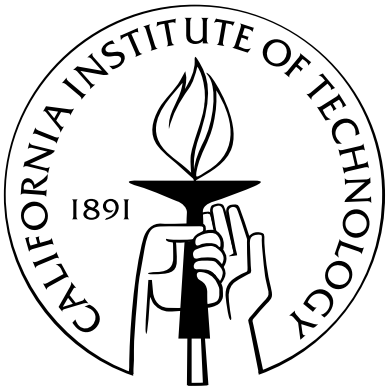
Kind Theory

Thesis by

Joseph R. Kiniry

In Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy



California Institute of Technology
Pasadena, California

2002
(Defended 10 May 2002)

Preface

This thesis describes a theory for representing, manipulating, and reasoning about structured pieces of knowledge in open collaborative systems.

The theory's design is motivated by both its general model as well as its target user community. Its model is structured information, with emphasis on classification, relative structure, equivalence, and interpretation. Its user community is meant to be non-mathematicians and non-computer scientists that might use the theory via computational tool support once integrated with modern design and development tools.

This thesis discusses a new logic called *kind theory* that meets these challenges. The core of the work is based in logic, type theory, and universal algebras. The theory is shown to be efficiently implementable, and several parts of a full realization have already been constructed and are reviewed.

Additionally, several software engineering concepts, tools, and technologies have been constructed that take advantage of this theoretical framework. These constructs are discussed as well, from the perspectives of general software engineering and applied formal methods.

Acknowledgements

I am grateful to my initial primary adviser, Prof. K. Mani Chandy, for bringing me to Caltech and his willingness to let me explore many unfamiliar research fields of my own choosing. I am also appreciative of my second adviser, Prof. Jason Hickey, for his support, encouragement, feedback, and patience through the later years of my work. If Jason had not appeared at Caltech in Autumn of 1999, I may well have not finished my Ph.D.

I am very much in debt to Joseph Goguen whose inspiring work started me on the path of using algebras and categories. José Meseguer and Francisco (Paco) Duran have been of tremendous help and inspiration in my use of Maude and rewriting logic.

Thanks also go to fellow graduate students and research group members Daniel Zimmerman, Eve Schooler, Michel Charpentier, Roman Ginis, Paolo Sivilotti, John Thornley, and Berna Massingill, and Adam Rifkin—the original reason for my coming to Caltech.

I am one that would loose my head if it wasn't screwed on at times, so administrative assistance is appreciated. To that end, I am in debt to Diane Goodfellow and Jeri Chittum, two of the key folks that keep Caltech Computer Science running.

I am most grateful to the Caltech Computer Science community for providing such an open, stimulating, and quality environment in which to do "way-out" work. Influential members of my community are, besides those fellow students mentioned above: Eric Bax, Eitan Grinspun, Rajit Manohar, Alexander Nicholson, Mika Nyström, Steven Schkolne, and Zoë Wood.

Life is not complete without friends, and I am lucky enough to have too many to list. Of particular importance over these past five years are Chara Williams and Mary Baxter. A large cadre of best friends, most nonlocal, have kept me sane, healthy, and high-spirited over the years: John Greene, Dara Thompson, Amy Wilkinson, Cici Koenig, Cynthia Fay, Patricia Wong, Robert Pastor, Robert Stacy, and the BoDO gang: Mark Baker, Nelson Minar, and last but not least, Ron Resnick.

Academic encouragement came from many fronts. First and foremost I thank Doug Lea for his subtle background influence in furthering my academic career—you are my invisible adviser. I also thank my various academic advisers over the years: Gregory Riccardi, David Kopriva, Steven Leach, and Charles Weems. Thanks also go to John Thornley for providing

remote encouragement when I needed it most and Eric Klavins for filling in as a last-second committee member.

Several teachers, instructors, and professors have had an enormous impact on me over the years. I think this is unavoidable for someone who has lived through five degrees, three universities, and numerous research stints at various places. I'd like to acknowledge the following individuals, some of whom might remember me with a smile, and others with a cringe. From Ft. Myers High School: Janet Marderness, Bruce Conover, and Barbara Kepler. From Florida State University: Theodore Baker, Paolo Aluffi, Steven Bellenot, Jim Gaunt, Robert Gilmer, Sam Huckaba, David Loper, Joe Mott, Daniel Oberlin, and Euliquio Young. From the University of Massachusetts at Amherst: James Kurose, Eliot Moss, Jack Wileden, Steven Cook, and Teresa Kellogg. From the Open Software Foundation: John Bowe, Murray Mazer, Ira Goldstein, and Ron Rebeiro.

Finally, I'd like to thank my family for all of their support, encouragement, faith, and love.

Abstract

My contribution, described in this thesis, is a theory that is meant to assist in the construction of complex software systems. I propose a notion of structure that is independent of language, formalism, or problem domain. I call this new abstraction a *kind*, and its related formal system, *kind theory*. I define a type system that models the structural aspects of kind theory. I also define an algebra that models this type system and provides a logic in which one can specify and execute computations.

A reflective definition of kind theory is reviewed. This reflective specification depends upon a basic ontology for mathematics. By specifying the theory in itself, I provide an example of how one can use kind theory to reason about reuse in general formal systems.

I provide examples of the use of kind theory in reasoning about software constructs in several domains of software engineering. I also discuss a set of software tools that I have constructed that realize or use kind theory.

A logical framework is used to specify a type theoretic and algebraic model for the theory. Using this basic theorem prover one can reason about software systems using kind theory. Also, I have constructed a reuse repository that supports online collaboration, houses software assets, helps search for components that match specifications, and more. This repository is designed to use kind theory (via the logical framework) for the representation of, and reasoning about, software assets.

Finally, I propose a set of language-independent specification constructs called *semantic properties* which have a semantics specified in kind theory. I show several uses of these constructs, all of which center on reasoning about reusable component-based software, by giving examples of how these constructs are applied to programming and specification languages. I discuss how the availability of these constructs and the associated theory impact the software development process.

Notation

Nearly all mathematical presentation constructs (e.g., theorems, axioms, etc.) are independently numbered. The only exception is corollaries which are numbered according to their associated theorem.

Proofs are denoted by a leading **Proof.** or **Proof of Theorem Title** and are ended by a box. □

Logical rules are written in the standard style:

Modus Ponens (MP)

$$\frac{U \quad U \Rightarrow V}{V}$$

A rule named with an asterisk is reversible and summarizes two rules. Variables in rules are written in the default font if they represent arbitrary free entities.

The basic Boolean values of **true** and **false** are denoted as $\top$ and $\bot$ respectively. The set of Boolean values is denoted by the symbol $\mathbb{B} = \{\top, \bot\}$. We denote a ternary truth value of **unknown** with the symbol ?. The set of ternary truth values is specified by the symbol $\mathbb{K} = \{\top, \bot, ?\}$.

Arbitrary sets are written as $A, B, C, \ldots$ in italics. Algebraic sorts are written in the typeface `Boolean`, `Int`, `Float`, ... and types are written in the typeface Boolean, Integer, Float, ... Models are written in calligraphic font $\mathcal{M}, \mathcal{N}, \mathcal{O}, \ldots$. Kinds are written as POSET, UNIVERSAL, LIST, ... and instances use the standard font I, J, ...

Functions are named with lowercase roman and Greek letters or basic symbols as in $f, \gamma, \ldots$; choice of symbol will be made clear in the presentation. Functions are written inline in the typical fashions: $<_p$: KIND → KIND or, occasionally, KIND $\overset{<_p}{\rightarrow}$ KIND.

The identity function on an object X is $id_X : X \to X$, or $id : X \to X$. Composition of functions $f : X \to Y$ and $g : Y \to Z$ is written $g \cdot f$.

Contents

List of Figures

List of Tables

The Extension of an universal Idea regards all the particular Kinds and single Beings that are contained under it. So a Bowl, in its Extension, includes a wooden Bowl, a brass Bowl, etc.—from *Watts Logic I*, iii, Section 3, 1725

The understanding seems to me not to have the least glimmering of any *ideas* which it doth not receive from one of these two. *External objects furnish the mind with the* ***ideas*** *of sensible qualities*, which are all those different perceptions they produce in us; and the *mind furnishes the understanding with* ***ideas*** *of its own operations.* —John Locke, *Essay Concerning Human Understanding*, Book II, Chapter I, Section 5 [247] Emphasis in original.

Nigel Tufnel: The numbers all go to eleven. Look, right across the board, eleven, eleven, eleven and...
Marty DiBergi: Oh, I see. And most amps go up to ten?
Nigel: Exactly.
Marty: Does that mean it's louder? Is it any louder?
Nigel: Well, it's one louder, isn't it? It's not ten. You see, most blokes, you know, will be playing at ten. You're on ten here, all the way up, all the way up, all the way up, you're on ten on your guitar. Where can you go from there? Where?
Marty: I don't know.
Nigel Tufnel: Nowhere. Exactly. What we do is, if we need that extra push over the cliff, you know what we do?
Marty: Put it up to eleven.
Nigel: Eleven. Exactly. One louder.
Marty: Why don't you just make ten louder and make ten be the top number and make that a little louder?
Nigel: [Pause] These go to eleven.
— "This is Spinal Tap", 1984.

Chapter 1

Introduction

1.1 Motivation

Software systems are more complicated today than at any point in the history of computing. Projects consisting of millions of lines of program code written in a dozen languages developed and maintained by hundreds of people across multiple companies are commonplace.

The development of software systems, realized by the practice of software engineering, is primarily characterized by its complexity and variety. There are hundreds of languages, tools, and technologies to choose from for any given domain. The field of software engineering research has similar variety as there are an enormous number of formalisms, methodologies, and approaches.

Software is a young discipline, so it is unsurprising that this state of affairs is similar to that which more mature fields like manufacturing faced many years ago. Until the development of standard measures, tools, and terminology, manufacturing was a cottage industry. Independent manufactures created their custom wares in their own manner and with their own tools and parts. The result of which often showed poor workmanship, irregularities, inconsistency, and incompatibility. Such can be said for software today.

The desire for an "a la carte" approach to software engineering is often expressed by researchers and practitioners. Developers want the ability to pick and choose formalisms, tools, and technologies for each new problem. But the current state of affairs, characterized by fiefdoms of theory and technology, does not lend itself to this approach.

One way to help realize this goal is to provide a kind of bridging formalism. Such a theory might help software engineers express the commonalities and differences between the various options available. If this information were to be collected from the software community and made publicly available, perhaps via a Web site, then the community as a whole would benefit.

Describing the commonalities and differences between the various theory, tools, and technologies of software engineering can be accomplished in many ways. Several abstract branches

of mathematics have been developed specifically to describe and reason about such structures (e.g., category theory, type theory, algebras). But these existing options are overly general, as they do not focus on the structures important to modern software systems. What is needed is a new logical system that focuses on the structures specific to software, but is general enough to cover the wide variety of structures involved in complex systems development.

The goal of this work is to develop an integrated set of theories and software tools for the purpose of helping software engineers describe, construct, and reason about their complex software systems.

1.2 Aspects of Software

Deriving a new theory involves many choices and trade-offs. These decisions are primarily driven by the problem domain: component- and object-oriented, concurrent and distributed software systems, with a focus on program correctness.

A large part of software engineering centers on analysis, design, and development. Specification and programming languages coupled with processes and methodologies are some of the primary facets of this domain. The major aspects of the theory, tools, and technologies for these domains drive the basic structure of this work.

1.2.1 Core Notions Derived from Software

Specification languages are used to express the desired properties of a software system. Most have a declarative and formal foundation, often a basic branch of mathematics like set theory or predicate calculus. As they are languages, they have a syntax, semantics, and some model. That model is often a domain of application, and the design of the language mirrors that domain.

Programming languages are used to express computation. They also have syntax, semantics, and a model, but are more expressive than specification languages because most are not declarative. Modern programming languages used in complex software development, exemplified by languages like Java and C++, are often object-oriented. Thus, the common structures of object-oriented systems are of interest in this work.

Programming languages embody computation, and computation is abstractly represented by functions. Most specification languages are declarative and focus on predicates (which are relations). As a result, functions and relations are necessary structures for describing and reasoning about the structures inherent in specification and programming languages.

Code and specification reuse are often represented by various notions of inheritance. Inheritance comes in many forms. Some focus on classification, while others center on information

sharing [263]. Classification is a notion of abstraction, but abstraction is not useful without some connection to the real world when building software systems. Thus some dual notion of thing which is classified, or what is called an instance, or realization, is necessary.

The standard manner in which complex systems are simplified is through decomposition. Breaking down a complicated construct into smaller, less complex pieces is the hallmark of structured problem solving. Systems are structured by and for this problem solving technique in several ways, the primary options of which are layering and modularization. In either case, the basic units of construction are parts which are assembled to produce wholes, which in turn are reused as parts. Thus, part-whole relationships are fundamental notions.

Operations that permit pieces to be assembled and disassembled seem to be fundamental as well, as solving the complex problems inherent in complex systems development necessitate problem decomposition, and thus composition. Therefore, if parts and wholes are to be assembled and disassembled, some basic notions of composition and decomposition are necessary.

The ability to analyze the similarities and differences between constructs means that operations that center on such comparisons are central to this work. Similarity is characterized by equivalence relations, and dissimilarity by difference operators, like subtraction and intersection. Additionally, to compare the closeness of constructs, some type of ordering of similarity is also necessary.

This completes a summary of basic operators necessary to a general logic of structure for software: functions and relations, inheritance and realization, part-whole relationships, composition and decomposition, equivalence and ordering.

But software systems are about more than objective pieces of reusable assets. People are a fundamental part of building software, and people are notoriously subjective, so capturing subjectivity is a necessary component of this work.

1.2.2 Subjectivity in Software

Many of the decisions made about software are subjective. There is no right answer to every question. Different developers have their favorite theories, tools, and technologies. Background, history, education, job experience, and more impact the decisions and justifications that go along with every step of the development process.

Differentiating the objective from the subjective is a difficult topic that has been analyzed by researchers in many domains. The primary classifications for such distinctions come from within the realms of philosophy (as studies of knowledge, or what is commonly known as epistemology) and linguistics (the study of language) [20, 66, 67, 174, 387].

Within these disciplines, objective statements are called claims, and subjective statements are termed beliefs [20, 66, 174]. Therefore, because software development involves people acting in subjective ways, claims and beliefs are used to classify all statements about software and its structures.

1.2.3 Aspects of Information

If a software asset is to be made available for others to use, it must be published in some fashion. Additionally, for interested parties to find published reusable software assets, it must be part of a searchable repository or archive.

Research in the field of software reuse has focused on the problems of publication and search [42, 43, 273]. Repositories of software constructs (code, design, documentation, etc.) have been built using a variety of technologies and formalisms [230, 357, 374, 375]. Nearly all of these solutions focus solely on characterizing assets for storage and search. They do not provide solutions for the related problems like the maintenance of assets and communication within, and feedback from, the community of reusers.

Technology for making software available for use and reuse has evolved since the birth of the web. While new means of publication and search become available every day, the current venues and mechanisms for the publication of software artifacts do little toward solving the problems of specifying and reasoning about structure.

XML provides a generic logic for reasoning about syntax. Generic computation on such structures, realized by XML-centric standards and technologies like those of the W3C and the Apache Foundation, provide powerful and flexible computational infrastructures for reasoning about syntax. But there is little work on developing semantics for such systems, especially in open, collaborative environments.

Several Web sites exhibit initial attempts toward solving some of these problems, particularly in the domain of Open Source software. Online communities like Slashdot, Web publication and collaboration forums like Freshmeat[1], SourceForge[2], the Giant Java Tree[3], and Flashline[4], and well-organized projects like the Apache Foundation[5] are examples. All are quality systems from which to learn what works in open, collaborative software development.

[1] http://www.freshmeat.net/
[2] http://www.sf.net/
[3] http://www.gjt.org/
[4] http://www.flashline.com/
[5] http://www.apache.org/

1.3 Related Work

Work from several communities is related to this work given its broad nature. In particular, the reuse, software engineering, and knowledge-related communities have had a great deal to say about related topics.

1.3.1 General Classification Theories

The earliest work on classification and categorization, that of Plato, Aristotle, and especially Leibniz was non-mathematical in nature [18, 19, 236, 237, 309]. All proposed various notions of classifiers; e.g., Plato's ideas, forms, and signs, Aristotle's genera, species, and telos, and Leibniz's monads, and instances of classifiers. Some work focused also on composition (aggregation) and equivalence, but the universality of these early theories is confounded by the lack of clarity that accompanies a non-formal discussion.

The field of semiotics, while a more modern discipline, is equally full of ambiguities [107, 305]. It has not been widely adopted within the computational community because of its focus on the behavior of living systems coupled with a lack of formality.

Mathematical approaches have come from two primary directions. One approach formalizes semiotics using either lattice theory or universal algebras.

The former is witnessed in the field of concept analysis which uses fixed but extensible properties and lattices to classify, sort, and reason about entities [136, 235, 377, 378, 379]. Early work exists on building logical generalization of such lattice-based structures [117]. Work on ontology construction and knowledge sharing using conceptual analysis has advanced rapidly with the emergence of the Web as a general knowledge sharing venue [145, 175, 292].

Some of this work has started to be applied to software systems. For example, Wendorff constructs terminological bridges for software classification constructs in a manner similar to the use of interpretive equivalences in kind theory [371]. Unfortunately, such work rarely has a sufficiently complete formal characterization to reason about the (non-first-order, autoepistemic) interpretations necessary for kind theory [13]. Additionally, concept lattices have been applied to the classification of object-oriented software artifacts, primarily for search and maintenance [143, 185, 376].

This work has also begun to impact programming languages design, particular for logic-based languages [62]. Such an approach is similar to that which is proposed in Section 8.2.6.

The second approach is the new field of algebraic semiotics, pioneered by Goguen [155]. This work has primarily been applied to user interface design, especially the presentation of mathematical proofs and the characterization of the interface of proof tools [152, 153, 160, 163, 254].

Unfortunately, none of this work, either lattice theory or algebraic-based, deals with subjective truths or autoepistemic knowledge. None of it has been applied to software constructs of granularity other than classes, and none has been applied to the constructs other than program code. Finally, very few systems that realize these theories have been built and none have been broadly tested.

1.3.2 Knowledge Theories

There is no theory of knowledge, or what is sometimes called the philosophy of knowledge or epistemology. There is a debate about the theory of knowledge, a debate that has been ongoing for over 2,000 years.

A historical study of the works of philosophers that are interested in this debate reveals the companion label theory of reality, usually termed metaphysics. This branch concerns itself with questions about the nature of the world and of man. While most view this branch as the more fundamental of the two, any explicit discussion of it will be avoided.

Implicitly, on the other hand, it must be recognized that kind theory engenders the process of metaphysics. Aristotle called it the study of "being *qua* being," that is, the study of what there is and of the properties of what there is. Collaboratively exploring knowledge, recording its structure, building ontologies, capturing experience—this is all collaborative constructive metaphysics.

Without some philosophical and mathematical backing, the metaphysician creating and recording knowledge is without a foundation and does not command attention. Identification, authentication, and credentials are necessary to enable judgments of the worth of what has been said.

