5. Two provided generalized plans are fully correct.

   a. A plan to reach any goal state without ever picking up correct blocks.
   b. A plan to place all blocks on the table.

Fig. 6.4.: Verifications conducted for Blocksworld

1. Exact effects of $\texttt{make-at}_{\texttt{truck}}(s, t, l)$ in Logistics.

   a. Executing the macro results in a single reachable state.
   b. For each location l', the type `at`(l') is unchanged, except `at`(l) gains t and `at`$(\texttt{at}^{-1}(t))$ loses t.
   c. For each truck t', the type `in`(t') is unchanged.
   d. For each object o, `location`(o) is unchanged, except `location`$(t) = l$.

2. Effects of $\texttt{make-at}_{\texttt{pkg}}(s, p, l)$ in Logistics.

   a-d. Like the other macros, except that which truck is used is left unspecified.

3. Exact effects, `make-in`(s, p, t), Logistics (4 claims).

4. A provided generalized plan is correct.

Fig. 6.5.: Verifications done for Simplified Logistics (trucks only)

of base-reasoner Socratic proof steps needed to verify a claim, with a value of zero indicating fully automatic verification.

Qualitative characterization of the user queries identified potentially eliminable interactions. Changes to quantifier processing, the second-try tactics and the addition of the typechecker described in this thesis were directly motivated by these identified eliminable interactions. However, the metric does not take into account the time spent by the inference engine on each user query. A quick response from the inference engine (even to signal failure) is generally preferable to a successful verification that takes much longer. Changes to the rule implementation, the addition of text and the pruning of the case analysis tree were motivated by this consideration. Any context extension taking longer than 10 seconds was examined to identify the source of the expense and direct improvements to the inference engine.

Table 6.1 provides the metric values for each of the verifications, for the two Ontic versions, the original Ontic from the 1990s and the current Ontic system.

Table 6.1.: Counts of human interactions for each verification evaluated in the original 1990s Ontic and the current Ontic.

English name	# Steps (1990s Ontic)	# Steps (current Ontic)
9 blocks invariants	0	0
`make-clear` well-defined	21	1
`make-clear` effects	0	3
`make-held` well defined	4	1
`make-held` effects	11	0
`make-ontable` well defined	5	0
`make-ontable` effects	6	0
`make-on` well defined	11	5
effect on `held`	0	0
effect on `on`	24	23
effect on `ontable`	9	3
effect on `clear`	28	0
generalized plan for any goal	418	242
genplan to put all blocks on table	14	14
5 logistics invariants	0	0
$\texttt{make-at}_{\texttt{truck}}$ well defined	2	2
effects of $\texttt{make-at}_{\texttt{truck}}$	1	2
$\texttt{make-at}_{\texttt{pkg}}$ well defined	6	6
effects of $\texttt{make-at}_{\texttt{pkg}}$	8	16
`make-in` well defined	9	4
effects of `make-in`	3	10
generalized plan	25	13

6.4.1 Discussion of Results

While the current Ontic significantly outperforms the previous version, there are still some verifications that are not yet fully automatic and require non-trivial user interaction. A discussion of these verifications that presented the greatest difficulty follows:

well-definedness of make-on The challenge in showing that `make-on` is well-defined is in proving that all actions are applied to states that satisfy the action's preconditions. Typechecking an action application expression, `(the result act s)` leads to Ontic attempting to automatically verify that state s is a member of the class of states satisfying the preconditions of *act*. However, due to the severely restricted second-try depth during typechecking, Ontic cannot automatically verify this fact about the action application expression
`(the result (the stack b1 b2) (the make-held (the make-clear s b2) b1))`
that is central to `(the make-on b1 b2)`. While Ontic can automatically infer that `b1` is a held block in the state resulting from the application of the `make-held` predicate achievement macro, verifying that `b2` is still clear in the state
`(the make-held (the make-clear s b2) b1)` is non-trivial. Since the action preconditions are represented using the basic domain predicates, a clear block is represented as the emptiness of the class of blocks on the block in question. Verifying that this fact holds for `b2`, then involves chaining inference about the effects of `make-held` and `make-clear` on the `on` predicate.

analysis of make-on The analysis of `make-on` requires composing the effects of `make-clear` and `make-held`. However, due to the two blocks involved, three separate cases need to be considered: first, where the two blocks are in distinct towers and two cases where the blocks are in the same tower with either one being higher up than the other. Ontic's automatic case analysis cannot discover these cases since they do not fall under any of the obvious sources of case formulas. In particular, natural

statements of the effects of the `make-clear` and `make-held` macros are theorems that quantify over the class of blocks that include all blocks except the ones above the block being operated on. If instead, the theorems were written to quantify over all blocks but included a conditional in the body testing for membership in these classes, that would provide the necessary pertinent cases. However, this observation is based on intimate knowledge of the implementation of Ontic case analysis. It is not the intent here to propose modeling verification statements based on the implementation of the inference engine. Future work is suggested that identifies additional sources of case analyses; in particular quantifier types that are specializations of other classes. It is important to note that the quantifier type in question is dependent and thus requires instantiation before it can be identified as seeding a useful case formula.