If Sara says, "I know that this algorithm is of computational complexity $O(n)$," she is advancing a claim for which you expect me to have reasons which she should be able to defend and explain. Such a claim is stronger than the statement "I claim that this algorithm is of computational complexity $O(n)$," though this belief will also be backed by some defense.

The theory of knowledge is in part interested with the relations between claims and reasons for claiming. This aspect is often called the normative or justificatory aspects of the theory of knowledge. To know requires not only that one has reasons which will justify one's knowing and one's claim; it requires an awareness, that one has perceived, understood, inferred, weighed evidence, etc. The analysis of these various sorts of mental operations—of perceiving, understanding, inferring, etc.—compromises what is called descriptive epistemology. Kind theory's truth structures are exactly these relations between a claimant and predicate.

1.3.2.1 Conceptual Spaces

Gärdenfors's theory of representation called conceptual spaces has some similarities to this work[137]. Conceptual spaces are a metatheory for cognitive science that uses geometric representation to model concept formation, semantics, nonmonotonic inference, and inductive reasoning. These connections are not surprising if an eye is cast broadly enough. Given the ties between kind theory and structure-centric theories like category theory (see below) which has its historical foundation in algebraic geometry, reinterpreting the concepts and semantics of kind theory into a geometric-centric logical framework is sure to result in something similar to concept spaces.

Conceptual spaces distinguish three levels of representation: the subconceptual, the conceptual, and the symbolic. Two key ideas, properties and concepts, make up the core of the theory. Properties are assigned to objects as a way of abstracting away redundant information about objects. Thus, if objects are classified as kind, properties are inclusion substructures.

The work on conceptual spaces is still in its relative infancy. The mathematics is tentative and light, and no application to software systems has been made.

1.3.3 General Theoretical Foundations

1.3.3.1 Category Theory

Certain aspects of (higher-order) categorical logic bear a similarity to the notion of kind [224]. The fact that categories focus on structure under isomorphism complements the notion of structure under equivalence. Structure-preserving notions within a categorical context are functors and natural transformations, and the general notion of truth-preserving maps is inherent in toposes.

Additionally, the identification of classifiers is intricately tied up with fibred and indexed categories [194]. Initial investigations into using category theory to specify kind theory in an extra-logic fashion led to Cartesian Closed Categories, so it is clear there are numerous connections to investigate and elaborate.

The Kestrel Institute's Specware system, which is based upon category theory, provides support for integrated software and knowledge engineering [183, 189, 345, 380]. This technology focuses on the use of categorical constructs for the representation and validation of composition, but has no support for open, collaborating systems.

1.3.3.2 Model Theory

On a similar note, model theory, often called the logic of syntactic structure, also has connections and application. The ability to push away from any precise foundation and focus on structure through the use of model theory is an attractive option [59, 170, 257, 310, 360, 370], or computation via domain theory [176, 285]. There is also work applying model theory to the problem of knowledge representation [115].

Metamathematical structure-centric notions like intuitions can also help avoid logical bias [21, 156, 157, 158, 282]. Institutions are the most closely related theoretical formulation to this research.

Institutions provide a model-independent way of talking about semantics, much like kinds do. And institution morphisms are semantics-preserving functions, as interpretations are. If this work were recast into institutions, it would be a kind of institutional higher-order logic of semiotics.

We chose not to use institutions for the current work not because it is not an appropriate formalism, but because the technical costs (in learning, applying, and implementing) such a formalism are so high. We discuss future work regarding institutions more in Section 8.3.

1.3.3.3 Paraconsistent and Other Deviant Logics

Several issues drive this work out of classical logic into the realm of what some call deviant logics [178, 179]. Truth values assigned by claims and beliefs have more than two values, thus non-classical logics like 3-valued [213] and many-valued logics, especially those adopted by the computation community like those of Blikle [220], should be considered. Fuzzy, modal, or non-monotonic logic are also legitimate models for this work [358].

This section heavily uses [361] as a historical framework for paraconsistent logics.

The field of paraconsistent logics has seen self-organization and rapid growth over the past fifteen years. The origins of these non-classical logics go back to Aristotle in his early questioning of the principle of excluded contradiction. His analysis claimed that this principle, together with tertium non datur, were the most certain of all principles. While tertium non datur has not been especially durable over the years, the principle of excluded contradiction has rules almost undisputed in logic for the next two thousand years.

The concept of inconsistency is still under debate. Besides logical contradictions there are a large number of related notions including refutations, incoherences, antimonies, etc. From a philosophical perspective, the question must be asked, are inconsistencies sometimes true descriptions of phenomena in nature, or are they only an epistemic construct? The first position is called strong paraconsistency; the second is referred to as weak paraconsistency.

The literature of paraconsistent logics is still relatively small, on the order of a few hundred articles to date. Applications in mathematics are the most prominent examples for the advantages of paraconsistent logics. Robert Meyer's relevant arithmetic $R^\sharp$ was the first example [270], followed by a whole class of inconsistent arithmetical theories [271]. The protagonists for a paraconsistent foundation of mathematics may call on prominent advocates, namely Wittgenstein who defended this view in the early 1930s [382, 383].

Paraconsistent logics have been used a great deal in the field of artificial intelligence, particularly in automated reasoning. One of the earliest theorem provers was called Kripke [354]. It implemented the relevant logic LR (R without distribution of $\wedge$ over $\vee$). A summary of the recent work in this field is contained in the second volume of the Handbook of Defeasible Reasoning book series [127, 128, 129, 130, 131], particularly in [39] and related work [108].

Two options for handling inconsistency are highlighted in these works. The actual contradictions view investigates inference mechanisms designed to cope with actual inconsistencies in a non-trivial way. The potential contradictions view assumes some mechanism for resolving conflicts, mostly in the field of defeasible reasoning. Both perspectives occur in artificial intelligence in proof-theoretic and semantics-centric forms. A survey of almost twenty semantics for program inconsistency is presented by Damasio and Pereira [97].

In recent years, logic programming has been applied to topics such as updates and belief revision, particularly in multi-agent systems. The basic idea is that an agent must not only reason with inconsistent or partial information, but perform reasoning by taking such inconsistencies into account. The crucial point is a mechanism for explicitly declaring the falsity of propositions. Such an explicit non-classical negation, in addition to the usual default one, is available only recently.

Belief revision is the study of rationally revising bodies of belief in light of new evidence. Within the foundationalist approach to belief change there are two prevailing ways of handling revision of belief. The horizontal perspective considers consequences classically inferred from a basis of beliefs. Very sophisticated procedures have been developed in order to operate on your basis, or change an agent's context, in kind theoretic-terms. The vertical perspective has more natural context change operations coupled with a refine consequence operation for drawing inferences from that basis [124, 125].

As cited by leaders in the paraconsistent community, the most important paraconsistent logics are those of da Costa [96], many-valued logics [8, 249, 358], relevant logics [14], inconsistency adaptive logic [31, 32], and non-adjunctive systems [197]. These logics all share a common theme of differentiating classical from non-classical inference and some have explicit representation of agents that impose semantics. Thus, these shared features are adopted as core features of kind theory.

1.3.4 Knowledge Representation

Work in computational knowledge representation has some relations to this research. Standard languages and models have been developed, particularly within the domain of multi-agent systems. Communication and representation languages like KQML, CKML, and OML are used to represent agents' knowledge and goals [69, 120, 300, 299].

This work has influenced new developments in the domain of the world wide web, particularly early work in general resource specification (RDF) as well as new work in what is being called the semantic web [114, 228, 229]. Such work relates to this work, particularly with respect to language design for specifying semantics and exposing Web interfaces to such data. Little work has been done in these areas on (autoepistemic, paraconsistent, constructive) inferential theories for such data.

Recent standards-based work has begun build standardized upper-level (most abstract) ontologies with the intent of providing a firm foundation for the Semantic Web and related inferential, knowledge-based technologies [193]. Two of the primary proposed upper ontologies have a category theoretic and a model theoretic foundation [202, 350, 351].

All of these ontologies are excellent starting points for future collaborative ontology construction because their core operations map directly to kind theoretic ones, primarily inheritance and inclusion. Collaborative ontology revision has seen some work in both a digital library and algebraic contexts [36, 116, 175]. Such work provides guidelines for future work in ontology construction with kind theory but does not otherwise overlap with this research.

1.3.5 Software Reuse

1.3.5.1 Reuse Models

Will Tracz's 3C model can be though of as a lightweight theory of reusable components [230, 356, 357]. The 3Cs are context, concept, and content. It seems to be the dominant high-level component model of the late 80s and early 90s [374, 375, 384], its only contender being REBOOT[280, 339, 344] and the formal models of languages like CDL [140, 317, 375], CIDER [375] and Π [6, 94, 132, 133, 146].

The evolution of the 3C model is witnessed in work like that of Latour and Meadow [231]. The authors of this paper propose that the formal specification of reusable components needs to focus on what they call the normalization of components within a framework, a term that they adopt from the relational database world. This term and concept of normalization coincidentally agrees with the canonicalization operation in terminology and intent.

1.3.5.2 Reuse Theories

The work of Goguen and Tracz is the most closely related work in all of the field of software and reuse. Goguen's 1986 paper on reusing and interconnecting software components laid the groundwork for many of the concepts here including core operators, theoretical directions for semantics, and application to software technologies [151]. Goguen and Tracz later extended this work, refining the semantics and application of composition to complex Ada systems [166].

This early work was very exploratory. Many papers were written about what could be done with the theory and system, but no tools were ever made available for experimentation and use. The work was applied only to a single programming language (Ada), using a single specification language (OBJ3), and a single module interconnection language (LIL and its descendants).

This work continues very strongly in this tradition. It extends the notions of that work by using and extending the set of core operators. It has broader application because it defines only a basic logical layer in which to express and reason about semantics. It is applied to multiple programming and specification languages. Finally, several tools have been constructed and made available.

The area in which this work most differs from Goguen's early work is in the use of non-classical logics and explicit agent-based semantics. Neither of these domains were ever considered in the earlier work. Thus, this early work never incorporated any of the results from the fields of knowledge representation and engineering.

1.3.5.3 Reuse Technologies

The bulk of the work in the field of software reuse focuses on languages, tools, and systems that promote, encourage, or simplify reuse.

Many of the early reuse researchers were theorists, but their results have not seen widespread adoption over the past twenty years. One reason for this is that few tools with firm theoretical foundations were constructed and made available for use[6]. Another problem is that much of the early work focused on algebraic and categorical formalisms, both of which fell out of favor in the early 90s for a number of technical and social reasons.

Fundamental work in building reusable libraries has helped us understand what programming language, reuse, and classification concepts are most useful to software reuse [268]. The primary mechanisms that have been identified for such (e.g., parameterization, fundamental data types, etc.) all provide a fertile ground for specifying fundamental kind theoretical concepts for software [355].

[6] In fact, I have been unable to find a single example of such.

1.3.5.4 Asset Repositories

Nearly all software asset libraries provide access via one or more of three methods: browsing, informal searches, and formal searches. The most popular kinds of informal searches are keywords-based (Computer Aided Prototyping System [CAPS] [252], the Operation Support System [OSS]), multi-attribute queries based upon facets (CAPS, DRACO, RAPID, OSS, the Reusable Software Library [RSL], the Common Ada Missile Packages project [CAMP]), and natural language (RSL). CAPS also supports retrieval based upon formal specifications.

In CAPS, components are specified using a prototyping language called PSDL. Retrieval queries can also be written using partial specifications in PSDL.

All components are stored after being syntactic and semantically normalized. Normalization exclusively applies to PSDL specifications; syntactic normalization involves primarily format changes and statistical calculations, while semantic normalization involves specification expansion and transformations.

Search refinement is based upon the ability to drive gross searches (via facets, for example) via the transformation of much more formal searches (via PSDL). Various test cases and filters are generated to refine candidate sets until no more candidates can be eliminated. The final phase of search is based upon a set of automated theorem proving techniques that attempt to conclusively show that one of the retrieved components does satisfy the query specification. These are based on algebraic specification, term rewrite systems, and a fast but limited inference method [252].

Found components are automatically transformed to match query form. Potential transformations including changing parameter types and operator names as well as instantiating generics.

This work was extended significantly by Goguen, Luqi, Meseguer, and others in the mid-1990s [150]. That work extended the search engine's capability to handle semantic matches and better incremental filtering. Its application was significantly limited when compared to this work as it demanded that all components have a full algebraic specification and user queries were expressed in algebraic specifications or programming language fragments. Also, there were a number of limitations with respect to matching generic and non-generic modules, and no provision was made for module composition.

Some projects have focused on integrating reuse with knowledge-based design environments. The CATALOGEXPLORER system from the University of Colorado and Software Research Associates, Inc. explored this synthesis [287]. It supported retrieval from specification and construction, much like the explicit and implicit searches mentioned here, information hiding via something called hidden features, and specification-linking rules that support analogical

matching. Other similar environments include LaSSIE [98],

Reuse certification is also a theme of several projects [304]. Patel and company's work focused on (checkable) documentation standards, identification of system-dependent features, path analysis to determine if functions return valid results on all logical paths, and measurable system quality indicators (cohesion, coupling, and complexity). In general, these features focus on not only reuse, but quality.

One of the most successful early research programs in component-based reuse was the STARS (Software Technology for Adaptable, Reliable Software) program. The STARS Asset Library Open Architecture Framework (ALOAF) was the reusable assets library platform and architecture that enabled the exchange of reusable assets among diverse libraries [343].

1.3.6 Formal Methods

Several formal methods are related to this work. Algebraic approaches are the most closely connected due to their constructive nature, recent application to object-oriented systems, and tool support.

Abstract data types can be viewed as more than the algebraic theory of ADTs a la Gunter and Goguen [176, 177]. The use of ADTs has been incorporated in an introductory computer science course series [113], taxonomies a la EiffelBase [268], facet-based classification [272, 344], and more into the ADT nomenclature.

Students of computer science are immersed in the world of ADTs by their second or third course. The classic ADTs: stacks, lists, queues, etc., are not only structured tools for programmers' use; they are the core of the ontology of computation construction. Understanding the use and construction of these key structural ideals is essential to writing software and learning new languages, libraries, and models of computation. Therefore, a kind representation of ADTs is essential to the theory's use in software engineering.

Capturing and manipulating taxonomies and ontologies of data types in kind theory is trivial through the use of the inheritance and containment operators. Additionally, providing localized re-interpretations, (e.g., dequeue for two-sided linked-list), is trivial with canonicalizations and personalized interpretations.

Typically these taxonomies are either defined via refinement by a library or a textbook author: e.g., Doug Lea's original `libc++` and Bertrand Meyer's Reusable Software [268]; or *de facto* via an organization or corporations influence: e.g., the POSIX standards. Regardless, kind theory's use of explicit agents and contexts captures both the source of the characterization as well as its dependencies and non-programmatic contexts.

For example, explicit bindings to documentation, text sections, usage, etc. create a sort

of a mega-literate programming for ADTs. A student's use of a data structure in a program is directly tied back to the text from which he learned the construct. Consequently, when the instructor grades the program and finds an error in usage, she can directly back-reference to the same source text to show the student, contextually within both his learning and programming environments, where he went wrong.

Taxonomic systems are often not explicitly structured at all beyond *ad hoc* documentation and file system conventions. This problem motivated the introduction of facet-based classification schemas which, while useful for small-to-medium sized projects, have been found to be less useful in large, dynamic environments [272].

A kind representation of facet schemas is trivial as well. The facet domain is defined as a fixed set of terms (instances of the kind FACET), and sets of facets are bound to assets with a facet binding operator like ISDESCRIBEDWITHFACETSBY. Not only does this kind representation capture the facets described in the reuse literature, but it is significantly more expressive and flexible because it permits user modification of the facet domain itself, emergent community definitions of facets with bottom-up ontology construction, and automatic deduction of facet correlations via bridges to other domains, as discussed below.

Kind theory's capability to unify all of the above aspects of ADTs shows the expressiveness and flexibility of the theory and realization. The same framework is used to represent: (a) the structure of taxonomies and ontologies, (b) educational and referential presentations via textbooks and manuals, and (c) facet-based classification and search. The theory ties all these aspects together into a single, searchable, customizable, and structured whole. Most importantly, the inherent capability of modeling algebraic ADT theory in the representation, given its algebraic foundations, moves this realization of the domain of ADTs past anything found in the reuse, knowledge representation, and programming languages literature.

1.3.7 Components and Composition

1.3.7.1 Concepts to Drive New Kinds

Advanced forms of object composition like delegation, split objects, and dynamic inheritance open up new possibilities for combining instances into new aggregations [22, 25, 359]. Investigating these new paradigms will result in new forms of kind composition. Likewise, new forms of abstraction like subject-oriented and aspect-oriented programming, provide a source of new abstraction realizations for kind theory [181, 203].

1.3.7.2 Semantic Composition

Straightforward structural subtyping can be used as a theoretical framework for semantic composition, especially in strongly typed languages; e.g., Muckelbauer's work in distributed object-based systems [283]. The problem with this approach is that only renaming and reordering operations are possible, and little theoretical infrastructure supports reasoning about the full semantics of components.

Stronger theoretical formalisms exists for specifying and reasoning about such compositional systems; for example, the recent work of Molina-Bravo and Pimentel [276]. But such work is far from being practically applicable in the near future as there is no strong connection with real programming languages, software engineering methods, and no tool support.

Other expressive formalisms for compositionality exist, primarily in the category theoretic domain [118]. This work is starting to see real application as Fiadeiro has moved into a more industrial role in promoting this technology.

Work at IBM by Yellin and Strom, inspired by the problems with the Aesop system [139], has covered this territory before in the context of object protocols and weak, non-formal semi-automatic adapter construction [386, 385]. The work has a very ad hoc feel, even after the development of a formal model for such architectures [11]. This model will be used as a motivating example for the evolution of the design language Extended BON [208].

In a Java context, Wang et al proposed an service-event model with an ontology of ports and links, extending the JavaBeans descriptor model [365].

Other related work comes from the areas of: domain-specific languages, especially for component specification and reuse [306]; the automatic programming community, e.g, early work by Barstow [29, 30] and Cleveland [74]; conceptual reuse such as Castano and De Antonellis [51]; and last but certainly not least, the software transformation systems community, including the voluminous works of Biggerstaff and Batory.

Much of this work is reviewed nicely in [105, 303, 320].

The primary difference between all of these formalisms and systems and this one is that this system has a broad, firm theoretic foundation. That foundation, kind theory, was specifically designed to reason about reusable assets in an open collaborative context. Thus, this work is not tied to a specific language or realization, integrates the domains of reuse, knowledge representation, automatic programming, and program transformation, and does so in a theoretical and systematic framework that supports global collaboration among many participants.

Documentation reuse is most often discussed in the literate programming [65] and hypertext domains [122]. Little research exists for formalizing the semantics of semi-structured documentation. Some work in formal concept analysis and related formalisms [337, 377] has

started down this path, but with extremely loose semantics and little to no tool support.

Recent work by Wendorff [371, 372] bears resemblance to this work both in its nature (concept formation and resolution) and theoretic infrastructure (category theory). The work here is differentiated by its broader scope, its more expressive formalism, and its realization in tools. Additionally, the user-centric nature of kind theory (not discussed in this article) makes for exposing the formalism to the typical software engineer a straightforward practice.