verification of the Blocksworld generalized plan While some of the simpler verifications involving action applicability are now eliminated with the improvements to Ontic, this proof still requires the development of non-trivial concepts such as a "correctly-solved-block", i.e. a block that is in its correct position all the way down to the table. This concept provides a notion of "progress" that can be used in an induction proof of termination. The generalized plan operator is defined for a Blocksworld state and a goal state, where the goal is expected to be "valid" for the source state; i.e. it needs to have the same type meaning. Recursive applications of this operator to intermediate states (where progress is being made towards the goal) require typechecking to ensure the validity of the original goal for these states as well. The general principle behind the generalized plan is to first move all the blocks on the table in the goal state to their correct position and then build the towers from the bottom up, achieving each desired `on` fact in turn and finally ensuring that the blocks that are at the top of the towers are clear. The uses of the `make-on` and `make-clear` macros need to be analyzed to ensure that they preserve or increase the number of correctly solved blocks. Termination is then proven based on the finiteness of the set of unsolved blocks. It is clear that a general-purpose verification system will be

unable to construct this proof completely automatically without the use of tactics specialized to automated planning. Future work is required to incorporate this into the Ontic system.

Logistics verifications Verifications of the effects of some of the predicate achievement macros for the Logistics domain can be seen to require more user interactions in the current Ontic. This can be attributed to the changes to quantifier instantiation. In order to prevent inference engine runaway, the layered interning approach now requires the presence of text to drive quantifier instantiation. Reasoning about the effects of the composition of predicate achievement macros and basic actions now require the user to explicitly mention the predicate interpretations of intermediate states. Similarly, quantified theorems involving dependent types such as the locations except for the location of a package in question, or the trucks except the one that a package is currently in, now require an instance of the type to be mentioned in the user proof to enable instantiation.

In general, the current Ontic system can be seen to have significantly reduced the number of verification steps required. The remaining user proof steps are either non-trivial concepts that need to be defined by the user and applied to the proof, non-trivial case analyses that need to be explicated or the result of restrictions requiring user-typed expressions to direct quantifier instantiation.

The primary goal in employing Ontic for these planning verifications has been to illustrate the use of a general-purpose, proof assistant in aiding automated planning with no prior planning knowledge built-in. Properties of state invariants and natural predicate achievement macros are exploited to construct a provably correct generalized plan that can solve any problem in the Blocksworld and Logistics domains without having to consider a specific planning problem. Integration with an automated planner that can drive the construction of the predicate achievement macros

and the generalized plan and further automation of these verifications is left for future work.

7. RELATED WORK

There are several proof assistants in existence, the intent in this thesis is not to compare Ontic against all of them or to conclusively establish its superiority over them. Proof assistants typically differ in their interaction and representation language and in the inference engine used to verify claims. While the amount of proof automation that can be obtained out of the box differs between proof assistants, most of them provide the ability to construct new tactics that can be used to automate certain repeated proof steps.

The use of taxonomic syntax in Ontic's representation enriches the quantifier-free fragment of the representation language, enabling the inference engine to verify more claims automatically. Claims involving quantifiers often require some user interaction to control the introduction of new expressions into the inferential context. Both taxonomic syntax and the Ontic proof language are close to mathematical English, enabling Ontic proofs to be very similar in structure to formal proofs in a textbook. The closeness to mathematical English also enables users to directly translate the terms and concepts from a mental informal proof into a formal proof in Ontic. In contrast, verifications using other proof assistants are often represented as a sequence of procedural instructions specifying the application of a particular proof tactic. In order to illustrate this difference, results from the verification of the Blocksworld state invariants in Coq are discussed below.

7.1 Planning-related Verifications in Coq

Initial experiments show a requirement for typically more human interaction in conducting the Coq verifications. There are 36 (9 invariants and 4 action types) invariant preservation theorems about the Blocksworld domain, stating that a given

```
Definition det_holding (s : state) :=                          Defining the invariant:
  forall b1 b2, holding b1 s ->                                   at most one block is
    holding b2 s -> b1 = b2.                                     being held in state s

Theorem pickup_det_holding :
  forall x s,
    osr s (take (pickup x) s)                          Hypothesis: (take (pickup x) s)
                                                          is one step reachable from s
    -> det_holding s                                   Hypothesis: holding at-most-one
                                                                          block in s
    -> det_holding (take (pickup x) s).                          Theorem: After taking
                                                                 (pickup x), will hold
                                                                     at-most-one block
Proof.
  intros x s Hosr Hprev.                                  Names variables and hypotheses
  inversion Hosr.                                  Regress one step reachable hypothesis
                                                       to action preconditions of pickup
                                                         -->results in H2, used below
  unfold det_holding.                               Expand the definition of det_holding
                                                                      in the proof goal
  intros b1 b2 Hb1 Hb2.                               Names for variables and hypotheses
                                                          generated in the previous step
  apply pickup_holding in Hb1.                              Apply the action dynamics
                                                              to get b1=x ∨ holding b1
  apply pickup_holding in Hb2.                              Apply the action dynamics
                                                              to get b2=x ∨ holding b2
  unfold handempty in H2.                             Expand the definition of handempty

  assert (not (holding b1 s)) by apply H2.                          Instantiate H2 on b1

  assert (not (holding b2 s)) by apply H2.                          Instantiate H2 on b2

  intuition.                                          Figure out propositional tautologies
  congruence.                                  Prove b1 = b2 based upon b1 = x and b2 = x
Qed.
```