1.3.8 Knowledgeable Software Environments

Several prototype environments have been developed over the years that reflect various aspects of the planned KDE. Several of these advanced IDEs focus on iconographic/pictorial interfaces and have all the aforementioned problems associated with them with regards to (a lack of) semantics [171].

Promising work from Cooke integrates a formal foundation, specification, semantics, and non-standard logics into CASE environments [82, 83, 85, 84, 87, 251, 314, 315]. Combined with logics for program synthesis [88], the resulting architectures are quite expressive and powerful. Planned developments in kind-centric KDEs include many of the best features of these environments.

1.4 Contributions

This thesis consists of a general, theory- and system-independent formalism and complementary set of tools to help solve the "a la carte" problem. This work is meant to help software engineers understand and practice reuse broadly. It is also meant to help practitioners bridge results, theoretical or practical, across related but disconnected disciplines.

1.4.1 Theoretical

The first contribution discussed in this dissertation is a user-motivated synthesis of formal methods from the domains of reuse-centric software engineering and knowledge engineering. This resulting synthesis is called kind theory.

Kind theory must be used with an underlying, more general formal system. It is a thin formal layer that is used to describe, reason about, and search for knowledge within the underlying formal system.

Kind theory's definition is the result of analyzing (a) several branches of mathematics, (b) several formal methods for software and knowledge engineering, and (c) the tools and technologies used in the software creation process, particularly those that help people collaborate.

As summarized in Section 1.2.1, this analysis drove the definition of the core constructs and axioms of kind theory.

If kind theory is to be used to reason about reusable structured information in a variety of domains, though primarily within software, it must have numerous models. Because kind theory's structure is so generic and relates to so many models, it needs (a) its own terminology and (b) its own syntax.

If some existing terminology were used, then the work might be characterized as being applicable only to a specific domain. In regards to notation, it is desirable that the syntax of a language is as close as possible to its model. This makes for a more natural style, use, and presentation.

Kind theory also provides a foundation for (a) reasoning about information under uncertainty, and (b) reasoning in the presence of multiple agents. Agents are entities (systems or people) that state, manipulate, and reason about knowledge.

Finally, there are some other tertiary new ideas presented in the theoretical discussions in Chapters 2 through 5. Examples include: an innovative syntax that combines aspects of syntax (well-formedness) and semantics; the use of canonicality for renaming; the classification of data and behavior by an interpretive context; and the explicit introduction of agents that are possibly human.

1.4.2 Analysis of Existing Software Engineering Constructs

A second contribution is a study of a set of core software engineering constructs from a general, domain-independent point of view. Several of these constructs (e.g., loops, assertions, predicates, documentation, etc.) have been characterized using kind theory in Chapter 6. Some discussion is also provided of how such a representation might impact the software development process.

1.4.3 Definition of New Constructs

Another contribution in this area is a set of new software engineering constructs called semantic properties. Semantic properties help software engineers document and reason about their systems. When used with some of the computational tools described below, semantic properties also can help characterize software components, more accurately reason about such components (e.g., their utility, correctness, reliability, etc.), and guide the generation of glue code to compose components.

1.4.4 Tools

A number of tools have been constructed to show off some of kind theory's capabilities. These tools are meant to empower individuals working with complex software systems.

The first tool is the specification of kind theory in an algebraic theorem proving environment. Such an environment permits the manipulation of logical contexts, the application of inference rules, and a direct use the formalism.

A second tool constructed is a general-purpose reusable asset repository called the Jiki. The Jiki has been designed to hold arbitrary reusable assets, not just source code. It provides both a web-based and programmatic interface through which assets can be stored, manipulated, searched for, and reasoned about. Actions taken on assets within the Jiki, exposed as web-based forms or API calls, currently are implemented directly in Java, but will eventually use kind theory semantics, once integrated with the theorem proving environment.

A third tool now underway is a design model checker. This tool is meant to help a software engineer analyze various aspects of a complex system's design, specification, and implementation.

1.4.5 Summary

To summarize, this work contains several contributions:

1. a new theory used to reason about reusable structured information in the presence of uncertainty and within a collaborative framework,

2. an analysis of some existing software concepts within this theory and a discussion of the applications thereof,

3. the definition of new software constructs using the theory that help reason about reusable component-based software systems, and

4. a set of tools that realize and utilize the above theory and concepts to help software professionals construct quality software with reuse.

1.5 Overview

Kind theory is a theory of software knowledge. It is a relatively small formalization which, when used in conjunction with a more general formal foundation, provides an initial framework for structuring and reasoning about software-as-knowledge within the formal foundation. Based upon early experimentation and application, the core constructs chosen seem to be a good

starting set. The theory and its realizations will be re-evaluated and refactored over time as limitations and problems come to light.

1.5.1 A Classification of Kind Theory

Kind theory is a self-reflective, autoepistemic, paraconsistent logic.

The word autoepistemic captures the notion that kind theory lets one describe and reason about one's own (thus "auto") knowledge and beliefs (thus "epistemic"). There are a large number of autoepistemic logics available today, most of which are non-monotonic. Their purpose is primarily to help draw conclusions with incomplete, introspective information. They also focus on the fact that not only knowledge, but the absence of knowledge, enables certain judgments [358].

Kind theory is autoepistemic for the same reasons. It is a non-classical logic with three truth values, one of which represents the absence of knowledge. Additionally, because its semantics are (intentionally, logically) weak, its models are existing autoepistemic logics.

Paraconsistent logics are those that can deal with logical inconsistencies of various types [361]. These logics' primary purpose is formally modeling the messy real world, a world that contains contradictions of various sorts, but in which humans can operate and reason without issue [33, 311].

Most paraconsistent logics are non-classical. Their development originates from artificial intelligence research in the fields of knowledge representation and engineering. As they are a relatively recent addition to the logician's toolbox, they are still outside the mainstream of logic. Thus, only a relatively small number of examples have been developed over the past twenty years.

Kind theory is a paraconsistent logic because, in formalizing and reasoning about a collaborating group of agents' beliefs, inconsistency is frequent. Likewise, temporary, intentional inconsistency is a hallmark of software engineering—systems under development are necessarily inconsistent and incomplete because they are not fully elucidated from a requirements or an engineering point of view [106]. Thus, kind theory has the aspects of existing paraconsistent logics, and many of such logics are models for those aspects of kind theory.

1.5.2 Theoretical Models

Information that describes kinds or instances must be structured so that computation can act upon it. Since a structural representation of data is necessary for computational realization, it seems natural to define a type system for kinds.

The specific type system defined for kind theory is called T_Γ. T_Γ structures the information

that is important to kind theory, in particular, the specification of kinds and instances. It also provides a number of decidable, computable, efficient operations on that structure—operations that are chosen specifically because they capture aspects of kind theory's basic functions and relations.

T_Γ is not a model for the whole of kind theory—it only models the structural aspects of the theory. What is missing are the functional pieces, the aspects of kind theory that express and execute functions.

In a type theoretical framework, the model for such computational structures is a lambda calculus of some sort [327]. But most type theory-based computational frameworks [307] (e.g., PVS [330], Coq [102], PRL [80], MetaPRL [186]) do not provide a direct realization of a lambda calculus that is an efficient computational abstraction. Thus, a model for these missing pieces is necessary in something other than a lambda calculus

To computationally realize T_Γ, a many-sorted algebra called $A_\mathcal{T}$ that models T_Γ is defined. In this algebra, one can create, manipulate, and reason about types as models for kinds. The functional part of kind theory is realized as sets of equations in an equational or rewriting logic.

1.5.3 Practical Models

Because this work can be characterized as a framework for collaboration as applying to software, it needs to be at least as capable as some of the systems that have been built and used in the past. Such systems come in three varieties.

The first are general purpose collaboration mediums, particularly in a distributed setting. Examples include general purpose discussion forums like Slashdot, knowledge archives like Everything2[7] and the AAAS's Signal Transduction Knowledge Environment (STKE[8]), and collaborative software development sites like SourceForge. This work need be at least complete enough to model the capabilities of such collaborative systems. The intent is to both reason about their use as well as incorporate some of their lessons and technologies.

The second type of systems this work must cover are those developed wholly within the software reuse community. For the most part, such technologies are used to classify and discover program code as reusable assets. Popular techniques include keyword searches, facet-based classification, and signature matching.

The systems used in the everyday experience of building software are the third, and most important type of system to consider. Developers are not likely to change the way that they work, so incorporating this technology into their development model, perhaps even in a sur-

[7] http://www.everything2.com/
[8] http://www.stke.org/

reptitious manner, is one of the only ways to enhance their capabilities and introduce a new formalism.

1.5.4 Systems

A system that realizes kind theory is called a kind system. The types of questions that a developer puts to such a system are many and varied. Example questions include "Does this new thing fit the definition of any kinds I understand?" and "How closely related are the following two kinds from a structural point of view?".

History and experience shows that the majority of developers have no interest in learning such a formalism, regardless of how it might impact their work. Thus, the system must be exposed in both an explicit and an implicit fashion.

Explicitly, a user can directly interact with the formal system, manually introducing new kinds and instances, making claims and beliefs about structure, and evaluating judgments. Only an expert, or a person interested in extending the fundamentals of kind theory, would use such a direct, explicit interface.

The majority of users are meant to implicitly interact with the system. This interaction is facilitated by integrating the kind system with the major tools used by software engineers in their day-to-day activities. The editor, the compiler, the configuration and version management tools, and the integrated development environment are all legitimate targets for integration.

Due to kind theory's general autoepistemic nature, more than program code can be formally characterized. A developer's tools, context, process, and more can be part of his explicit kind theoretic context, implicitly captured by these knowledgeable environments. Also, because kind theory exposes hooks to the non-expert user in the form of truth structures with intuitive semantics, the hope is that the feedback mechanism inherent in the theory is usable, almost accidentally, by the common software engineer.

Tools that have been augmented with a kind theoretic infrastructure are termed knowledgeable. Performing software engineering with such infrastructure is knowledgeable software engineering, and similarly, such software development environments are called knowledgeable development environments.

1.6 Thesis Structure

The outline of the thesis is as follows.

Chapter 2 presents the core of this thesis, a logic called kind theory. Its syntax, semantics, meta-theoretical properties, and use are discussed.

Chapter 3 presents some basic theorems of kind theory.

A type system that is the underlying assembly language for basic structural reasoning in kind theory is discussed in Chapter 4. An algebra that models this type system and provides a first implementation of kind theory is also discussed.

A reflective description of kind theory is presented in Chapter 5. The theory is used to describe itself, thereby providing a foundation for intertheory and intermodel bridges between kind theory and other formal systems.

Chapter 6 reviews a variety of different ways that kind theory can be used in software engineering and mathematics. Several canonical structures from the programming, reuse, and software engineering communities are focused upon. Problems in asset repositories, including storage, search, and architecture are discussed. Finally, ideas about new development environments that use kind theory are reviewed.

The application of kind theory to problems in component specification and composition is covered in Chapter 7.

Finally, Chapter 8 rounds out the dissertation. It lists some open problems and the many directions for future research available.

Chapter 2

Kind Theory

The structure of a kind is captured by (i) a set of core function and relations defined on all kinds, (ii) a set of universal properties defined on every kind, and (iii) whatever extra functions and relations that are defined specific to K. The semantics of a kind, on the other hand, are determined by the truth values bound to these structures by agents via claims and beliefs.

This chapter describes the syntax and semantics of the formal system KT which proscribes kind theory.

2.1 Basic Terminology

Throughout the rest of this work, a number of terms are used that have overloaded meanings in the variety of research domains that are touched upon. To avoid confusion, explicit informal definitions are provided here. Each of these terms is also formally defined in kind theory.

The terms **use** and **reuse** are often misused because no widely accepted definition of the terms exists. In this work, something is **used** if it is referenced, utilized, referred to, or otherwise mentioned as part of describing, building, or manipulating a new thing. Something is **reused** if it is used in a context other than that which it was originally used.

A **construct** is a piece of knowledge or information that is physical or nonphysical, and that is referenced, identified, and characterized in some fashion. A construct is not necessarily interesting or worthwhile, so a term is needed to distinguish that which is reusable. The term **asset** is used to make this differentiation.

An **asset** is a construct that has some reuse value; it is a construct that can be, is, or has been reused. A **kind** is an abstraction used to describe an asset.

Some term is needed to describe a specific thing that fits or matches an abstraction, a intention for the extension that are kinds. An **instance** is a construct that **realizes** a particular kind. For example, the paperback book on my bookshelf entitled “A Portrait of the Artist as a Young Man” is an instance of the kind BOOK.

An **agent** is a person or system that interacts with a kind. Four examples of agents are the author of a paper, or the tool that he uses to write the paper, read his email, or mechanically check a proof.

This definition might seem quite anthropomorphic, since artificial agents cannot discern any meaning in anything; they can only manipulate data according to ways that have meaning for the person who programmed the agent[1]. To clarify, artificial agents do not have some magical oracular ability to discern truth or define semantics. Agents are necessary only to support the representation of external, often computational, entities within the theory. In this manner, the decisions of external agents, be they derived by a flip of a coin, a call to a function in a library, or the execution of a judgment within an automated proof system, are all equally representable.

Some kind are only chunks of data, but others are observed by an agent which defines their semantics. A kind is **interpretable** if some agent, human or machine, can, with the proper context, interpret, execute, or otherwise discern meaning from it. A kind is a **behavior** if it is interpretable by a particular agent in a specific context, otherwise it is called **data**.

Some people or programs can discern meaning from anything: tea leaves, the palm of a hand, the number of characters in a documents[2]. Thus, potentially, every kind is interpretable. What differentiates the actual from the potential is the current context. Likewise, only those interpretable kinds that are actually interpretable within a context are called behaviors.

If kinds are not only data, but are also algorithmic or interpretable, then some classification schema is necessary for such functional assets. The term **function** is used for such as it is normally defined.

A function f from a set A to a set B maps each element of A to exactly a single element of B. Such a function is written f :A →B or $A \xrightarrow{f} B$. Functions can map from kinds to kinds, instances to instances, or any combination thereof. Behaviors, as interpretable entities, are thus functions.

Each function is also represented by a kind. Kind functions, or what are called **functional kind**, are defined with respect to the agent which interprets them. In other words, functions are classified by kinds like any other entity; e.g., functions of a certain form belong to a particular kind that describes that form.

A realization (instance) of a functional kind is called a **functional realization**. A specific mathematical function (e.g., $f(x) = x^2$) or executable entity (e.g., the program “wc”) realizes a functional kind by realizing all structure of that kind. Such structure includes the signature of the function as well as all of its semantics.

Additionally, only some functions are computable in the Church/Turing sense. Thus, a

[1] Thanks to José Meseguer for pointing this out.
[2] Thanks to Mani Chandy for bringing this point to light.

functional kind called **computation** is introduced to denote exactly those that are computable.

A **relation** is also defined as usual. A (binary) relation between sets A and B is defined as a function r that maps each pair (a, b), where $a \in A$ and $b \in B$, to exactly one element of the set of Boolean truth values *true* and *false*, denoted $\top$ and $\bot$ respectively. A third truth value called *unknown* is also used in this theory, and it is denoted ?. The truth values (codomains) of all relations that follow have at least three truth values: $\{\top, \bot, ?\}$.

A binary relation is typically written as ArB rather than r :A $\times$B $\rightarrow\{\top, \bot, ?\}$. Relations are not only binary. N-ary relations are defined in the standard manner as the functional composition of binary relations. Since a relation is a function, all relations are also classified by kinds as well.

To interpret something, an agent needs a frame of reference, otherwise the interpretation is ill-founded (it relies upon undefined concepts). The term used for this frame of reference is **context**.

Finally, knowledge itself is often organized into bins. It is a natural byproduct of how the human mind uses information. A **domain** is a context covering a specific realm of knowledge. Example domains are "many order sorted algebra with membership equational logic" and "crayon colors."

2.2 Related Generic Constructs

Many similar constructs have been defined in the past. OBJ's theories are algebraic abstractions of structure and semantics [148]. Sorts and types are structural representations of well-ordered constructs [159, 349]. Clichés, patterns, modules, templates, archetypes, and intentions are also similar programmatic constructs that focus on generic program abstraction via fragments [9, 134, 260, 289, 367].

Kinds differ from these abstractions in several ways. In short, first, the formal foundation of kind theory is basic logic, thus it is applicable to universal algebra and type theory (but not, perhaps, viceversa). Second, the explicit incorporation of agents and agent-imposed semantics makes kind theory unique among generic formalisms for reuse. Third, the basic operations of kind theory are motivated by a careful analysis of structured systems, not just software artifacts of the day. Fourth, multiple models for kind theory can be realized for tuned application to different domains.

In essence, because *KT* is a theory-independent formalism for reuse, it is unsurprising that many of these abstractions might be legitimate models for kinds within different theoretical and practical domains. Constructing such a realization follows a regular pattern. First, kinds are mapped to the abstraction construct (e.g., sorts for algebras, types for type theory, etc.),

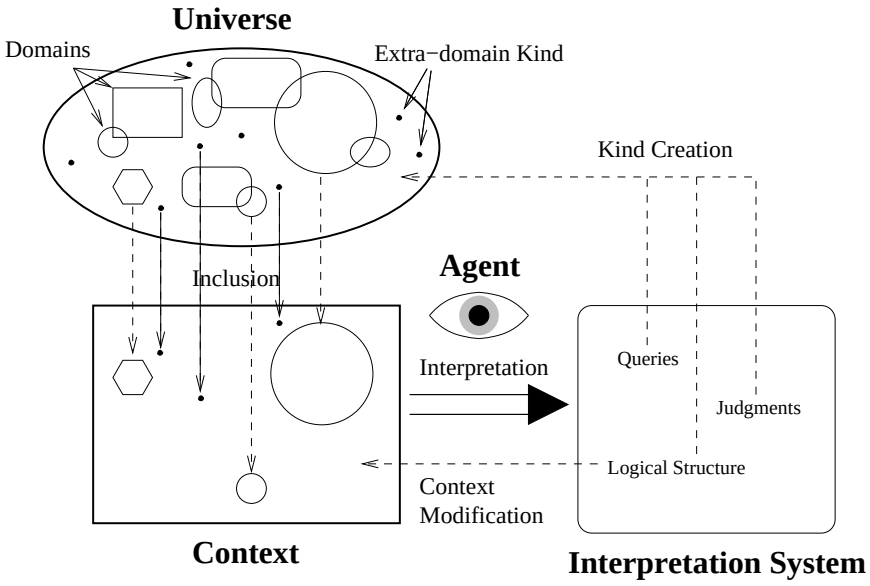
instances are mapped to the corresponding realization construct (e.g., sorted terms in algebras, typed sequents in sequent calculi and related type theories).

Next, the basic operators of *KT* are mapped to the appropriate constructs within the target system. For example, within many order sorted algebras, the inheritance maps to the subsort relation, inclusion maps to structural containment (e.g., equations are part of theories, variables and operators are part of equations, etc.), equivalency maps to term equivalency (e.g. "=" after canonical rewrites), etc.

2.3 Structure

The cornerstone of kind theory is structure. Structure is characterized by the properties of an object, the relationships in which it is involved, and the manner in which it is interpreted by an observer. Consider Figure 2.1.

Figure 2.1: Pictorial Overview of Kind Theory



The *Universe* is the unbounded set of all kinds and instances. Portions of that universe are domains which collect specialized areas of knowledge in the universe. Individual kind can belong to many such domains or none at all. The latter of which are called extra-domain kind. A context is created by an agent. The agent chooses specific domains and extra-domain kinds and collects them in a context.

This context is interpreted in a number of different ways, depending upon exactly which domains and kinds are part of the context. The interpretation takes place in an interpretation

system, specified again, by the current context. The interpretation system need not be a formal system; a human expert reading a context, performing the interpretation in his or her mind, and making some inference based upon his or her experience is a legitimate interpretation system. Other legitimate interpretations are queries (for example, on a database) and judgments (on a logical system, for instance). The results of these interpretations can be named, reinserted into the universe, then introduced into the current context, starting the cycle anew.

The most important point about these interpretations is that they are structure-preserving. And, since semantics is a part of structure, interpretations are also semantics-preserving. Thus, interpretations can be viewed as semantics-preserving functions.

2.3.1 Examples of Structure

Consider the empty context $\varnothing = \{\}$. This context is the empty set. What structure does this context have? The structure inherent in this context is the trivial structure: it does not have any elements.

Now add a couple of kinds to this empty context and call this new context $C = \{k, k'\}$. If the kinds k and k' do not relate to each other in any way, then the only structure inherent in C is that it contains two kind that do not relate to one another.

Adding two relations "$<$" and "$\supset$" to the context C creates a new context $C' = \{k, k', <, \supset\}$. By adding the sentences "$k < k'$" and "$k \supset k'$" to C' the context C'' is derived. The structure that exists within C'' is that it contains two kinds, and they are related to each other exactly in two ways, one of which is called "$<$" and the other "$\supset$", as specified by the two sentences $k < k'$ and $k \supset k'$.

Note that, because the context contains no other terms, no semantics are attached whatsoever to the symbols k, k', "$<$", and "$\supset$". Thus you, as the reader, cannot and should not attempt to attach some meaning to these symbols.

2.3.2 An Example Interpretation

Consider the notion of composition within the context of writing a textual document, like this chapter. The fundamental units of data, the most basic ontological constructs of this domain, are elements like words, sentences, paragraphs, sections, etc. Thus, these are the basic kinds of this domain.

What does it mean to compose two sentences? One semantics for such an operator is textual concatenation. Such a semantics can be stated in variety of ways, though the use of a programming language, a formal model, or some other interpretable behavior.

A formalism need not be used to provide a basic semantics. The meaning of this compose

operator can be denoted, for example, with a specification written in the Bourne-again shell script language (bash). In particular:

Example 1 *Let A and B be instances of the kind* SENTENCE *in the context FoundationsExample. Define the new kind* SENTENCECOMPOSE *whose domain and codomain are both* SENTENCE*. A functional realization (instance) of that kind might be the following sequence of symbols which happen to be bash code:*

```
function SentenceCompose() = { $1$2 }
```

An example use of this functional realization is the sentence "`SentenceCompose $A $B`*" where A and B are realizations of the kind* SENTENCE*.*

Until this sequence of symbols is interpreted, it has no semantics: it is data. Once this sequence of symbols is placed in the right context (e.g., Bash version 2.0 running on any architecture), it provides a model for SENTENCECOMPOSE, and thus a semantics.

Such a semantics can also be specified solely in natural language. It is widely recognized that additional (semi-)formal information like this Bash specification decreases the possibility of misinterpretation.

2.4 Basic Formal Definitions

2.4.1 Definitions of Core Notions

$\mathbb{U}$ is the universe of all potential kind and instances. $\mathbb{U}$ is an unbounded set. A context is a subset of the universe: $\mathbb{C} \subset \mathbb{U}$. Evidence $\mathbb{E}$ is some testament (e.g., documents, a proof, etc.) that is used to validate a truth structure and is a finite subset of a context: $\mathbb{E} \subset \mathbb{C}$. The elements of the universe are partitioned into two classes: instances ($\mathbb{I}$) and kinds ($\mathbb{K}$).

With respect to any given interpretation i, the universe is partitioned into two unbounded sets: $\mathbb{D}_i$ and $\mathbb{B}_i$, called data and behavior respectively. With respect to a specific interpretation, data and behavior are disjoint. Kinds that are elements of $\mathbb{B}$ are called behavioral kinds or functional kinds.

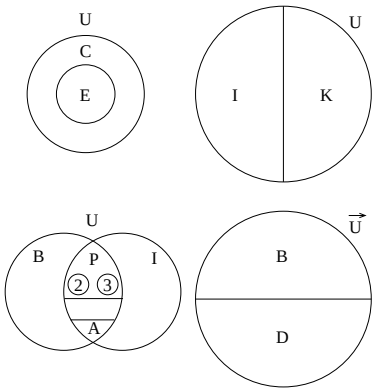
This bipartition of data and behavior is also seen in varying degrees in reflective languages like Lisp, Java, and Smalltalk and higher-order mathematics.

The basic Boolean values of **true** and **false** are denoted by $\top$ and $\bot$ respectively. The set of Boolean values is denoted by the symbol $\mathbb{2} = \{\top, \bot\}$. A third truth value of **unknown** is denoted with the symbol ?. This third truth value is different from true and false. The set of ternary truth values is specified by the symbol $\mathbb{3} = \{\top, \bot, ?\}$.

Predicates, denoted with the symbol $\mathbb{P}$, are behavioral kinds with codomains $\not\Vdash$.

An **agent** is an observing entity within the system. Agents are a subset of the intersection of behaviors and instances, but are not predicates.

Figure 2.2: Core Notion Sets Overview



These basic notions are pictorially summarized in Figure 2.2. Note that the label $\mathbb{U}$ in the figure denotes everything outside of the outermost circles.

2.4.2 Overview of Core Behaviors

As discussed in Section 1.2, and motived by the desired models for this logic, a number of primary behaviors (relations and functions) are predefined in kind theory.

2.4.2.1 Inheritance

Inheritance is a relation that holds between a child and one or more parents, thus inheritance defines a lattice. The generic notion of inheritance is that some qualities are transferred by some means from parent to child. Inheritance is the relation that is used to classify kind. When discussing inheritance, words from genealogy are used: child, parent, sibling, etc. Additionally, to describe the relation of inheritance itself, the connotative term "*is-a*" is used.

Classification is one of the most common structuring notions known in science. The Linnaean method has been used in various incarnations for hundreds of years in biology to classify organisms. For example, the plant species commonly known as wild ginger *is-a asarum* and *Asarum is-a Aristolochiaceae*. Another example of inheritance comes from library science where researchers have developed several classification schemas for books. For example, the novel "Portrait of the Artist as a Young Dog" by Dylan Thomas is classified as "English Literature" (denoted by the prefix "PR") by the Library of Congress classification system.

A final example of classification comes from the domain of reusable software, is seen in the EiffelBase class library. Eiffel has a very flexible form of class inheritance, and the designers of the EiffelBase class library took advantage of it to the utmost. For example, the class LIST[G] directly inherits only from CHAIN[G], thus LIST[G] is-a CHAIN[G]. But since CHAIN[G] inherits from further classes (e.g., CURSOR_STRUCTURE[G], INDEXABLE[G, INTEGER], and SEQUENCE[G]), then LIST[G] is-a SEQUENCE[G] (an others) as well. In all, LIST[G] inherits from twenty other classes.

2.4.2.2 Inclusion

Inclusion is a relation that holds between two constructs, the whole and the part. A part is a portion of the whole; and a whole contains a part. The term *has-a* is used to connote inclusion.

Consider this document. Its physical structure can be hierarchically decomposed in a variety of different ways. For example, syntactically, letters make up words, words make up sentences, sentences make up paragraphs, etc. Thus, a given paragraph *has-a* particular sentence. Alternatively, a particular sentence is *part-of* a particular paragraph.

More generally, grammars (like those used in compiler construction) have a basic inclusion structure. Tokens are part-of rules, and rules are (recursively) part-of rules.

Likewise, within programming languages, inclusion structures are everywhere. Methods are part-of classes. Methods have (has-a) signatures, bodies, specifications, documentation, etc. Files have classes, authors, copyrights, licenses, etc.

2.4.2.3 Equivalence

Equivalence is a relation that holds between constructs that are the same, either syntactically or semantically, in some context. Two constructs can be equivalent for more than one reason. The phrase *is-equivalent-to* is used to connote equivalence.

A basic example of equivalence involves the earlier mention of wild ginger. The scientific name for this species is *Asarum canadense*. Thus, *wild ginger is-equivalent-to Asarum canadense*.

Many more complex examples of equivalence can be found in software systems. The features “out” and “tagged_out” of the class ANY in Eiffel are equivalent by definition, as they are what are called synonyms [269]. The methods java.io.InputStream.read() and java.io.InputStream.-read(byte) are equivalent if the second uses a byte array of length one and the return value of the first is mapped to the single element of that array. Additionally, java.io.InputStream.read() is equivalent to the method java.io.FileInputStream.read() simply because the latter inherits from the former and does not override its definition.

Equivalence is a context-sensitive relation. Consider the two strings "$Y = 2 * X$" and "$Z = X * 2$". Textually, they are not equivalent because they contain different characters. If interpreted in an algebraic setting, they could be equivalent equations, depending upon the specific algebras in question. If they both are interpreted in the same algebra, and if that algebra is commutative and alpha-renaming is a part of the definition of equivalence, then they are equivalent.

Equivalence is not only applicable to formal mathematical statements. Consider a pinball machine and a Sony PlayStation. How are these two things equivalent? Obviously, both are games that people play for enjoyment, so this classification criterion is one equivalence class. Another equivalence is that both devices have buttons, thus some property-based criteria imply equivalence. Another more subtle equivalence class is the fact that both devices encode the a finite subgroup of $(\mathbb{Z}, +)$ because both keep score in some fashion.

2.4.2.4 Composition

Composition relates to the general notion of taking two or more constructs and putting them together or taking them apart. Composition in kind theory is a constructive, functional operation—its result is a new thing that has some of the properties of its constituent pieces. The semantics of the specific composition operation used dictates the properties of the new construct and a generic semantics are defined to which all composition operations must adhere.

Any constructive or deconstructive operation is composition. A father putting together the parts of a bicycle on Christmas Eve is performing a type of composition. The `cat` command in UNIX systems is another example of composition, this time of two byte streams.

The semantics of multiple inheritance within programming languages is one of the most prevalent and complex forms of composition in software systems. For example, Java has interface-based multiple inheritance. The class java.lang.String implements java.lang.Serializable, java.lang.Comparable, and java.lang.CharSequence. Within kind theory, the relationship between String and these three interfaces is expressed by (a) defining the semantics of Java interface composition, (b) using that composition operator to compose Serializable, Comparable, and CharSequence, and (c) defining the is-a relationship between String and the result of the composition in step (b).

Java's interface composition operator has a fairly simple semantics: simple collect together the methods of all the interfaces and, so long as they are unique, the resulting set is the composed construct (see Section 9.3.2 of [173]).

Eiffel, on the other hand, has a more complex class-based multiple inheritance semantics due to the flexibility inherit in its interface manipulation mechanisms. A kind theoretic composition operator for Eiffel class inheritance, call it eif-compose, must capture all of the interface manipulations available in the inherit clause. This includes support for feature redefinition,

undefinition, renaming, etc. Only those aspects important to the kind theory representation of Eiffel inheritance need be represented in e-compose. Consequently, the corresponding kind theoretic inheritance operator for Eiffel's notion of class inheritance, call it eif-inherit, must capture only the complementary semantics.

For example, an Eiffel kind theory domain would likely capture the rename Eiffel's class operator using canonicality. Thus, it would be completely unrepresented in composition, and likewise in inheritance. In other words, domain mappings need not be one-to-one.

Assets are created through the process of composition. Since new kinds are defined via different kinds of composition, there needs to be an ontological foundation on which to build—some set of basic kinds from which all other kinds derive. This basic set of kinds are called **ground kinds**. The term "ground" is used because of its ontological and reuse connotations. The set of ground kinds need not be non-empty, but since context creation is constructive, nothing could be created from an empty set of ground kinds.

Ground kinds are defined on a per-context basis. That is to say, for any given context, some subset of its kinds are ground kinds; all other kinds are derivative, having been defined via inheritance or composition. As to which kinds are ground, that is up to the creator of the context. Example sets of ground kinds are letters for the domain of written language, natural numbers for the domain of number theory, or all those classes directly inheriting from java.lang.Object in Java.

Ground kinds can be as basic or as complicated as necessary. The choice of ground kinds is driven by the assumption principles of the agent defining the context. This flexibility lets an agent find the right abstraction level for their domain, ignoring any underlying constructs that are not appropriate or important for a particular problem.

2.4.2.5 Realization

Realization is the relation between a specific instance and its kind. If I is an instance (realization) of a kind K, I *realizes* K, or I *is an instance of* K, or even I *is of kind* K.

Realization is the kind theory peer to typing, but is different because it is used both for classification as well as construction. When one states "let Z be a variable of type integer," a type is being related to the symbol Z. Likewise, the statement Z : INTEGER means both (a) an instantiation of the instance Z in the current context, and (b) **kinding** the instance Z as having the kind INTEGER.

2.4.2.6 Interpretation

Interpretation is the process of evaluating a construct in some context, translating it from one form to another. An interpretation is a behavior that realizes such a process. Associated with every interpretation are an agent and a context. The agent performs the interpretation on elements of the context.

Any evaluation process is a type of interpretation. Reading this document is one kind of interpretation, evaluating a expression with Mathematica is another. In each case, data is transformed via an agent (you, the reader, in the former case, and the Mathematica process in the latter) within a specific context.

Environments that evaluate executable specifications are examples of interpretation systems that are part of both the computational and mathematical domains. Theorem proving environments like Coq, Maude, and PVS are examples of such systems.

Software has numerous examples of interpretation. A Python runtime and a Java virtual machine are both examples of programs that exhibit interpretations (of Python and Java programs, as data, respectively). Likewise, the GNU compiler gcc is another example of an interpretation system, one that compiles C, C++, Objective-C, and Java source code (as data) into various other forms.

2.4.3 Relations as Ternary Functions

As mentioned earlier, relations are a specific class of functions.

Definition 1 (The Knowledge Trinity) *The three-valued set $\mathbb{K}$ is defined as $\mathbb{K} = \{\top, \bot, ?\}$. $\mathbb{K}$ is the standard set of truth values of many three-valued logics consisting of the classical Boolean truth values **true** ($\top$) and **false** ($\bot$) and the third value (?) representing **unknown** [178, 358].*

No formal semantics is specified for the third truth value **unknown**. Philosophically, this third value is viewed as a potential—at some point, when more evidence is available, the unknown truth value can be replaced by either a **true** or a **false**. This perspective is contrary to some three-valued logics, see the above references for more details. Thus, no specific three-valued (or many-valued) logic is imposed on kind theory; all consistent variants are legitimate.

Unknown truth values are used by paraconsistent logics and inference to both make standard judgments as well as drive tactics.

Definition 2 (Relation) *Relations are functions whose codomain is $\mathbb{K}$.*

Relatively few software-centric formal methods take this non-classical point of view, VDM being the primary example to the contrary [201].

2.4.4 Statements about Kinds

Semantics are specified in kind theory using truth structures. Truth structures come in two forms: claims and beliefs.

The fundamental distinction between claims and beliefs is a widely supported notion within the field of epistemology [20, 129, 174, 275, 387]. This distinction is also realized in many of the various mathematics used to reason under uncertainty including modal, non-monotonic, paraconsistent, conditional, and possibilistic logics, probabilistic networks, Dempster-Shafer Theory [127, 128, 129, 130, 131].

Kind theory is meant to specify a minimal semantics with many models. Therefore the formal definitions of these truth structures are just loose enough that they cover the common structural aspects of these choices in mathematical foundations. This opens up the possibility of many realizations of kind theory, with the choice of underlying reasoning system driven by the problem domain.

2.4.4.1 Claims

A **claim** is a statement made by an agent that is accompanied by a proof or other validation data of some form. Example validation data, or what is called evidence, are a formal mathematical proof, a well-reasoned argument (or what is called a demonstration by Martin-Löf [258]), or a set of consistent supportive data. A claim is a predicate on a behavioral kind. It does not make sense to make a claim about a construct that is not a function.

For example, the statement "a tree is a plant" is a claim. The behavior of this sentence is the predicate relation "is-a" (recall that behaviors encompass relations and functions) and the truth value associated with this claim is "true". The agent is whomever is making the claim. Thus, the expanded version of this claim is something of the form: "Joe Kiniry claims that the statement a tree is a plant is true because it is supported by the following set of botany textbooks".

A mathematically proven statement that is widely accepted is a claim. This phrasing (i.e., "widely accepted") is used because, for example, there are theorems that have a preliminary (not widely verified, or partial) proof, but are not yet recognized as being true by the mathematical community. For example, Andrew Wiles's proof of Fermat's Last Theorem is now broadly accepted by the mathematical community, but early versions of the proof were not because they had not yet been widely reviewed. Thus, early versions of this truth structure ("Andrew Wiles states that Fermat's Last Theorem is true because of the following proof") were not claims, but latter versions are.

Various statements made during program development, both implicit and explicit in the

development process, are also claims. For example, stating that a class C inherits from a parent class P is a claim. The inheritance relation is the behavior, the author of the class C is the agent involved, the context included the definition of inheritance, classes, etc., the truth value associated with the claim is "true", and the evidence is the source code itself. The statement within documentation or specification that a particular function f has complexity $O(g)$ because of some detailed provided analysis is another example of a claim.

2.4.4.2 Beliefs

A **belief** is a statement made by an agent that is accompanied by a verification structure and has some associated conviction metric. A belief is a statement with no formal evidence and is instead supported by statements of experience, intuition, etc.

Beliefs range in surety from "completely unsure" to "absolutely convinced" [20, 129]. Within kind theory, no specific metric is defined for the degree of conviction. The sole requirement placed on the associated belief logic is that the surety elements form a partial order.

The reason that a partial order is chosen over a total order or lattice is because (a) a partial order is more general than the alternatives as both a total order and a lattice are partial orders and (b) it is widely recognized that some belief sureties are incomparable [20].

2.5 Foundation Logic

Kind theory is an open system [388]. It depends upon external systems, formal ones in particular, to rigorously specify assets' qualities. Thus it may, to borrow a phrase from Cerioli and Meseguer, "borrow" the logic of other systems [53].

As discussed in the introduction, kind theory is merely a layer of formalism that helps structure some underlying formal domain. The underlying formalism is called the **foundational logic** of a particular realization of kind theory. When discussing a specific realization R of kind theory, the term FL_R is used to denote its foundational logic.

If a classical logic is chosen as a foundational logic, the third "unknown" truth value "?" must still be represented. In such systems, introduce a new symbol "?" that can be used anywhere a classical truth value can be used. Next, extend all truth tables of the logic to include this new placeholder truth value. For each slot of a table that involves the use of the new value "?", consider the two alternatives, were one to replace "?" with $\top$ or $\bot$. If both truth values are identical, use that value for the slot. If they differ, then put the placeholder "?" in the slot.