Fig. 7.1.: An example proof in Coq. Instances of the unfold and assert tactic are domain-specific human inputs. Instances of inversion and apply could easily be automated in a planning-specific system. Both count as interactions here for comparison to Ontic.

action preserves a given invariant. *Each of these theorems is proven without human interaction by the Ontic system. All of these theorems required interaction in the Coq system.*

These 36 verifications in Coq required 237 human proof interactions, not counting the 36 theorem statements, or any human invocations of the tactics intros, auto, simpl, intuition, and unfolding of definitions appearing in the theorem, which are assumed to be easily automatable. An example Coq verification is shown in Figure 7.1 and the corresponding Ontic proof in Figure 7.2.

```
(suppose-there-is ((act (a blocksworld-action))
                   (s (both (a legal-bw-state)            Invariants assumed in s
                            (a member (the applicable-states act))))
 (suppose (is act (a pickup (a block-in-state s)))
   (show (at-most-one (a held-block (the result act s))))))
```

Fig. 7.2.: The Ontic proof corresponding to Figure 7.1.

Ontic appears to provide the following qualitative advantages over Coq for the planning verifications conducted in this work:

Taxonomic Representations. Due to the class-based syntax used in Ontic, the quantifier-free fragment of the Ontic language is more expressive than that of Coq. Ontic can state formulas such as $\forall x \exists y\, \texttt{ontable}(y) \wedge \texttt{on}(x, y) \rightarrow \texttt{clear}(x)$ without quantifiers (as the formula `(is (on ontable) clear)` stating a subtyping relationship between the type "on a block on table" and the type "clear block"). This quantifier-free expressiveness reduces the need for human interactions that assist in instantiating such quantified theorems.

Type-based Automatic Quantifier Instantiation. Ontic's quantifier instantiation mechanisms (in particular, subsumption matching) are much richer than simply instantiating theorems with conclusions that match pieces of the desired theorem—the effect of the "auto" tactic applied to a desired theorem (somewhat simplified). As a result, no human-selected instantiations are needed to verify the domain invariants in Ontic, whereas 146 human-selected instantiations were needed in the corresponding Coq verifications.

7.2 State Invariants and Generalized Planning

While there exist several prior approaches for the generation and verification of planning state invariants [1, 11, 31–33], they are still mostly remote to this work. In addition, the "no block cycles" invariant is inaccesssible to these previous systems. Furthermore, this work addresses a much more general set of questions than the verification of state invariants in a richer, more expressive language.

An example of the use of logical representations for generalized planning is the Manna and Waldinger work [19] using deductive tableaux to generate a provably-correct recursive plan to clear a block. While the plan has the same recursive structure as the `make-clear` predicate-achievement macro; that work provides no mechanism for analyzing the side-effects of such recursive plans. Such analysis can inform the use of the macro in generalized plans for arbitrary goal conjunctions.

Most relevant among recent work in generalized planning is the work of [2]. The work exploits a *role*-based abstraction to represent a generalized plan and detect loop termination by decreasing role counts. There are several respects in which this prior work is not directly comparable to the work described here. First, their method assumes domain invariants rather than verifying them. Second, the applicability of their method is unclear outside of $\mathcal{FC}^3$ domains, where the role-based abstraction can represent the choices needed. Finally, class expressions provide a richer abstraction than roles and will support a wider range of analyses.

8. SUMMARY

The contributions of this thesis are two-fold. First, it describes a novel application of a proof assistant, Ontic to automated planning in conducting a purely deductive verification of generalized plans for two benchmark planning domains. Second, it describes algorithmic improvements to Ontic that are directly inspired by both qualitative and quantitative characterizations of the interactions required for these planning verifications. These improvements result in a significant reduction in the required user interactions for the same planning verifications in the modified Ontic.

The application of Ontic to these planning verifications represents the first (to the best of our knowledge) purely deductive approach to generalized planning for arbitrary goals in the Blocksworld and simplified Logistics domains that did not require consideration of any specific planning problem from the domain. This task resulted in several contributions to both automated planning and the Ontic verification system:

1. A general principle for the translation of planning domains from the standard PDDL representation to a class-based syntax, the Ontic language.

2. Extensions to the Ontic language to naturally handle concepts involved in these verifications. In particular, support for representing and reasoning about transitive closure, operator subrelations and disjointedness was added.