This use of "?" corresponds to one of the (logically) weakest forms of three-valued logic, but without the introduction of a new truth value or any other fundamental change to the classical logic [178]. Instead, it is said that the unknowns pollute judgments, rapidly making

any inferences impossible [178, 275]. Such a structure maintains the metalogical properties of the classical logic but makes a third unknown placeholder available for semantics.

2.6 Syntax of Kind Theory

Table 2.1 provides a more detailed summary of the syntax, taking care to highlight all related domains and codomains of the basic operations. Table 2.3 proscribes the precedence rules for all basic operators.

2.6.1 Alphabet

The alphabet consists of the following sets:

- The variables k, l, m, n (in upper and lower case form) denote members of the set $\mathbb{K}$, the classifying kind. The letters h, i, j and their primes denote members of the set $\mathbb{I}$, instances of classifying kind. Variables u, v, w denote members of the set $\mathbb{U}$. The symbol $\aleph$ denotes the top of the *is-a* lattice.
- The set of punctuation symbols consists of the symbols of the foundational logic *FL*, augmented by the following symbols: { '?', '$\top$', '$\bot$', '(', ')', '\', '|' }.
- The set of operators consists of the operators from *FL* combined with the new operators:

$$\lambda = \{<_p, <, \nabla, \subset_p, \supset, [\,], \equiv, =, \eqcirc, \lessdot, :, <_r, \circ, \cdot, \otimes, \times, \oplus, +, \rightsquigarrow, \hookrightarrow, \dashrightarrow, \gamma, \beta\}$$

2.6.2 Well-formed Formulas

Unless otherwise noted, well-formed formulas (*wffs* for short) are denoted by the upper-case letters corresponding to their identity, as mentioned in the discussion on the alphabet above.

A sentence S is a *wff* if it conforms to the well-formedness rules of *FL* or one of the following conditions:

1. S is the top element of inheritance $\aleph$.
2. S is a variable representing a kind or instance.
3. S is one of the basic relational forms $U <_p V$, $U < V$, $I <_r K$, $U \subset_p V$, $U \supset V$, $U \equiv V$, $U = V$, $U \eqcirc V$, $U \lessdot V$, or ∇K where U and V are either both kinds or both instances within a given expression and K is a kind and I is an instance.
4. S is a statement of kinding as in $I : K$ where I is an instance and K is a kind.

Table 2.1: A Detailed Summary of the Syntax of Kind Theory

$\Gamma \vdash \diamond$	Γ is a well-formed context.
$\Gamma \vdash U$	Γ is a well-formed context and U is some kind or instance (asset) in, or derivable from, that context.
$\Gamma \setminus S \vdash \diamond$	Γ is a well-formed context that does not include sentence S.
$\Gamma \vdash K \to L$	The *functional kind* $K \to L$ is in, or is derivable from, the context Γ.
$\Gamma \vdash I : K \to L$	The *functional realization* I of *functional kind* $K \to L$ is in, or is derivable from, the context Γ.
$\Gamma \vdash J$	J is a judgment derivable from the well-formed context Γ.
$\Gamma \vdash \nabla K$	K is a *ground kind*.
$\Gamma \vdash C <_p P$	The kind C's *parent is* the kind P.
$\Gamma \vdash C < P$	The kind C *inherits from* the kind P.
$\Gamma \vdash P \subset_p W$	The kind (instance) P is *part-of* the kind (instance) W.
$\Gamma \vdash W \supset P$	The kind (instance) W *has-a* kind (instance) P.
$\Gamma \vdash M \equiv N$	The kind (instance) M *is equivalent to* the kind (instance) N under equivalence class $\equiv$.
$\Gamma \vdash [M] = N$	The *canonical form* of kind (instance) M is the kind (instance) N.
$\Gamma \vdash K \eqslantless L$	The kind (instance) K is *fully equivalent* to the kind (instance) L.
$\Gamma \vdash K \eqslantless_{\equiv} L$	The kind (instance) K is *fully equivalent* to the kind (instance) L under equivalence class $\equiv$.
$\Gamma \vdash K \lessapprox L$	The kind (instance) K is *partially equivalent* to the kind (instance) L.
$\Gamma \vdash K \lessapprox_{\equiv} L$	The kind (instance) K is *partially equivalent* to the kind (instance) L under equivalence class $\equiv$.
$\Gamma \vdash F \cdot G = (F \circ G)$	The composition of kinds (instances) F and G is the compositional kind (instance) called $F \circ G$.
$\Gamma \vdash F + G = (F \oplus G)$	The *coproduct* of kinds (instances) F and G, is the compositional kind (instance) called $F \oplus G$.
$\Gamma \vdash F \times G = (F \otimes G)$	The *product* of kinds (instances) F and G, is the compositional kind (instance) called $F \otimes G$.
$\Gamma \vdash I : K$	The instance I *realizes* the kind K.
$\Gamma \vdash I <_r K$	The instance I *is a kind of* K.
$\Gamma \vdash K \underset{\Lambda}{\overset{A}{\rightsquigarrow}} L$	The agent A states that the *full interpretation kind* from kind (instance) L to kind (instance) K is derivable from the subcontext $\Lambda \subseteq \Gamma$.
$\Gamma \vdash K \underset{\Lambda}{\overset{A}{\hookrightarrow}} L$	The agent A states that the *partial interpretation kind* from kind (instance) L to kind (instance) K is derivable from the subcontext $\Lambda \subseteq \Gamma$.

Table 2.2: Claims and Beliefs

$\Gamma \vDash \gamma_A(\top P \mid E)$	The agent A claims that, within context Γ, and due to evidence E, the predicate P is *true*.
$\Gamma \vDash \gamma_A(\bot P \mid E)$	Likewise, but *false*.
$\Gamma \vDash \gamma_A(?P \mid E)$	Likewise, but *unknown*.
$\Gamma \vDash \beta_A(\top P \mid E) = D$	The agent A, within context Γ, believes, because of evidence E, that the predicate P is *true* with certainty D. D is a member of a *Poset* defined in the context Γ.
$\Gamma \vDash \beta_A(\bot P \mid E) = D$	Likewise, but *false*.
$\Gamma \vDash \beta_A(?P \mid E) = D$	Likewise, but *unknown*.

Table 2.3: Precedence

The precedence order is as follows, from strongest to weakest:

composition	(e.g., $\circ, \oplus, \otimes, \cdot, +, \times$)
functional kind	(e.g., $\rightarrow, \dashrightarrow, \rightsquigarrow, \hookrightarrow$)
inheritance	(e.g., $<_p, <$)
containment	(e.g., $\subset_p, \supset$)
equivalence	(e.g., $[\,], \equiv, =, \lessdot$)
realization	(e.g., $:, <_r$)

5. S is a statement of kind (instance) canonical form as in $[U] = V$ where U and V are kinds (instances).

6. S is a compositional kind (instance) of the form $U \otimes V$, $U \times V$, $U \oplus V$, $U + V$, $U \circ V$, or $U \cdot V$ where U and V are kinds (instances).

7. S is a functional kind (instance) of the form $U \rightarrow V$, $U \dashrightarrow V$, $U \underset{\Lambda}{\overset{A}{\rightsquigarrow}} V$, or $U \underset{\Lambda}{\overset{A}{\hookrightarrow}} V$ where U and V are kinds (instances), A is an agent, and Λ is a context.

8. S is a truth statement of the form $\top P$, $\bot P$, or $?P$ where P is a predicate kind or instance (i.e., an expression whose outermost operator's codomain is $\{\top, \bot, ?\}$).

9. S is one of the following claim or belief forms: "$\gamma_A(T \mid E)$" or "$\beta_A(T \mid E) = D$", where A is an agent, T is a truth statement of the previous form, E is any (possibly empty) *wff*, and D is a surety value.

2.6.3 Shorthand Notation

A shorthand notation is available that helps readability and clarity. The turnstiles used here are defined in Section 2.12.

The following syntactic rules of thumb hold:

- Unnecessary parentheses are dropped through the use of operator precedence.

- The syntactically asymmetric operators $<_p, <, \subset_p, \supset, \preccurlyeq, \rightarrow, \dashrightarrow, \rightsquigarrow, \hookrightarrow$ are syntactically reversible. For example, $U < V$ can be written $V > U$ if it is convenient. The symmetric operators $\equiv, =, \eqcirc$ can also be flipped. All composition operators cannot be reversed, as they have a definite left and right. In the rules that follow, the non-flipped varieties are always used.

- A normal turnstile $\vdash$ is used in the place of the more precise $\vdash_{KT}$ since only a single logic, that of KT is in use. The same holds for indications of truth ($\models$ for $\models_{KT}$).

- While contexts are literal sets in the realization of KT discussed in Chapters 4 and 5, this is not a necessary condition. Thus, the notation $\Gamma, U \vdash$ is used in place of the more cumbersome, and possibly mistaken $\Gamma \cup \{U\} \vdash$.

- A context is not made explicit if only a single context is being discussed. If the symbol S is used to represent a well-formed sentence, the actual sequent is $\Gamma \vdash S$ or $\Gamma \models S$, depending upon whether the discussed is about syntax or semantics.

- All variables on the right side of the turnstile need not be mentioned on the left side. For example, $\Gamma, K, L \vdash K < L$ is written as $\Gamma \vdash K < L$ where K and L are implicitly recognized as being members of Γ.

- Finally, the following shorthand rules for claim and belief kind are used:

$$\text{(Claim Turnstile*)}\quad \frac{\Gamma \models \gamma_A(P|E)}{\Gamma \models_A (P|E)} \qquad \text{(Claim Null-Evidence*)}\quad \frac{\Gamma \models_A (P|)}{\Gamma \models_A P}$$

$$\text{(Single-Agent Claim*)}\quad \frac{\Gamma \setminus A \text{ contains no agents} \quad \Gamma \models_A (P|E)}{\Gamma \models (P|E)}$$

$$\text{(Positive Claim*)}\quad \frac{\Gamma \models \gamma_A(\top P|E)}{\Gamma \models_A P} \qquad \text{(Negative Claim*)}\quad \frac{\Gamma \models \gamma_A(\bot P|E)}{\Gamma \models_A \neg P}$$

The rules of kind theory are presented as general logical rules without a particular foundational logic. The horizontal bar is meant as an extra-logical implication, not an operation at the object level. A rule named with an asterisk ("*") is reversible, meaning that it summarizes two rules, one read top-to-bottom, and one read bottom-to-top.

The rule (Claim Turnstile*) is used to rewrite claims. Claim expressions are written by dropping the γ and labeling the turnstile with the agent making the claim. This modification makes claim expressions similar to normal truth expressions in sequent-based systems.

(Claim Null-Evidence*) applies to claim expressions that do not have evidence terms. Since no evidence is used to validate P, the extraneous | and parentheses are dropped.

The rule (Single-Agent Claim*) is used when only a single agent exists in the current context. In typical autoepistemic fashion, if an agent is only describing, reasoning about, and manipulating its own context, there is no need to explicitly label every claim with an agent [275, 358].

The last two rules only make sense when the foundational logic *FL* has standard logical implication and negation. As every logic discussed within this work fulfills these conditions, the rules are included as a matter of convenience.

No shorthand rules for truth structures involving beliefs are provided because such structures are not part of the common nomenclature of classical logics; there is no standard syntactic simplification. Sentences involving beliefs are never abbreviated.

2.7 Functional Kinds

Functional kind are broken down into a few primary classes: general functions, computable functions, and interpretations.

2.7.1 Functional Kind

Definition 3 (Function) *A **functional kind** is a kind that represents a function. The general symbol that is used for functions is* $\rightarrow$*. A functional kind that embodies all functions that map realizations of a **domain** kind* D *to realizations of a **codomain** kind* C *is denoted by* $D \rightarrow C$*.*

The functional kind $D \rightarrow C$ classifies functions only by their signature (domain plus codomain), but specializations of this basic kind add additional structure, thus refining classification.

For example, “functions on integers that return odd numbers” and “functions on natural numbers that return prime numbers” are two functional kinds that classify functions by specifying more structure than signature.

2.7.2 Computable Kind

Definition 4 (Computable) *Computable functional kinds are **subkinds** of functional kinds; that is to say, a computable kind* is-a *functional kind. Computable kind are denoted with the symbol* $\dashrightarrow$*. Computable functions are exactly that—computable functions in the Church/Turing sense.*

2.7.3 A Formal Definition of Structure

Informally, structure is characterized by the properties of an object, the relationships in which it is involved, and the manner in which it is interpreted by an observer.

Definition 5 (Structure) *The **structure** of an asset U consists of*

- *all substructures P such that $P \subset_p U$,*
- *if U is a kind, all functional kind of the form $L \to M$ such that $U \subset_p L \to M$, and*
- *if U is an instance, all functional realizations of the form $L \to M$ such that $U \subset_p L \to M$, and*
- *all truth structures T such that $U \subset_p T$.*

The important thing to note about the structure of an asset is that it includes all semantics of the asset via its truth structures. Thus, structure-preserving functions are semantics-preserving.

2.7.4 Interpretations

Definition 6 (Interpretation) *An **interpretation** is a computable kind that preserves structure.*

Two sorts of interpretation kinds are defined.

Definition 7 (Partial Interpretation) *A **partial interpretation** is an interpretation that preserves some substructure (possibly none) from its domain.*

Definition 8 (Full Interpretation) *A **full interpretation** is an interpretation that preserves all structure from its domain.*

With respect to preserving semantics, partial interpretations are partial functions. Likewise, full interpretations are complete functions from the same point of view.

Since full interpretations preserve all structure, the semantics, that is, the validity, of all related constructs is maintained across interpretation. Therefore, in a validity-preserving sense, partial interpretations are partial functions.

Partial and full interpretations are not unique; multiple (often infinite) interpretations can exist between two assets.

An instance called ω is introduced to act as the trashcan for all partial interpretations. All structures unpreserved by a partial interpretation are mapped to ω. The concept and use of a trashcan in this manner is borrowed from terminal objects for forgetful functors in category theory and the related constructs of error sorts universal algebras [17, 256].

2.8 Operators

Core operators like $<$ and $\supset$ listed in Table 2.1 are functional kind. All core operators have both functional and relational representations. Functional representations are the most simplified form of the operators.

2.8.1 Inheritance

Definition 9 (Inheritance Operators) *The computable kind $<_p$ (read as **"parent"**) is the functional form of the inheritance operator. The subscript 'p' denotes parent. This function's domain and codomain are both kinds. When $<_p$ is applied to a specific kind, its direct parent is the result.*

The basic semantics of inheritance are defined in the inference rule (Parent Interp) (see Table 2.10). This rule states that, if $C <_p P$, then a full interpretation $P \rightsquigarrow C$ exists that takes Ps to Cs, and a partial interpretation $C \hookrightarrow P$ exists that takes Cs to Ps.*

Additionally, $\hookrightarrow$ is the left inverse of $\rightsquigarrow$. That is to say, if $\hookrightarrow$ is applied to the result of $\rightsquigarrow$, (written as $\hookrightarrow \cdot \rightsquigarrow$), then the identity function on P is the result (written id_P). The full interpretation preserves all structure of the parent in the child, and the partial interpretation from parent to child preserves all structure from the child to the parent that the full interpretation preserved originally.

Inheritance imposes a partial order on kind. The relation that realizes this partial order is denoted by the symbol $<$. Thus, $\Gamma \vdash K <_p L$ implies $\Gamma \vdash K < L$, as captured in inference rule (Parent Is-a). The unique top of this partial order, the kind from which all other kinds inherit, is called $\aleph$. This fact is summarized in the rule (Is-a Univ).

Thus the structure of inheritance is a lattice with a greatest upper bound of $\aleph$.

This definition matches intuition when it comes to notions of inheritance in programming languages, taxonomies, and knowledge representation. Evidence of this is provided in later chapters when this notion is applied to several examples.

2.8.1.1 Forms of Inheritance

While each kind has exactly one parent kind, as discussed in Section 2.4.2.4, parents are often compositional assets. When an asset has a compositional parent, it is participating in a multiple inheritance relation.

Within most classification systems, e.g., taxonomies, hierarchies, and ER diagrams, multiple inheritance has a simple conjunctive semantics: e.g., A is-a (B and C) is equivalent to A is-a B and A is-a C [78, 110, 143, 286, 326].

Multiple Inheritance in Programming Languages. In the domain of programming languages, multiple inheritance semantics are often much more complex [47, 81]. This is unproblematic because the basic composition operators of kind theory have such a weak semantics that they remain models for these more complex notions.

For example, multiple inheritance only occurs in Java with respect to interfaces, as discussed earlier in Section 2.4.2.4. Interface inheritance semantics boils down to signature conjunction, which is just a slightly refined version of the conjunctive semantics of classification systems.

Algebraic denotational semantics has been used to specify the semantics of multiple inheritance in many programming languages [162, 167, 256]. The subsort relation of many order sorted algebras supports abstract data types with multiple inheritance in roughly the sense of object-oriented programming [162]. Algebras have also been used to specify the semantics of ontologies with multiple inheritance in a similar fashion [36].

Kind theory's basic inheritance operator covers these alternatives as well because (a) subsorts relations are reflexive, transitive, and antisymmetric, and (b) the composition operators for sorts are based upon avoiding signature collision ("no junk") and fulfilling semantic obligations ("no confusion"). The former is an extended form of conjunctive semantics. The latter is realized by algebraic signature enriching morphisms which are models of interpretations due to their coincident model-preserving (via validity) nature.

Functional Inheritance. Functional inheritance comes in three forms: covariant, contravariant, or invariant. Method/function inheritance in object-oriented and module-based languages consequently sees these three forms as well.

Covariance and contravariance characterize two different mechanisms: subtyping and specialization. Castagna has shown that both can exist within a type-safe calculus [50]. Early work using the object-as-records model (e.g., [47] and many of the papers in [177]) nearly uniformly supported only contravariant specialization. But few programming languages support a contravariant type system (e.g., Sather is a rare example). It is widely agreed that a combination of covariant subtyping with an unsound type checker is more natural, flexible, and expressive [50].

This is another reason why kind theory's inheritance operator has a substitution-based semantics. Since contravariant typing is so important, particularly in functional systems, a possible extension to kind theory is a new operator that has such semantics. More investigation is necessary to understand how to model such inheritance, and whether it is important. This limitation is rarely problematic as only a handful of research languages (e.g., Sather), and no mainstream languages, have contravariant semantics. This possibility is discussed in more detail in Chapter 8.

Operational Semantics. Operationally, new kinds are classified with inheritance relations using truth structures. Thus, the sentence $\gamma_A(\top(K <_p L)|)$ states (operationally) that agent A claims that the kind K's parent is L for no reason beyond the sentence itself (i.e., there is no evidence).

2.8.1.2 Subtyping and Subclassing

Within software systems, the primary two types of classification use are subtyping and subclassing [245, 246, 263]. Both can be modeled with kind theory's inheritance operator, though not with full detail or generality. As subclassing is primarily modeled using inclusion, it will be discussed in the next section.

Subtyping. Subtyping is characterized by several substitution principles.