3. The notion of a predicate-achievement macro was developed, and its effect on the domain predicates analyzed to aid in the composition of these macros to produce provably-correct program-like generalized plans.

4. A new proof method that automatically attempts to derive equivalent expressions for transitive closure applications was added. This enables the automatic

verification of the no block cycles invariant that has not been handled in prior related work.

The composition of the individual predicate-achievement macros into a generalized plan for conjunctive goals is only possible when the goals are serializable [34]. Ontic is intended as just a part of an automated planning system that can be used to represent candidate generalized plans and verify their correctness. The actual task of generating candidate generalized plans is left to the other components of such a system. However, the translation to Ontic classes enables the logical analysis of planning domains, and can aid in the construction of the generalized plans.

The original Ontic system from the 1990s was initially used to conduct these verifications. Subsequent characterization of the interactions required, identified several improvements that could be made. In addition, the expense of each rule was metered by counting the number of times a loop across objects in a relation list was executed. Sorting the rules by this count identified the most expensive rules and changes to the inference engine were made to either incorporate alternate rule orderings or rule splitting. In particular, careful characterization of the interactions required for these verifications as well as verifications on other benchmark problems resulted in the following improvements to Ontic:

1. Rather than determining a single antecedent ordering, rule antecedents are organized into a tree with the choice of the branch to take determined at rule execution time.

2. Rule variable coverage as well as causal links between variable bindings was used to determine efficient rule splitting for Ontic orcfuns. The split rules are chained using a new continuation predicate that mentions just the variables necessary for the chaining.

3. The identification of potential rule splits was modeled as the task of determining rule variables that can function as articulation points for the rule antecedent graph.

4. A new Herbrand category of Ontic objects, `text` was added. This enables the identification of expressions typed by the user and the subsequent control of new expression interning as well as quantifier instantiation.

5. Control predicates on `text` objects enables a distinction between expressions typed by the user and `text` expressions interned via Ontic rules. This distinction supports a layered interning approach with higher priority assigned to expressions typed by the user.

6. A reorganization of the quantifier matching and instantiation process now requires all dependent quantifier types to be instantiated first. This significantly reduces the potential substitutions considered for instantiation due to early typechecking.

7. A new typechecker proof form steps through Ontic expressions, verifying that each operator is applied to a class expression of the right domain type. Failure to verify this fact is signaled to the user, requiring a user proof before processing can continue. The typechecker is automatically employed on expressions from every user typed proof form. Due to the richness of the types involved, the typechecker can often help prove the original goal.

8. Case analysis is significantly reined in by only choosing case formulas where the goal is automatically inferrable by the inference engine on one side. This results in a linear tree structure, making case analysis quadratic in the number of case formulas.

9. Proof tactics that attempt to automatically construct a proof of a goal based on its syntactic structure can now automatically expand definitions of symbols present in the goal formula. This causes several claims to now be automatically verified when they required user proofs before.

While the improvements to Ontic listed above resulted in significant savings in the user interaction steps required, some verifications did require more user proof

steps than before. This can be attributed to the changes to the quantifier instantiation process that now require dependent quantifier types to be instantiated from matches to pre-existing context expressions. The ability to construct complex type expressions allows quantified facts to be very specific, quantified over just the types that the fact applies to. However, this in turn leads to such instantiations requiring instances of the types to already be interned and present in the current context. In general, maintaining a balance between the level of proof automation achievable and the efficiency of the inference engine requires restrictions that prevent inference engine runaway resulting from excessive interning of new expressions. The challenge in any future work improving Ontic is in maintaining this balance.

REFERENCES

REFERENCES

[1] M. Fox and D. Long, "The Automatic Inference of State Invariants in TIM," *Journal of Artificial Intelligence Research*, vol. 9, pp. 367–421, 1998.

[2] S. Srivastava, N. Immerman, and S. Zilberstein, "A new representation and associated algorithms for generalized planning," *Artificial Intelligence*, vol. 175, no. 2, pp. 615–647, Feb. 2011. [Online]. Available: http://dx.doi.org/10.1016/j.artint.2010.10.006

[3] D. McAllester and R. Givan, "Taxonomic syntax for first order inference," *Journal of the ACM*, vol. 40, no. 2, pp. 246–283, 1993.

[4] D. McAllester, "Observations on cognitive judgments," in *Proceedings of the Ninth National Conference on Artificial Intelligence*. Morgan Kaufmann Publishers, 1991, pp. 910–915.

[5] G. Nelson and D. C. Oppen, "Fast decision procedures based on congruence closure," *Journal of the ACM*, vol. 27, no. 2, pp. 356–364, April 1980.

[6] D. Cantone and M. N. Asmundo, "A further and effective liberalization of the δ-rule in free variable semantic tableaux," in *Automated Deduction in Classical and Non-Classical Logics*. Springer, 2000, pp. 109–125.

[7] M. Ghallab, D. Nau, and P. Traverso, *Automated Planning - Theory and Practice*. Elsevier, 2004.