The general principle is attributed to Liskov. It simply states that if each object o_1 of type S, there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2, then S is a subtype of T [244].

Alternative generic principles exist within alternative formalisms. Examples include algebraic subsorts and categorical object inheritance [93, 162]

These generic notions map to kind theory's inheritance because they focus on substitutability, which is checked via validating semantics post-substitution. Such validity preservation is represented via the full interpretation that exists between parent and child.

Other more specific notions fall within the realm of type theory. Many of these alternatives fulfill the basis semantics. Examples include behavioral subtyping, interface inheritance, and specification subtyping [232, 245, 246].

Some subtype relations in programming languages like Java, Eiffel, and C++ can also be modeled from a type system (substitutability) and operational perspective.

The latter is evident when one considers the accessibility of parent methods via a "super"-like mechanism (e.g., "Precursor" in Eiffel). It is not clear if those situations where parent methods are completely hidden (e.g., in Eiffel when using the "undefine" keyword in an "inherit" clause) can be modeled.

2.8.2 Inclusion

Containment is captured by a function/relation pair as well.

Definition 10 (Inclusion Operators) *The computable kind $\subset_p$, read as **"is-part-of"**, is the functional dual to the **has-a** relation $\supset$. Part-of returns the unique immediate enclosing container of*

the construct to which it is applied. Thus, $\Gamma \vdash P \subset_p W$ *implies* $\Gamma \vdash W \supset P$*, as summarized in rule (Part-of Has-a).*

Part-of is applicable to kinds and instances. When applied to a kind (instance), it returns a kind (instance).

The $\subset_p$ *operator imposes partial orders on assets, and it is realized by the* $\supset$ *operator.*

An asset a can be a part of more than one whole through the use of composition. For example, a basic form of composition is logical conjunction. Thus, to specify that a is part of b and c, logical conjunction ("$\wedge$") is defined as a subkind of composition. Then the statement $a \subset_p (b \wedge c)$ is made.

There is no notion of aliasing here because each part of a whole is directly accessible via the has-a operator by rule (Part-of Has-a). Practically, each part has a different name, perhaps realized by an index; the details are immaterial.

2.8.2.1 Shared Resources, Exists Dependency, Subclassing

Three of the most familiar notions of inclusion are shared resources, exists dependency, and subclassing.

Shared Resources. Resource sharing is often modeled with some form of inclusion. For references to generic resources to be shared, some form of name or other reference is necessary. If references can be copied or duplicated, then aliasing is a potential problem.

Resource allocation and permissions are often modeled with tokens, ACLs, or some other form of access control [313]. Many of these models are based upon the assumption that, if the resource itself cannot prevent access, then well-behaved clients will mediate through some protocol involving inviolate objects like tokens. Thus, the uniqueness property is transferred from the resource to the key, and this is modeled in a straightforward fashion with inclusion given its lack of aliasing and name duplication.

Exists Dependency. The exists-dependency paradigm is another model for inclusion and access. It focuses on life-cycle dependencies; a child object can only exist so long as its parent object(s) exist [341].

A subkind of inclusion must be defined to realize this semantics.

Subclassing. Some aspects of programming language inheritance must be modeled by a refinement of kind theory's inclusion operator, or some combination of inheritance and inclusion. If some notion of structure-preserving is evident, then inheritance is involved. This includes

those concepts that can be thought of as "source code cut-and-paste" (ad hoc reuse) that is at the core of subclassing.

Several generic notions of such operators exist including basic subclassing [168, 238], descriptive subclassing [328], parameterized/generic classes (as in CLU, Ada, C++, the OBJ family, Eiffel, and others [15, 147, 243]), delegation [255, 348], and prototypes and exemplars [45, 101, 223, 241].

The fact that these concepts have a common semantics is of no surprise to many who have studied generic notions of inheritance (e.g., see [191, 348]).

Other Whole-Part Associations. Civello and others state that whole-part associations have differing semantics in the analysis and design phases of software construction [68]. This viewpoint is reinforced by the existence of many modeling languages that have one or more whole-part association constructs (e.g., OOAD, OOA, Fusion, OPEN, Objectory, OMT, UML, etc.) [44, 75, 76, 79, 121, 196, 324, 325]. See [277] for a summary.

2.8.3 Equivalence

First, the general notion of equivalence is defined.

Definition 11 (Equivalence Operators) *Any relation that is reflexive, symmetric, and transitive is an* ***equivalence relation****. The symbol* $\equiv$ *is used to denote this sort of relation.*

Given an equivalence relation E *defined on kind, the* ***equivalence class*** *of a kind* K *with respect to* E *is defined as*

$$[K]_E = \{K' \mid \Gamma, K, K' \vdash \diamond \quad \textit{and} \quad (K, K') \in E\}$$

The relational form of equivalence is $K \equiv_E L$ *if and only if* $L \in [K]_E$.

The rules summarizing equivalence are found in Table 2.6.

2.8.3.1 Equality

The symbol = within kind theory in a very basic fashion.

Definition 12 (Equals) *The sentence* $M = N$ *is true exactly when* M *is* ***syntactically equal*** *(symbol-for-symbol identical) to* N.

2.8.3.2 Canonical Forms

Canonical forms play a key role in kind theory. It is important to distinguish between full interpretation, full equivalence, and canonicality.

Associated with every equivalence operator e is a function $[]_e$, read as "**canonical**". Given a kind (instance) it returns the **canonical form** of that kind (instance). Canonical functions are defined as follows.

Definition 13 (Canonical Functions) *Any interpretation* $[]$ *that preserves* ***all*** *structure of its domain (recall Definition 5) is a* ***canonical function****. The canonical form of a canonical form is itself.*

Canonical functions are written using the syntax $[k] = l$. If k is a kind, the kind l is called the **canonical kind** of k. Likewise, for instances, the term **canonical instance** is used. The general term (kind or instance) is **canonical realization** or **canonical asset**.

Multiple canonical functions can exist for any given kind as it can be a part of multiple equivalence classes. It is **possible** to define a full interpretation between any two canonically equivalent assets, but not **necessary** that such interpretations exist. Thus, canonicality induces an equivalence relation; interpretations in general do not.

The rules summarizing canonical's semantics are found in Table 2.10.

Discussion. Canonical functions are interpretations by definition. Recall from Definition 6 that all interpretations are computable. Thus, all canonical functions are computable. During the course of defining a new equivalence relation, a step that often goes along with defining a new kind, a computable canonical function must be defined. If one cannot be defined, the equivalence relation is unsound within KT.

Most generically, canonical functions are validity-preserving functions. Categorically, they are full functors, and algebraically, they are homomorphisms. Within many (type, algebraic, etc.) theories, they are realized by some form of alpha-renaming.

Examples.

Example 2 (Examples of Canonical Forms) *Legitimate canonical forms include the following examples:*

- *Syntactic changes in instances (e.g., representation changes).*
- *Alternate structural forms of a programming language concept. (e.g., a loop, as seen in detail in Chapter 6).*

- *Type-equivalent optimization structures within a compiler. (e.g., the compiler optimization example in Section 6.1.4)*
- *Behaviorally equivalent theoretical (algebras, ADTs, etc.) or technical (software components, documents, etc.) structures (e.g., see Chapters 6 and 7 for several examples).*

***Illegal** canonical forms include*

- *Assume the canonical form of a realization of a function is its value at one point or all points. Such a canonical form throws away too much information about the original function. While functional equivalence can be definable in such a manner, canonicality cannot.*
- *Any non-invertible map cannot be a canonical interpretation, because such would violate the possibility of change in canonical kind. Consider that an agent can redefine at any time which asset is considered canonical by replacing the definition of canonical in the current context.*

2.8.3.3 Full and Partial Equivalence

Also associated with every kind K are two equivalence relations: *full equivalence* denoted $\doteq_K$, and *partial equivalence* $\lessdot_K$.

Definition 14 (Full Equivalence) *An asset U is **fully equivalent** to an asset V if and only if $[U] = [V]$. This is written as $U \doteq V$ and it is read as "U and V are **fully equivalent**".*

Definition 15 (Partial Equivalence) *An asset U is **partially equivalent** to an asset V if and only if $[U] \supset V$ or $[V] \supset U$. This is written as $U \lessdot V$ and it is read as "U and V are **partially equivalent**".*

The semantics of full and partial equivalence are summarized in Table 2.10.

2.8.4 Realization

Realization, defining an instance of a kind, is represented by the functional ":" and the relational $<_r$.

As discussed earlier, for an instance to be a realization of a kind, it must cover all of the kind's substructures.

Definition 16 (Realization Operators) *function ":" is presented an instance, it returns its unique immediate kind.*

The relation $<_r$ is the functional closure of ":" with $<$. Thus $I <_r P$ just in case $I : K$ and $K < P$.

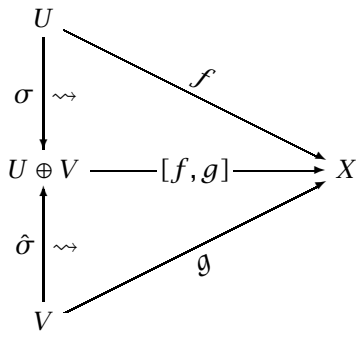
The semantics of realization are summarized in Table 2.7.

2.8.5 Composition

A *compositional asset* is an asset that is composable from parts. Composition comes in three main forms: product ($\times$ and $\otimes$), coproduct ($+$ and $\oplus$), and the general notion of composition ($\cdot$ and $\circ$). Only binary compositional and decompositional assets are defined since N-ary composition can be defined recursively with these dyadic functions.

Definition 17 (Compositional Assets) *An asset $U \oplus V$, together with a pair of full interpretations $\sigma : U \oplus V \rightsquigarrow U$, $\hat{\sigma} : U \oplus V \rightsquigarrow V$ is called a **compositional asset** of U and V if, for each asset X, and each pair of functional asset $f : U \to X$ and $g : V \to X$, there is exactly one functional asset $h : U \oplus V \to X$ for which both $f = h \cdot \sigma$ and $g = h \cdot \hat{\sigma}$.*

Figure 2.3: Compositional Assets



The relation $\oplus$ is the relational dual of the function $+$. If $U + V = W$, then $U \oplus V \equiv W$, $\sigma : U \to W$, and $\hat{\sigma} : V \to W$.

Definition 18 (Compositional Operators) *An operator $\oplus'$ that is a subkind of the basic compositional operator $\oplus$ is called a **compositional operator**.*

To reiterate, defining a new compositional operator $\oplus'$ means (a) subkinding the basic $\oplus$ operator, and (b) introducing the two full interpretations, σ and $\hat{\sigma}$, that map the parts that make up a composition into the whole.

This functional kind h, since it is uniquely determined by f and g can be denoted by $[f, g]$. The full interpretations σ and $\hat{\sigma}$ are called **join** functions for this compositional asset.

These joins are full interpretations because they preserve all structure from their domain asset, and are total functions because they operate on all assets of the appropriate instance or kind. Recall that, because they are interpretations, they are necessarily computable.

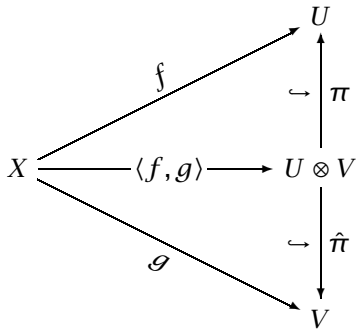
The preceding definition shows a first use of diagrams. They are used to visually summarize contexts. Points indicate assets and arrows indicate functions, both of which are labeled as

necessary. These diagrams are used for proofs as well in the next chapter.

A *decompositional asset* is an asset that is decomposable into parts. Decompositional assets are the dual of compositional assets.

Definition 19 (Decompositional Assets) *An asset $U \otimes V$, together with a pair of partial interpretations $\pi : U \otimes V \hookrightarrow U$, $\hat{\pi} : U \otimes V \hookrightarrow V$ is called a **decompositional asset** of U and V if, for each asset X, and each pair of functional asset $f : X \to U$ and $g : X \to V$, there is exactly one functional asset $h : X \to U \otimes V$ for which both $f = \pi \cdot h$ and $g = \hat{\pi} \cdot h$.*

Figure 2.4: Decompositional Assets



The relation $\otimes$ is the relational dual of the function $\times$. If $U \times V = W$, then $U \otimes V \equiv W$, $\pi : W \to U$, and $\hat{\pi} : W \to V$.

Definition 20 (Decompositional Operator) *An operator $\otimes'$ that is a subkind of the basic decompositional operator $\otimes$ is called a **decompositional operator**.*

This functional kind h, since it is uniquely determined by f and g can be denoted by $\langle f, g \rangle$. The partial interpretations π and $\hat{\pi}$ are called **projection** functions for this decompositional asset.

The semantics of composition are summarized in Table 2.9.

2.8.5.1 Comments on Composition

These definitions of compositional and decompositional assets and operators are derived from those of categorical of products and coproducts. Formalizations are also found in all basic mathematics (e.g., set theory's intersection and union, logic's conjunction and disjunction, etc.), all of which are models for the categorical notions.

Composition, as a notion, has seen intense scrutiny in the programming languages community (e.g. Nierstrasz's extensive work [250, 293, 294, 296, 295], categorical and algebraic [118,

164, 166, 261, 322], predicate calculus-based [61, 58], knowledge-based [332], software architecture [34, 35, 57, 138], interaction/metaobject-style [182, 226, 284], and other approaches [281, 302]. This is particularly true of the component and object subcommunities as well as the more esoteric fusion community [183, 189, 352].

Now that compositional assets and operators have been defined, let us look at the inference rules and axioms of kind theory.

2.9 Inference Rules and Axioms

The inference rules and axioms of *KT* are the basic formula from which all theorems are derived. Since the formal definitions of the core operators of kind theory have already been provided in Section 2.8, operator semantics will only be summarized here with inference rules.

2.9.1 Structural Rules

It is presumed that the foundational logic *KT* provides basic structural rules like thinning, exchange, etc., so they are not detailed here.

2.9.2 Inheritance Rules

Table 2.4: Inheritance Rules

(Parent Is-a)
$$\frac{\Gamma \vdash K <_p L}{\Gamma \vdash K < L}$$

(Is-a Ref)
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash K < K}$$

(Is-a Trans)
$$\frac{\Gamma \vdash K < L \quad \Gamma \vdash L < M}{\Gamma \vdash K < M}$$

(Is-a Asym)
$$\frac{\Gamma \vdash K < L \quad \Gamma \vdash \gamma(\bot(K \equiv L)|)}{\Gamma \vdash \gamma(\bot(L < K)|)}$$

The rules of inheritance are summarized in Table 2.4. As stated earlier, subkinding is a reflexive, transitive, and asymmetric relation.

2.9.3 Inclusion Rules

Inclusion rules are summarized in Table 2.5. As mentioned earlier, inclusion is a reflexive, transitive, asymmetric relation.

Table 2.5: Inclusion Rules

$$\text{(Part-of Has-a)}\quad \frac{\Gamma \vdash U \subset_p V}{\Gamma \vdash V \supset U} \qquad \text{(Has-a Ref)}\quad \frac{\Gamma \vdash \diamond}{\Gamma \vdash U \supset U}$$

$$\text{(Has-a Trans)}\quad \frac{\Gamma \vdash U \supset V \quad \Gamma \vdash V \supset W}{\Gamma \vdash U \supset W} \qquad \text{(Has-a Asym)}\quad \frac{\Gamma \vdash U \supset V \quad \Gamma \vdash \gamma(\bot(U \equiv V)|)}{\Gamma \vdash \gamma(\bot(V \supset U)|)}$$

2.9.4 Equivalence Rules

Table 2.6: Equivalence Rules

$$\text{(Equiv Ref)}\quad \frac{\Gamma \vdash \diamond}{\Gamma \vdash U \equiv U} \qquad \text{(Equiv Trans)}\quad \frac{\Gamma \vdash U \equiv V \quad \Gamma \vdash V \equiv W}{\Gamma \vdash U \equiv W} \qquad \text{(Equiv Sym)}\quad \frac{\Gamma \vdash U \equiv V}{\Gamma \vdash V \equiv U}$$

Table 2.6 holds no surprises. The operator $\equiv$ in these rules represents any equivalence relation. Equivalence is reflexive, transitive, and symmetric as usual.

Note that, because of the shorthand rules, some of these rules look particularly terse. For example, because U is used on the right of the turnstile in the lower half of (Equiv Ref), then the left side of the turnstile is a shorthand for something of the form Γ', U. Thus, the left side of the top half of the same rule would read Γ', U as well. But, as already mentioned, that which is explicitly mentioned on the right of a turnstile need not be explicitly mentioned on the left, thus a rule of the form shown above is the result.

2.9.5 Realization Rules

Table 2.7: Realization Rules

$$\text{(Isakindof)}\quad \frac{\Gamma \vdash I : K \quad \Gamma \vdash K < P}{\Gamma \vdash I <_r P} \qquad \text{(Realizes A)}\quad \frac{\Gamma \vdash I : K \quad \Gamma \vdash P \subset_p K \quad \Gamma \vdash J}{\Gamma \vdash J \subset_p I} \qquad \text{(Realizes B)}\quad \frac{\Gamma \vdash I : K \quad \Gamma \vdash P \subset_p K \quad \Gamma \vdash J}{\Gamma \vdash J : P}$$

These two rules summarize realization. (Isakindof) describes the relationship between the functional and relational forms of realization. (Realizes) summarizes the relationship between instances and their kinds. It states that if an instance I is of kind K, and P is some substructure of K, then I necessarily has some substructure J that is of kind P.

2.9.6 Interpretation Rules

Table 2.8: Interpretation Rules

$$\text{(PartInterp Trans)}\quad \frac{\Gamma \vdash U \underset{\Lambda}{\overset{A}{\hookrightarrow}} V \quad \Gamma \vdash V \underset{\Lambda}{\overset{A}{\hookrightarrow}} W}{\Gamma \vdash U \underset{\Lambda}{\overset{A}{\hookrightarrow}} W} \qquad \text{(FullInterp Trans)}\quad \frac{\Gamma \vdash U \underset{\Lambda}{\overset{A}{\rightsquigarrow}} V \quad \Gamma \vdash V \underset{\Lambda}{\overset{A}{\rightsquigarrow}} W}{\Gamma \vdash U \underset{\Lambda}{\overset{A}{\rightsquigarrow}} W}$$

Interpretation has only two rules, presented in Table 2.8. Interpretation, both partial and full, is transitive. For transitive interpretations, their agents and contexts must be identical, in their most general form.

2.9.7 Composition Rules

Table 2.9: Composition Rules of Kind Theory

$$\text{(Cocompose Joins Left)}\quad \frac{\Gamma \vdash U + V}{\Gamma \vdash \sigma : U \rightsquigarrow U + V} \qquad \text{(Cocompose Joins Right)}\quad \frac{\Gamma \vdash U + V}{\Gamma \vdash \hat{\sigma} : V \rightsquigarrow U + V}$$

$$\text{(Prcompose Projections Left)}\quad \frac{\Gamma \vdash U \times V}{\Gamma \vdash \pi : (U \times V) \hookrightarrow U} \qquad \text{(Prcompose Projections Right)}\quad \frac{\Gamma \vdash U \times V}{\Gamma \vdash \hat{\pi} : (U \times V) \hookrightarrow V}$$

$$\text{(Coproduct Cocompose*)}\quad \frac{\Gamma \vdash U \oplus V}{\Gamma \vdash U + V = U \oplus V} \qquad \text{(Product Prcompose*)}\quad \frac{\Gamma \vdash U \otimes V}{\Gamma \vdash U \times V = U \otimes V}$$

$$\text{(Composition*)}\quad \frac{\Gamma \vdash U \otimes V \quad \Gamma \vdash U \oplus V}{\Gamma \vdash U \circ V}$$

The rules in Table 2.9 summarize the semantics of composition.

2.9.8 General Rules

The rules in Table 2.10 are the keystones of the theory and understanding them is critical to comprehending and using it.

The rule (Parent Interp*) states that, if an inheritance relationship exists between kind, then two interpretations must also exist: one that takes the parent to the child, preserving all

Table 2.10: General Inference Rules of Kind Theory

$$\text{(Parent Interp*)}\quad \frac{\Gamma, L \rightsquigarrow K, K \hookrightarrow L \vdash \diamond \quad \Gamma \vdash K <_p L}{\Gamma \vdash L \rightsquigarrow K \cdot K \hookrightarrow L = id_L}$$

$$\text{(Ground*)}\quad \frac{\Gamma \vdash K <_p \aleph}{\Gamma \vdash \gamma(\top \nabla K|)} \qquad \text{(Is-a Univ)}\quad \frac{\Gamma, K \vdash \diamond}{\Gamma \vdash K < \aleph}$$

$$\text{(Fully Equiv*)}\quad \frac{\Gamma \vdash K \backsimeq L}{\Gamma \vdash [K] = [L]} \qquad \text{(Partial Equiv*)}\quad \frac{\Gamma \vdash K \lessdot L}{\Gamma \vdash [L] \supset [K]}$$

structure, and a left inverse of that map that takes the child to the parent. This rule essentially subsumes the related notions of type coercion, structural type checking, and classification perspective. By (Parent Is-a), $<$ implies $\hookrightarrow$ as a corollary to this inference rule.

(Is-a Univ) states that all kind inherit from the top of the inheritance hierarchy $\aleph$.

The ground kind are identified by the ∇ operator whose semantics is summarized in the reversible rule (Ground*).

Finally, the definitions of full and partial equivalence is summarized in rules (Fully Equiv*) and (Partial Equiv*), as discussed earlier in Sections 2.4.2 and 2.8.3.

2.9.9 Axioms of Kind Theory

Each of the core functional kind (e.g., inheritance, inclusion, etc.) is a function. Since functions are composable, so are functional kind. Recall that K represents arbitrary classifying kind, and I an arbitrary instance.

Functional kind composition obeys the following identity and associative laws:

$$K \cdot 1_K = 1_K \cdot K = K$$

$$I \cdot 1_I = 1_I \cdot I = I$$

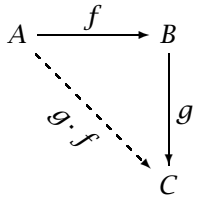
$$K \cdot (L \cdot M) = (K \cdot L) \cdot M$$

Whenever there are two functional kinds $f : A \to B$ and $g : B \to C$ such that the codomain of the first equals the domain of the second, their composite $g \cdot f$ is defined as the functional kind $A \to C : a \to g(f(a))$. The semantics of this composition are "do f, then g".

Diagrammatically, this is depicted as seen in Figure 2.5.

Thus, the first (and only) axiom of kind theory.

Figure 2.5: Basic Compositionality of Functional Kind



Definition 21 (Compositionality of Functional Kind: CFK) *If two functional kinds are composable in a context, then their composition exists in that context.*

$$\frac{\Gamma, K \to L, L \to M \vdash \Delta}{\Gamma, K \to L, L \to M, K \to M \vdash \Delta, (K \to M) \equiv (L \to M \cdot K \to L)}$$

This rule not only states that such a functional kind must exist, but also specifies that it is equivalent to the functional composition of the original two functional kind.

Since $\to$ is used in this definition, any functional kind suffices. This includes all basic operations (e.g., $<_p, \subset_p, \cdot, \times, +$, etc.) and all extensions to such notions (via inheritance, as discussed below).

2.9.10 Some General Notes on the Inference Rules of Kind Theory

Most of the basic theorems of kind theory are summarized in the next chapter. Some simple semantics that seem necessary are not obvious given this set of inference rules. This section reviews these issues.

2.9.10.1 The Interplay between Inheritance and Equivalence

The standard relationship between inheritance and equivalence exists in kind theory. Theorem 1 shows this connection.

Theorem 1 (Inherit Equiv)

$$\frac{K < L \quad L < K}{K = L}$$

Proof. Inheritance defines a poset on kind as specified by the definition of the $<_p$ function and $<$ relation. Therefore, the result follows by the definition of a poset. □

2.9.10.2 The Interplay between Composition and Inclusion

The relationship between composition and inclusion is indirect, but intuitive.

The properties of a whole depend upon both the identities of its parts as well as how they are put together. Within the basic framework of kind theory, all that can be said about a whole is that it maintains all of the structure of its parts. This is true by virtue of the fact that joins are full interpretations.

The substructures mapped to by joins might not be distinct. If this is the case, then some mixing has occurred, and it is likely that no decomposition operator can separate the parts again. This is why separate composition and decomposition operators are defined.

This relationship is summarized in the following basic theorem.

Theorem 2 (Compose Inclusion)

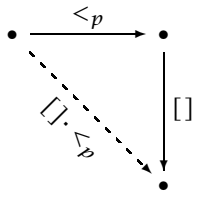
$$\frac{\Gamma \vdash K + L \quad \Gamma \vdash P \subset_p K}{\Gamma \vdash P \subset_p K + L}$$

Because injections into compositions maintain all substructure, then all parts of the composed assets are part of the whole as well.

2.9.11 Identifying Inferred Functional Kinds

Often, but not always, the rules of kind theory can derive the identity of an unknown compositional functional kind. Consider the diagram in Figure 2.6.

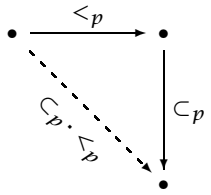
Figure 2.6: Known Functional Composition



The compositional functional kind denoted by $[] \cdot <_p$ in the diagram is in fact the functional kind $<$ due to Theorem 8.

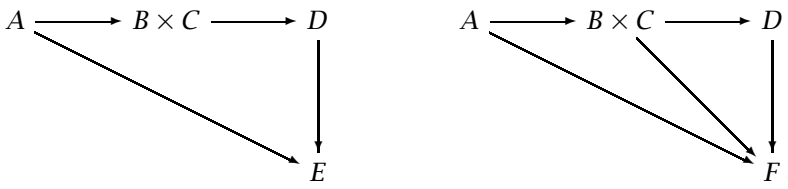
In other cases, the composition is not yet named or identified by a theorem. Consider the diagram in Figure 2.7. The map $\subset_p \cdot <_p$ must exist due to the CFK axiom. But, as of yet, the identity of this compositional functional kind has not been determined. Thus, until such time that a new functional kind is named, introduced, and defined as equivalent to this composition, the function must be named $\subset_p \cdot <_p$.

Figure 2.7: Unknown Functional Composition



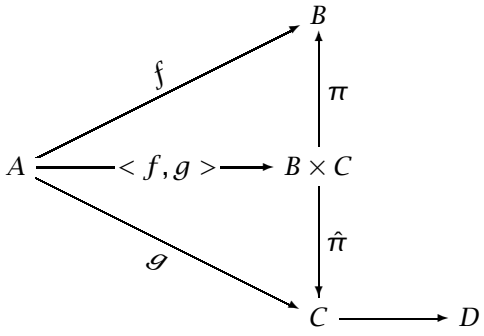
2.9.12 Functional Kind Matching

Figure 2.8: Matching Examples



Consider the kind $A \to B \times C \to D$. This kind composes with the kind $D \to E$ because the final codomain of the first kind matches with the domain of the second. This composition is shown in the left-hand diagram of Figure 2.8. This kind also matches the kind $B \times C \to D \to F$, as seen in the right-hand diagram of the same figure.

Figure 2.9: Another Matching Example



Likewise, products can be reinterpreted as functions[3]. Consider Figure 2.9. The functional

[3]Such reinterpretation is used in variants of the ML programming language, in the specification languages Clear [46] and OBJ3 [149]. This viewpoint is also considered in Chapter 5's discussion of parameterization.

kind $A \to B \times C$ matches with $C \to D$ because interpretations π and $\hat{\pi}$ exists such that $B \times C \xrightarrow{\pi} B \to C$ and $B \times C \xrightarrow{\hat{\pi}} C \to B$. Thus the original $A \to B \times C \to D$ can match, for example, $A \to B \to C \to D$.

2.10 Context Modification

There are only four ways to change a context: add or remove a kind, or add or remove an instance. Rather than specify four different semantics for these context-modifying functions, a single semantics is stated that covers all four.

Before discussing these concepts, a few words about contexts.

First, many sequent-based systems do not define remove operations on a logical context. Such operations are, in a sense, superfluous because removing an element is the same as reconstructing the context without that element.

Second, many of the concepts relating to context contraction (e.g., cuts, elimination rules, etc.) are viewed as niceties, but not necessities. All make for a logic that is (subjectively) easier to use, but no stronger than the logic without the rules.

Finally, contexts are meant to capture the state of their agents' knowledge. As such, as knowledge changes (over time, perhaps), so can a context. While such temporal notions are not necessary since there is no fundamental notion of time or monotonicity here, and thus some basic indexing mechanism on contexts might have use, contexts are not tagged thusly at this time.

Definition 22 (The Context Change Function) *The context change function Δ maps $C \times A \times O \dashrightarrow C$, where A represents the asset to manipulate and O the operation to perform (add/remove) on said asset. Such is represented concisely as Δ_u^+ for "add asset u" and Δ_u^- for "remove asset u".*

Δ is an interpretation on contexts. If O is a remove operation, then Δ is a partial interpretation*. In particular, the only structure forgotten by the remove operation is exactly that structure associated with the removed asset. If O is an asset addition operation, then Δ is a normal* full interpretation *on the context.*

The structure that these interpretations preserve (maintain across application in the sense of Section 2.7.4) are exactly those truth structures involving assets in the context that are not directly referenced in the context change. In other words, the structure not maintained by the removal function Δ_u^- is exactly the set of truth structures involving u.

2.10.1 Instance-Related Operations

Examining some specific examples of manipulating contexts will help us get a handle on this definition. The basic case is considered first: adding an instance to a context.

2.10.1.1 Adding an Instance

Example 3 (Adding an Instance to a Context) *Instances are added to a context by declaring a new name i and assigning a kind k to the new instance. The context change function that represents this operation is Δ_i^+. The only direct connection with Γ possible for an instance is the realization operator ":".*

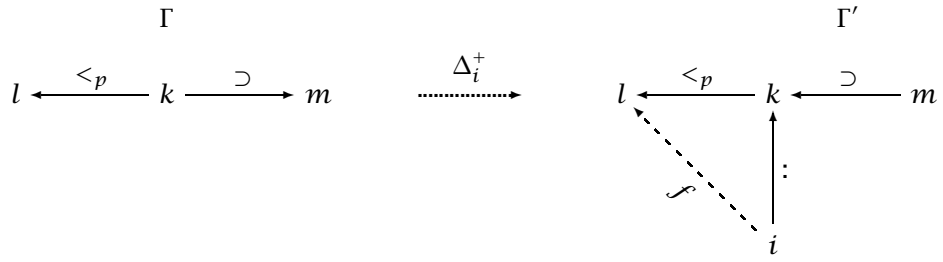


Figure 2.10: An Example of Instance Addition

By the inference rule CFK*, all relationships that k has with other assets are commuted to i. Thus, in Figure 2.10, the functional realization $f : i \mapsto l$ exists.*

In Chapter 3, every pairwise combination of functional kind compositions is analyzed. By Theorem 16 of that chapter, f is the relation $<_r$. In this example, the kind k is also involved in a relation with m. The inference rule CFK *does not help elucidate this relation because there is no opportunity for commutativity. Instead, the rules and theorems of kind theory must be examined to understand if any relation exists between i and m.*

A context does not automatically include all instances derived from a particular kind. For example, a context Γ that has a kind called NATURALNUMBER does not automatically contain the instances $1, 2, 3, \ldots$. Those instances need be added, intentionally (e.g., via a sequence of instance definitions) or extensionally (e.g., via something akin to Peano arithmetic), to the context before they are valid.

2.10.1.2 Removing an Instance

Example 4 (Removing an Instance from a Context) *Removing an instance, on the other hand, does not require any inference or logic rules. Removal is like thinning, albeit a complex form of such.*

The partial interpretation Δ_i^- removes a specific instance i and "forgets" (i.e., destroys) all functional realizations involving i that are no longer well-formed. Only explicit relations between neighbors of i, not those due to CFK*, still hold.*

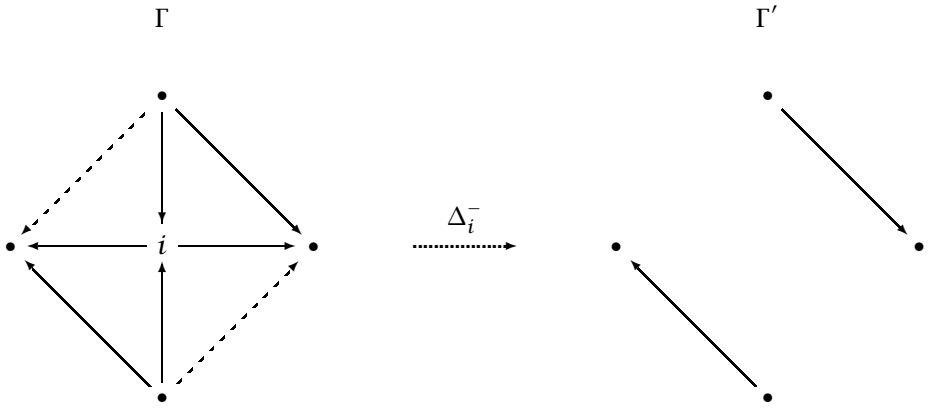


Figure 2.11: An Example of Instance Removal

Consider the diagram in Figure 2.11. Since the instance i is being removed from context Γ*, all functional realizations involving i (the solid arrows touching i on the left side of the figure) are removed as well. Functional realizations due to* CFK *(dotted arrows) which were secondary in* Γ *are removed in* Γ' *(the right side of the figure). Functional realizations that were independent of i, perhaps even those that were originally due to* CFK *(the solid diagonals) but then were made independent of it via kind or instance addition, remain in the new context.*

2.10.1.3 Removing a Compositional Instance

Compositional instances and their manipulation are considered next.

Example 5 (Removing a Compositional Instance from a Context) *Loss of structure is witnessed for compositional instances as well. Consider the diagram in Figure 2.12. The instance i is removed, and such implies that the compositional instance $i \circ j$ can no longer exist ant it is removed. All maps to either instance are destroyed, and any* CFK *or rule-induced functional realizations dependent on i are removed as well. The resulting context is* Γ' *on the right of the figure.*

2.10.2 Kind-Related Operations

Kind-related context manipulation operations are examined next.

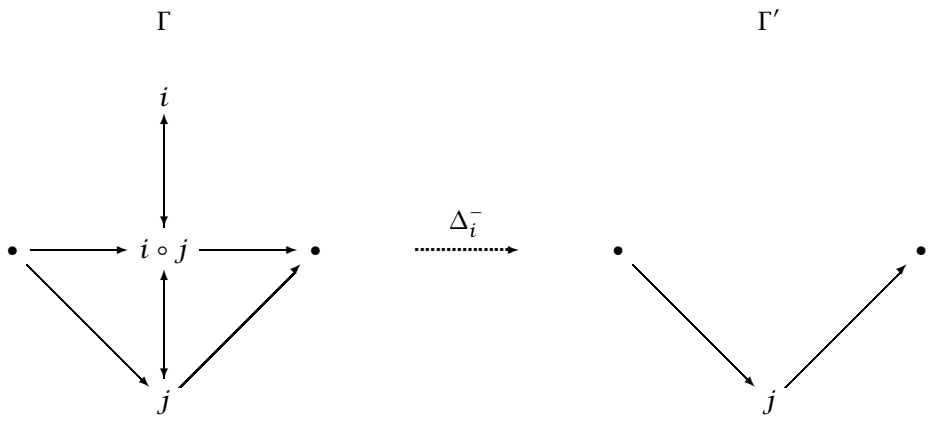


Figure 2.12: An Example of Compositional Instance Removal

2.10.2.1 Removing a Kind

Removing a kind is like removing an instance, except the impact on the context can be larger. Because kind are often involved in more relations than instances, the impact of removing one from a context is likely more severe.

Example 6 (Removal of a Kind from a Context) *Consider the diagram in Figure 2.13. Assume all nodes in this diagram are kinds. The removal of leaf kinds like s or t do not have much, if any, impact on the context. On the other hand, the removal of a critical kind, like the derived l or the grounds b or i, causes a cascade of kind removals from the context. E.g., see the right side of the figure.*

This example was not maliciously constructed. This context is a part of the domain of basic structures (lists, bags, sets, etc.). The kind b and i are the ground kind BOOLEAN *and* INTEGER *respectively. u is the universal kind* UNIVERSAL*. l is the parametric kind* LIST *and c is the kind* COMPARABLE*. The other kind in the diagram are illustrative. Thus, the removal of the notion of list causes a cascade of removals until all that is left are some ground kind and* COMPARABLE*.*

2.10.2.2 Adding a Kind

The addition of a new kind k to a context Γ is the most complex Δ-operation. The inference rules involving inheritance uniformly state that relations of a parent kind are preserved by its children. When a kind k is added to a context Γ, by CFK these functional kinds exist, and by the aforementioned theorems, its precise semantics are known. Typically, after k is added, these functional kinds are realized via the addition of new, or references to existing, kinds.