[8] M. Fox and D. Long, "PDDL2.1: An extension to PDDL for expressing temporal planning domains," *Journal of Artificial Intelligence Research*, vol. 20, pp. 61–124, 2003.

[9] J. Hoffmann and B. Nebel, "The FF planning system: Fast Plan Generation Through Heuristic Search," *Journal of Artificial Intelligence Research*, vol. 14, no. 1, pp. 253–302, 2001.

[10] B. Bonet and H. Geffner, "Planning as heuristic search: New results," in *Recent Advances in AI Planning*. Springer, 2000, pp. 360–372.

[11] M. Helmert, "Concise finite-domain representations for PDDL planning tasks," *Artificial Intelligence*, vol. 173, no. 5, pp. 503–535, 2009.

[12] A. Blum and M. Furst, "Fast Planning Through Planning Graph Analysis," *Artificial intelligence*, vol. 90, no. 1, pp. 281–300, 1997.

[13] F. Bacchus and F. Kabanza, "Using temporal logics to express search control knowledge for planning," *Artificial Intelligence*, vol. 116, 2000.

[14] E. D. Sacerdoti, "A structure for plans and behavior," DTIC Document, Tech. Rep., 1975.

[15] D. McAllester and D. Rosenblitt, "Systematic nonlinear planning," in *Proceedings of the ninth National conference on Artificial intelligence - Volume 2*, ser. AAAI'91. AAAI Press, 1991, pp. 634–639.

[16] D. Chapman, "Planning for conjunctive goals," *Artificial intelligence*, vol. 32, no. 3, pp. 333–377, 1987.

[17] R. Reiter, "Proving properties of states in the situation calculus," *Artificial Intelligence*, vol. 64, pp. 337–351, 1993.

[18] C. A. Knoblock, J. D. Tenenberg, and Q. Yang, "Characterizing abstraction hierarchies for planning," in *Proceedings of the Ninth National Conference on Artificial intelligence - Volume 2*, ser. AAAI'91. AAAI Press, 1991, pp. 692–697.

[19] Z. Manna and R. Waldinger, "How to clear a block: A theory of plans," *Journal of Automated Reasoning*, vol. 3, no. 4, pp. 343–377, 1987.

[20] W. Stephan and S. Biundo, "Deduction-based refinement planning," in *Proceedings of the third International Conference on Artificial Intelligence Planning Systems (AIPS-96)*. AAAI Press, 1996, pp. 213–220.

[21] H. Kautz and B. Selman, "Planning as satisfiability," in *Proceedings of the 10th European conference on Artificial intelligence*, ser. ECAI '92. John Wiley & Sons, Inc., 1992, pp. 359–363.

[22] R. Khardon, "Learning action strategies for planning domains," *Artif. Intell.*, vol. 113, no. 1-2, pp. 125–148, Sep. 1999.

[23] M. Martín and H. Geffner, "Learning generalized policies from planning examples using concept languages," *Applied Intelligence*, vol. 20, no. 1, pp. 9–19, 2004.

[24] Y.-C. Huang, B. Selman, H. Kautz *et al.*, "Control knowledge in planning: benefits and tradeoffs," in *AAAI/IAAI*, 1999, pp. 511–517.

[25] A. Botea, M. Enzenberger, M. Müller, and J. Schaeffer, "Macro-ff: Improving ai planning with automatically learned macro-operators." *J. Artif. Intell. Res.(JAIR)*, vol. 24, pp. 581–621, 2005.

[26] A. Coles, M. Fox, and A. Smith, "Online identification of useful macro-actions for planning." in *ICAPS*, 2007, pp. 97–104.

[27] E. Winner and M. Veloso, "Loopdistill: Learning looping domain-specific planners from example plans," in *International Conference on Automated Planning and Scheduling, Workshop on Artificial Intelligence Planning and Learning*, 2007.

[28] H. J. Levesque, "Planning with loops," in *Proceedings of the 19th international joint conference on Artificial intelligence*, ser. IJCAI'05. Morgan Kaufmann Publishers Inc., 2005, pp. 509–515.

[29] Y. Hu and H. Levesque, "Planning with loops: Some new results," in *ICAPS Workshop on Generalized Planning*, 2009.

[30] S. Srivastava, N. Immerman, and S. Zilberstein, "Applicability conditions for plans with loops: Computability results and algorithms," *Artificial Intelligence*, vol. 191-192, pp. 1–19, Nov. 2012. [Online]. Available: http://dx.doi.org/10.1016/j.artint.2012.07.005

[31] J. Rintanen, "An iterative algorithm for synthesizing invariants," in *Proceedings of the Seventeenth National Conference on Artificial Intelligence*. AAAI Press, 2000, pp. 806–811.

[32] A. Gerevini and L. Schubert, "Discovering state constraints in DISCOPLAN: Some new results," in *Proceedings of the Seventeenth National Conference on Artificial Intelligence*. AAAI Press, 2000, pp. 761–767.

[33] F. Lin, "Discovering state invariants," in *Proceedings of the Ninth International Conference on Principles of Knowledge Representation and Reasoning, KR*, vol. 4, 2004, pp. 536–544.

[34] R. E. Korf, "Planning as search: A quantitative approach," *Artificial Intelligence*, vol. 33, no. 1, pp. 65–88, 1987.

APPENDICES

A. ONTIC GRAMMAR

A.1 Ontic Expression Syntax

Ontic expressions typed by the user broadly fall into two categories: class expressions and formula expressions. Below is a syntax grammar for each of these two categories.

A.1.1 Class Expressions

⟨*article*⟩ ::= `a` | `an` | `the`

⟨*unary-impredicative*⟩ ::= `member` | `subset` | `subtype`
| `domain` | `range` | `inverse` | `transitive-closure`
| `car` | `first` | `cdr` | `rest`

⟨*binary-impredicative*⟩ ::= `operator` | `total-operator` | `partial-function` | `function`

⟨*binding*⟩ ::= `(`⟨*ident*⟩ ⟨*class*⟩`)`

⟨*case*⟩ ::= `(`⟨*formula*⟩ ⟨*class*⟩`)`

⟨*type-list*⟩ ::= ⟨*class*⟩ [`and`⟨*class*⟩]*

⟨*class*⟩ ::= `(either` ⟨*class*⟩*`)` | `(both` ⟨*class*⟩$^+$`)` | `(except`⟨*class*⟩ ⟨*class*⟩`)`
| `(if` ⟨*formula*⟩ ⟨*class*⟩ ⟨*class*⟩`)` | `(when` ⟨*formula*⟩ ⟨*class*⟩`)`
| `(cond` ⟨*case*⟩$^+$`)`
| `(`⟨*article*⟩ ⟨*class*⟩$^+$`)` | `(`⟨*article*⟩⟨*unary-impredicative*⟩ ⟨*class*⟩`)`
| `(`⟨*article*⟩ ⟨*binary-impredicative*⟩ `from` ⟨*type-list*⟩ `to` ⟨*class*⟩`)`

| | `(the-set` ⟨*class*⟩`)` | `(the-type` ⟨*class*⟩`)`
| | `(`⟨*article*⟩ `cons` ⟨*class*⟩ ⟨*class*⟩`)`
| | `thing` | `set` | `type` | `symbol` | `operator` | `cons-cell` | `integer`
| | `(some-such-that` ⟨*ident*⟩ ⟨*class*⟩ ⟨*formula*⟩`)`
| | `(lambda (`⟨*binding*⟩*`)` ⟨*class*⟩`)`
| | `(let (`⟨*binding*⟩*`)` ⟨*class*⟩`)`
| | `(`⟨*article*⟩ `wishful-version` ⟨*class*⟩`)`
| | `'`⟨*symbol*⟩
| | `(fail)`

A.1.2 Formula Expressions

⟨*formula*⟩ ::= `(there-exists` ⟨*class*⟩`)` | `(at-most-one` ⟨*class*⟩`)`
| | `(singleton` ⟨*class*⟩`)`
| | `(is` ⟨*class*⟩ ⟨*class*⟩`)` | `(is-never` ⟨*class*⟩ ⟨*class*⟩`)`
| | `(implies` ⟨*formula*⟩ ⟨*formula*⟩`)` | `(iff` ⟨*formula*⟩ ⟨*formula*⟩`)`
| | `(=` ⟨*formula*⟩ ⟨*formula*⟩`)`
| | `(not` ⟨*formula*⟩`)`
| | `(small` ⟨*class*⟩`)` | `(small-operator` ⟨*class*⟩`)`
| | `(operator-class` ⟨*class*⟩`)` | `(set-class` ⟨*class*⟩`)`
| | `(type-class` ⟨*class*⟩`)` | `(cons-cell-class` ⟨*class*⟩`)`
| | `(subrelation` ⟨*class*⟩ ⟨*class*⟩`)`
| | `(exists (`⟨*binding*⟩*`)` ⟨*formula*⟩`)`
| | `(forall (`⟨*binding*⟩*`)` ⟨*formula*⟩`)`
| | `(true)` | `(false)`

A.1.3 Definitions

Ontic definitions are used to define a symbol to denote a singleton Ontic class object. These can take two primary forms:

⟨*arglist*⟩ ::= `(`⟨*ident*⟩ ⟨*class*⟩`)` ⟨*arglist*⟩ | ⟨*empty*⟩

⟨*definition*⟩ ::= `(define-constant` ⟨*symbol*⟩ ⟨*class*⟩`)`
| `(define (` ⟨*article*⟩ ⟨*symbol*⟩ ⟨*arglist*⟩`)` ⟨*class*⟩`)`
| `(define` ⟨*symbol*⟩ ⟨*class*⟩`)`

A.2 Ontic Proof Forms

Ontic proof forms can be divided into two categories, user proof forms and non-user proof forms. The user proof forms correspond to the proof constructs typically used by the user when interacting with Ontic. Non-user proofs forms include second-try tactics and proof forms defined to implement control predicates. A brief grammar of the user proof forms is given below:

⟨*proof*⟩ ::= `(show` ⟨*formula*⟩ ⟨*proof*⟩*`)`
| `(suppose` ⟨*formula*⟩ ⟨*proof*⟩*`)`
| `(suppose-there-is (`⟨*binding*⟩*`)` ⟨*proof*⟩$^+$`)`
| `(suppose-there-is (`⟨*binding*⟩*`)` such-that ⟨*formula*⟩ ⟨*proof*⟩$^+$`)`
| `(suppose-not` ⟨*proof*⟩*`)`
| `(show-by-induction-on (`⟨*binding*⟩$^+$`)` ⟨*formula*⟩ ⟨*proof*⟩*`)`
| `(suppose-for-refutation` ⟨*formula*⟩ ⟨*proof*⟩*`)`
| ⟨*non-user-proof*⟩

In general, none of the subproofs in the user proofs include any non-user proof form. However, Ontic tactics can generate proofs whose top-level construct is a user proof form, while using non-user proof forms in the body.

B. PDDL DOMAIN DEFINITIONS

B.1 Blocksworld Domain

```
(define (domain blocksworld)
  (:requirements :strips :typing)
  (:types block)
  (:predicates (on ?x - block ?y - block)
       (ontable ?x - block)
       (clear ?x - block)
       (handempty)
       (holding ?x - block)
       )

  (:action pickup
     :parameters (?x - block)
     :precondition (and (clear ?x) (ontable ?x) (handempty))
     :effect
     (and (not (ontable ?x)) (not (clear ?x))
                  (not (handempty)) (holding ?x)))

  (:action putdown
     :parameters (?x - block)
     :precondition (holding ?x)
     :effect
     (and (not (holding ?x)) (clear ?x)
                  (handempty) (ontable ?x)))

  (:action stack
     :parameters (?x - block ?y - block)
     :precondition (and (holding ?x) (clear ?y))
     :effect
```

```
    (and (not (holding ?x)) (not (clear ?y))
            (clear ?x) (handempty) (on ?x ?y)))

  (:action unstack
     :parameters (?x - block ?y - block)
     :precondition (and (on ?x ?y) (clear ?x) (handempty))
     :effect
     (and (holding ?x) (clear ?y)
              (not (clear ?x)) (not (handempty)) (not (on ?x ?y)))))
```

B.2 Simplified Logistics Domain

```
(define (domain logistics)
  (:requirements :strips :typing)
  (:types truck
          package
          truck - physobj
          location - place)

  (:predicates  (at ?obj - physobj ?loc - place)
          (in ?pkg - package ?truck - truck))

(:action load
   :parameters    (?pkg - package ?truck - truck ?loc - place)
   :precondition  (and (at ?truck ?loc) (at ?pkg ?loc))
   :effect        (and (not (at ?pkg ?loc)) (in ?pkg ?truck)))

(:action unload
  :parameters   (?pkg - package ?truck - truck ?loc - place)
  :precondition (and (at ?truck ?loc) (in ?pkg ?truck))
  :effect       (and (not (in ?pkg ?truck)) (at ?pkg ?loc)))

(:action drive
  :parameters (?truck - truck ?loc-from - place ?loc-to - place)
```

```
  :precondition
   (at ?truck ?loc-from)
  :effect
   (and (not (at ?truck ?loc-from)) (at ?truck ?loc-to)))
)
```

C. PREDICATE-ACHIEVEMENT MACRO AND GENERALIZED PLAN DEFINITIONS

C.1 Blocksworld Domain

```
;;macro to make an arbitrary block clear
(define (the make-clear (s (a legal-bw-state)) (b (a block-in-state s)))
  (if (there-exists (a held-block s))
    (the make-clear (the result (the putdown (a held-block s)) s) b)
    (if (not (there-exists (a (the on-relation s) b)))
      s
      (the result (the putdown (a (the on-relation s) b))
                  (the result (the unstack (a (the on-relation s) b) b)
                              (the make-clear s (a (the on-relation s) b)))))))

;;macro to place an arbitrary block on the table
(define (the make-ontable (s (a legal-bw-state)) (b (a block-in-state s)))
  (if (is b (an ontable-block s))
      s
      (if (is b (a held-block s))
          (the result (the putdown b) s)
          (let ((s1 (the make-clear s b)))
            (the result (the putdown b)
            (the result (the unstack b
                        (a (the inverse (the on-relation s1)) b))
                        s1))))))

;;macro to make an arbitrary block held
(define (the make-held (s (a legal-bw-state)) (b (a block-in-state s)))
  (if (is b (a held-block s))
      s
      (let ((s1 (the make-clear s b)))
```

```
        (if (is b (an ontable-block s1))
            (the result (the pickup b) s1)
            (the result (the unstack b
                        (a (the inverse (the on-relation s1)) b)) s1)))))

;;macro to achieve an arbitrary on fact
(define (the make-on (s (a legal-bw-state))
                     (b1 (a block-in-state s))
                     (b2 (a block-in-state s) such-that (not (= b1 b2))))
      (the result (the stack b1 b2)
                  (the make-held (the make-clear s b2) b1)))

;;block-correctly-on is an operator whose range is the
;;class of blocks that are both on the block in the current
;;state and in the goal
(define (a block-correctly-on (s (a legal-bw-state))
                              (g (a valid-goal-for s)))
  (lambda ((x (a block-in-state s)))
      (both (a (the on-relation s) x)
            (a (the on-relation g) x))))

;;correctly-solved-block - true when block is correctly-on down
;;all the way to the table
(define (a correctly-solved-block (s (a legal-bw-state))
                                  (g (a valid-goal-for s)))
  (either (both (an ontable-block g) (an ontable-block s))
          (a block-correctly-on s g (a correctly-solved-block s g))))

;;generalized plan to achive any conjunctive blocksworld goal
(define (the genplan (s (a legal-bw-state)) (g (a valid-goal-for s)))
  (if (there-exists (except (an ontable-block g) (an ontable-block s)))
      (let ((b (except (an ontable-block g) (an ontable-block s))))
        (the genplan (the make-ontable s b) g))
    (if (there-exists (some-such-that b
                        (a correctly-solved-block s g)
```

```
                          (not (= (a (the on-relation s) b)
                                  (a (the on-relation g) b)))))
        (let ((b (some-such-that b
                     (a correctly-solved-block s g)
                        (not (= (a (the on-relation s) b)
                                (a (the on-relation g) b))))))
          (if (not (there-exists (a (the on-relation g) b)))
              (the genplan (the make-clear s b) g)
            (the genplan (the make-on s (a (the on-relation g) b) b) g)))
      (if (there-exists (a held-block g))
          (the make-held s (a held-block g))
        s))))

;;generalized plan to get all the blocks on the table
(define (the all-ontable (s (a legal-bw-state)))
  (if (is (a block-in-state s) (an ontable-block s))
      s
      (let ((b (except (a block-in-state s) (an ontable-block s))))
        (the all-ontable (the make-ontable s b)))))
```

C.2 Simplified Logistics Domain

```
;;macro to move a truck to a specified location
(define (the make-at-truck (s (a legal-log-state))
                           (t (a truck-in-state s))
                           (l (a location-in-state s)))
  (if (is t (a (the at-relation s) l))
      s
      (the result (the drive t (a (a location-in s) t) l) s)))

;;macro to get an arbitrary package into an arbitrary truck
(define (the make-in (s (a legal-log-state))
                     (p (a package-in-state s))
                     (t (a truck-in-state s)))
  (if (is p (a (a package-in s) t))
      s
      (if (is p (a (a package-in s) (a truck-in-state s)))
          (the make-in (the result (the unload p (a (a container-in s) p)
                                       (a (a location-in s) (a (a container-in s) p)))
                               s)
                   p
                   t)
          (if (= (a (a location-in s) p)
                 (a (location-in s) t))
              (the result (the load p t (a (a location-in s) p)) s)
              (the result (load p t (a (a location-in s) p))
                    (the result (the drive t
                                     (a (a location-in s) t)
                                     (a (a location-in s) p))
                               s))))))

;;macro to get an arbitrary package to an arbitrary location
(define (the make-at-package (s (a legal-log-state))
                             (p (a package-in-state s))
                             (l (a location-in-state s)))
```

```
  (if (is p (a (the at-relation s) l))
      s
      (let ((t (a truck-in-state s)))
        (the result  (the unload p t l)
              (the make-at-truck
               (the make-in s p t)
               t l)))))

;;package-correctly-at relation - packages that are in the
;;correct goal location in the current state
(define (a package-correctly-at (s (a legal-log-state))
                                (g (a valid-delivery-goal-for s)))
  (lambda ((l (a location-in-state s)))
    (both (a package-in-state s)
          (both (a (the at-relation s) l)
                (a (the at-relation g) l)))))

;;delivered package is one that is at its desired
;;goal location
(define (a package-delivered-in (s (a legal-log-state))
                                (g (a valid-delivery-goal-for s)))
  (a package-correctly-at s g (a location-in-state g)))

;;generalized plan to deliver all packages to their destination
(define (the deliver-all (s (a legal-log-state))
                         (g (a valid-delivery-goal-for s)))
  (if (is (a package-in-state g) (a package-delivered-in s g))
      s
      (let ((p (except (a package-in-state g)
                       (a package-delivered-in s g))))
        (the deliver-all (make-at-package s p (a (the location-in g) p)) g))))
```

VITA

VITA

Rajesh Kalyanam grew up in Bangalore, India where he attended the National Public School from kindergarten through grade twelve. He then received a Bachelor's degree in Computer Science and Engineering from the National Institute of Technology, Karnataka. After working as an Application Developer at Oracle, India for three years, he embarked on his graduate studies at Purdue University.