Example 7 (Addition of a Kind to a Context) *Consider the diagram in Figure 2.14.* LIST *has a* COUNT *kind, consequently the new kind* ORDEREDLIST *has a* COUNT *kind as well. If an explicit*

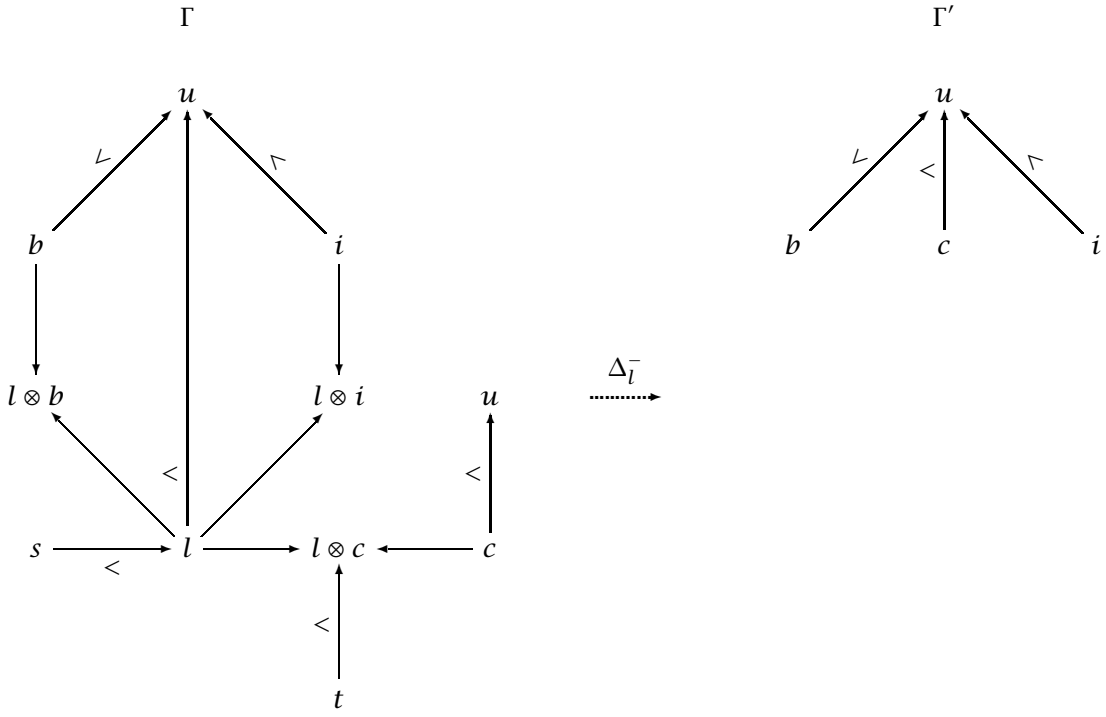


Figure 2.13: An Example of Cascade Kind Removal

COUNT *kind is not provided, then the* CFK*-inferred one is available in the form of* $\supset \cdot \hookrightarrow$*. The kind* ORDEREDLIST *interprets into* LIST*, and thence the map* $\supset$ *applied. If at some later time a new (functional) kind is introduced to replace* $\supset \cdot \hookrightarrow$*, it is known that, by the semantics of* Δ*, that the semantics of* ORDEREDLIST *is preserved and the old* CFK*-inferred map can be removed.*

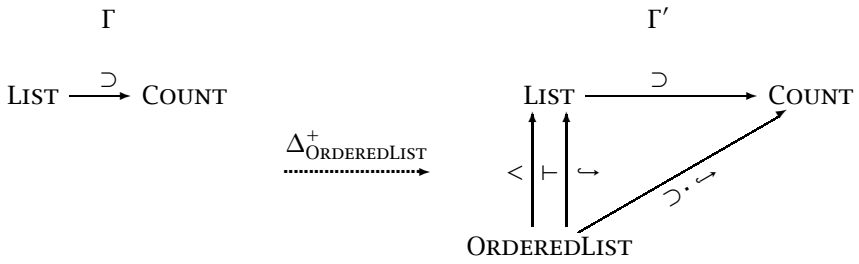


Figure 2.14: An Example of Kind Addition

2.10.3 Context Manipulation Under Equivalence

An important aspect of asset removal is an implication of the explicit equivalence operator $\equiv$ in kind theory. If an asset a is removed from a context Γ and $a \equiv b$ for some $b \in \Gamma$, then all structure dependent upon a is not destroyed.

In this case all references to a become b, because the two assets were fully equivalent and are completely substitutable. All functional kinds are augmented by the appropriate full interpretation.

Example 8 (Kind Removal Under Equivalence) *Consider the diagram in Figure 2.15. The kind a, which is fully equivalent to the kind b, is being removed. But a has a relationship with c via $a \circ c$. h exists due to the definition of product kind. Even if $f \circ h$ did not explicitly exist in Γ, the resulting context Γ' does explicitly contain a new map $h' = f \circ h$ to maintain the semantics of the composite kind under substitution.*

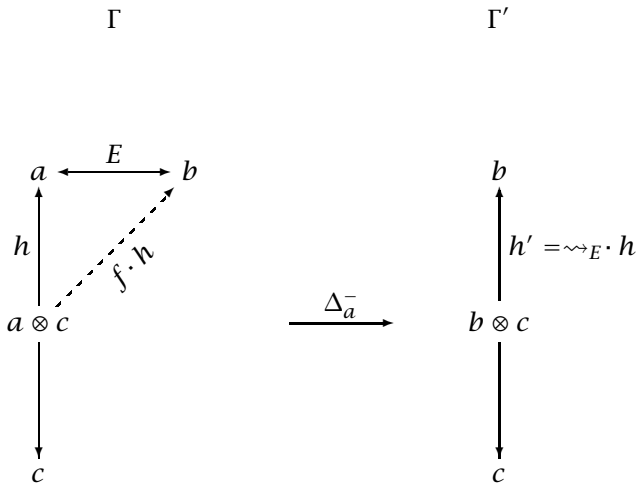


Figure 2.15: An Example of Kind Removal Under Equivalence

2.11 Proofs and Deductions

The notion of proof and deduction in kind theory is defined here for completeness.

2.11.1 Proofs

A formal proof in kind theory is defined in a standard fashion. Chapter 5 covers some of the basics of reflective definitions of proof substructures, (e.g., lemmas, theorems, etc.). Due to these reflective definitions, proofs are representable in a kind-wise fashion.

A proof in kind theory is a sequence of *wffs* of *KT* such that each *wff* is either

- an instance of an axiom of *KT*, or
- an instance of an axiom of the foundational logic of *KT*, or

- derivable from a set of earlier *wffs* in the sequence using one of the rules of inference of *KT*, or
- derivable from a set of earlier *wffs* in the sequence using one of the rules of inference of the foundational logic of *KT*.

The sequence of *wffs* leading to a given *wff* T is said to be "a proof of T in *KT*" and T is "a theorem of *KT*".

2.11.2 Deduction

A deduction from a set of *wffs* Γ in the system *KT* is defined as a sequence of *wffs* such that each *wff* is either

- an instance of one of the rules of *KT*, or
- an instance of one of the rules of the foundational logic of *KT*, or
- one of the hypotheses (either in Γ or a *wff* written on the left of the turnstile), or
- is derivable from earlier *wffs* in the sequence using the rules of inference of *KT*, or
- is derivable from earlier *wffs* in the sequence using the rules of inference of the foundational logic of *KT*.

Effectively, via deduction, theorems are added to the set of rules.

2.12 Semantics

Kind theory's semantics are unusual due to the explicit notion of agent. Agents alter notions of truth because they can stipulate the verity of well-formed sentences. Additionally, because agents free will, truth is necessarily non-monotonic.

The core constructs that define semantics are claims and beliefs. When discussing claims and beliefs, or what are generally called truth structures, the turnstile $\vdash$ becomes the models operator, $\models$, to indicate the shift from syntax to semantics.

The design of semantics statement, manipulation, and verification in kind theory is inspired by, and derived from, the corresponding notions in epistemic logics [63, 129, 358]. Evidence, claims, and beliefs are all generalizations of the shared features of a variety of these popular, nonstandard logics, like those discussed in Section 2.4.4.

2.12.1 The Nature of Evidence

Evidence was defined earlier as a substructure of a context. This definition does not coerce any structure on what is a very important aspect of truth in kind theory.

In fact, evidence does have significantly more structure. For example, evidence often comes in the form of a formal proof of a claim. To this end, in Chapter 5 some of the common (sub)structures of proofs are defined: axioms, lemmas, theorems, corollaries, etc.

Some of these proof structures are formal and are machine checkable. As with most proof systems, checking an existing proof in kind theory has linear computational complexity.

The reason that the semantics of evidence are so loose is that legitimate claims (and beliefs) are not necessarily mathematical statements. For example, valid evidence comes in the form of reference to documents, statements of best practices, and even appeals to common sense.

2.12.2 Claims

A claim is an assertion made by an agent (a person, system, community, etc.) that a given predicate has a specific truth value in a context. Claims are used to impose truth, evaluate judgments, reason about kind, and provide an agent-directed reinforcement mechanism.

A claim does not have any vagueness or sureness. A claim's truth value is either *true*, *false*, or *unknown*. Most statements about knowledge are not claims, they are beliefs. The only predicates that are claims are typically those derived from or by formal systems.

2.12.2.1 Syntax

The shorthand syntax that is used for claims within the kind theory is of the following form (reiterating Table 2.2):

- $\Gamma \vDash \gamma_A(P|E)$ - The agent A claims that within context Γ, and due to evidence E, the predicate P is *true*. The full syntactic form for this sentence is $\Gamma \vDash \gamma_A(\top P|E)$.
- $\Gamma \vDash \gamma_A(\neg P|E)$ - Likewise, but *false*. The full syntactic form for this sentence is $\Gamma \vDash \gamma_A(\bot P|E)$.
- $\Gamma \vDash \gamma_A(?P|E)$ - Likewise, but *unknown*. Recall that unknown claims have no simplified syntax.

2.12.2.2 Theorems as Claims

Theorems, on the other hand, require an agent and evidence to have valid meaning.

The evidence necessary to state a theorem R reflectively, and thus add it to the context Γ, is exactly the proof structure P of the theorem R. The agent A identified in such a claim is either the user that constructed the specific proof P, or a system (e.g., a proof assistant) that generated the proof.

Theorems need not be added to a context. Adding theorems (reflectively stated) complicates context manipulations since the removal of an axiom, rule, or other asset can invalidate enormous numbers of theorems. This is true of all reflective logics with cut rules.

This complexity in manipulation operations is outweighed by the benefits of explicit representation. Proofs in logics with cut rules are shorter and more straightforward. A complexity reduction for asset search and judgment evaluations is also a possibility, as discussed in Chapters 5 and 6.

2.12.2.3 Induced Inconsistency

An absurd claim (a claim of truth value $\perp$) of an axiom or theorem induces an inconsistent context in kind theory. Checking consistency after each context manipulation is best avoided. Such checks are, at best, computationally quite expensive, and at worse, uncomputable.

Additionally, there is value in temporary inconsistency [32, 106, 108, 311]. Such situations are common when manipulating knowledge constructs: consider a judge wrestling over a decision involving conflicting witness testimony, or a mathematician struggling with a new logic that exhibits temporary inconsistency. Additionally, during the design of complex systems, participants often have to work with partial knowledge and conflicting requirements—inconsistency at its finest.

No attempt is made to provide a theoretical infrastructure for reasoning with inconsistency, as it is provided by the many paraconsistent logics that were discussed in Section 2.5. Such logics are some of the most attractive foundational logics for kind theory.

An unknown claim for such an axiom or theorem is not erroneous and does not introduce an inconsistency; it states that the agent does not know if the sentence is valid or not. This is sometimes completely warranted: e.g., the sentence itself is the statement of an important theorem which has not yet been explicitly proven.

2.12.2.4 No-Agent Claims

A context with no agent-bound claims at all (no sentences of the form "$\gamma_a(p|e)$") is the base case for semantics that are considered first.

The only legal sentences that do not reference an agent are axioms. Thus, the agentless context is exactly the initial context of kind theory. For a given model of kind theory, the

initial context contains definitions for all basic constructs necessary for the reflective definition of kind theory, no more and no less. This includes all axioms and inference rules of the foundational logic.

An example of such a model is found in Chapter 5.

2.12.2.5 Single-Agent Claims

Single-agent contexts are the most important piece of the claim evaluation and manipulation puzzle because higher-order and multi-agent claims are all reduced to the single-agent case during judgment evaluation. Sections 2.12.2.6 and 2.12.3.2 discusses these reductions in more detail.

A single agent A making claims imposes few complications because

- There are not any multi-agent resolutions to consider.
- All searches are short-circuited because there are not any trust graphs with which to contend. See Section 2.12.3 for details on trust graphs.
- Inconsistency checking is feasible because the context is always in a stable state when assets are manipulated because no other agent is interacting with the system simultaneously.

The primary issue with claims made by a single agent is the detection and tracking of inconsistencies. When a claim $\Gamma \vDash \gamma_A(\top P|E_A)$ is made, the concern is whether there exists any $\Gamma \vDash \hat{P} \neq P$ such that $\Gamma \vDash \gamma_A(\bot\hat{P}|E_\bot)$ or $\Gamma \vDash \gamma_A(?\hat{P}|E_?)$.

If $\Gamma \vDash \gamma_A(?\hat{P}|E_?)$, then the new claim replaces this weaker one. Such a context change indicates a strengthening in the knowledge of the agent A; it has moved from the condition of not knowing to knowing. This replacement only automatically takes place if E_A and $E_?$ are machine verifiable. In all other situations, it is up to the agent A to decide if the replacement is indeed warranted, or if there is some problem with E_A or $E_?$. This decision is made with a yes/no query.

If $\Gamma \vDash \gamma_A(\bot\hat{P}|E_\bot)$, then there is a more serious situation. If E_A and $E_\bot$ are machine checkable, and both prove to be valid, there is either a problem with the prover(s), or an inconsistency in the logical framework supporting the evidence, perhaps in the foundational logic itself. In neither of these situations can kind theory help—in a sense, it has already done its job detecting the problem in the first place.

If E_A and $E_\bot$ are not machine checkable, then either (a) agent A has made an error and some action must be taken outside of kind theory to rectify that error and accordingly adjust the claim(s), or (b) there is an inconsistency in the logic supporting the evidence; the same

situation as above, or (c) agent A intentionally and knowingly holds a conflicting viewpoint on the predicate.

In this final case the two claims are replaced by two new beliefs that account for the new knowledge state. Each claim of the form $\gamma_A(P|E)$ is replaced by a belief of the form $\beta_A(P|E) = D$ where the magnitude of the belief is addressed by the agent A. In the case of a human agent, such information is gathered via two questions of the form "how strongly do you believe X?", again fulfilling the user-centric goals for the theory.

The key to this claim framework is the ability to determine when two assets U and V are fully equivalent ($U \rightleftharpoons V$).

Under most formalisms, determining when two structures have identical semantics is a difficult, if not impossible, task. The key reason for the introduction of canonicality and full equivalence in kind theory is to not only make such checks possible, but efficient. Since every kind is accompanied by a full interpretation that is used to define full equivalence, and full interpretations are computable functions (by definition), there is a computable (and hopefully efficient) means of determining the validity of $U \rightleftharpoons V$ for all assets.

A model of such computations in discussed in Chapter 4, taking care to consider the computational complexity of such full equivalences.

2.12.2.6 Multi-agent Claims

Multi-agent claims are the most common case when dealing with an open system like that which is proposed here to realize a kind system.

Concurrent access by hundreds of individuals and tools will not be an uncommon scenario in such a system. Thus, efficiently supporting that many simultaneous agents is part of the theory-influencing requirements.

Inconsistency detection and tracking is an issue with multiple agents, but not a serious one. The primary situation that must be dealt with in the multi-agent scenario is the adoption of expert-constructed domains. An agent A is relying upon the expert testimony of an expert agent A_E, importing some of A_E's knowledge into the current context.

Each imported domain contains a finite number of claims and beliefs. The adoption of each truth statement (claim and belief) from the domain is identical to the case where the agent is making the truth statement itself except that (a) conflict resolution of claims involves multiple agents, and (b) second-order belief statements are now part of the problem.

The first issue is resolved in the same manner as discussed in Section 2.12.2.5, except now the decision making agent must take into account the source of each claim as part of the evidence. For example, a proof of a proposition that has been understood and analyzed for hundreds of years holds more weight than one recently discovered.

Discussion on the second issue is deferred since it is covered at length in Section 2.12.3.

2.12.2.7 Community Claims

An agent is not necessarily a single individual or system. A collection of agents when composed together can constitute a new agent representing a community. Such collective constructs are common in multi-agent systems[24, 63, 240, 274, 275].

As mentioned previously, the relative strength of a claim is, in part, determined by its agent. Thus, a claim made by a community of experts can hold more weight than one advocated by a single member of a community.

The key aspect to community claims is that the individual claims from the community members are typically semi-independent. Experts in the community make claims independently, proving and reaffirming the same predicates in many different ways. Decoupling claim results from direct dependence (as so often happens with beliefs, as is discussed in Section 2.12.3) means that the strength of aggregate claims is accumulative (additive or subtractive).

With respect to claims, this additive effect is witnessed in an agent's attempt at introducing a new claim that contradicts a unified community's position. Each and every equivalent claim of that community must be refuted by the the agent to maintain a consistent context. This refutation series is necessarily linear, insofar as the claims are independent, thus invalidating one has no impact on any of the others.

Kind theory does not provide any new mechanisms for dealing with these situations. It only provides a means by which one can model communities, and thereby attempt to apply new (sociological and computational) ideas to such community structures.

2.12.3 Beliefs

A belief is a truth structure that does not have a formal proof or an absolute truth value. Associated with every belief is a certainty metric that defines the strength of the belief. Certainty metrics vary greatly across domains, but a few axioms provide enough structure so that (a) there are still numerous useful models and, (b) reasoning under the uncertainty inherent in belief systems is still possible.

For the following discussion, a belief system is a context containing beliefs. A semantic context is a context which contains truth structures.

2.12.3.1 Consistency

A system of beliefs is a collection of belief statements. A belief system need not be consistent to have utility, as witnessed by the paraconsistent logics referenced earlier. While correlation

between the structures of the statements is possible, (e.g., two statements about predicates that are partially equivalent), their value and certainties can be radically different.

The only situation that is problematic within kind theory is identical to that which has already been discussed for claims: two beliefs about fully equivalent predicates having different truth values or measures. Again, as in the case with claims, a check for consistency is made when each belief is added to a semantic context.

While a human being can have two minds about something, kind theory only permits agents one position for each judgment in the system. A belief can change from context to context through context change operators. Such a waffling system is not very useful because of its transient truth nature makes finding stable states difficult, but such mutable truth structures are not disallowed.

2.12.3.2 Higher-order Truth Statements

Truth statements made about the truth statements of other agents are called higher-order truth statements. This transition from internal to external truth is necessarily a weakening semantic action. As Locke puts it:

> *What perception is*, everyone will know better than reflecting on what he does himself, when he sees, hears, feels, etc., or thinks, than by any discourse of mine. Whoever reflects on what passes in his own mind cannot miss it. And if he does not reflect, all the words in the world cannot make him have any notion of it.—from *Essay Concerning Human Understanding*, Book II, Chapter IX, Section 2. [247] Emphasis in original.

Higher-order truth statements are sentences of the form

$$\Gamma \vDash \beta_A(P(\psi_{A'}(|E'))|E) = D \text{ where } A \neq A'$$

where ψ is a truth structure of some kind. Agent A is stating something (P) about the beliefs or claims (ψ) of agent A'.

Each time a new truth statement of A' is included in the current truth context Γ, a truth reduction takes place to rewrite this general, higher-order truth statement to a first-order form. Such reductions are not defined here, as they are outside the direct purvey of kind theory. All that is necessitated is that such reduction is order-preserving, as discussed next.

2.12.3.3 Trust Graphs

A trust graph is the dependency relationship that exists between agents and/or predicates within higher-order truth structures. Sentences of the form "agent *A* believes the beliefs of agent *B*" impose trust graphs on semantic contexts.

One axiom for trust graphs is introduced.

Axiom 1 (Higher-order Truth Structures Maintain Metric Structure) *Higher-order statements about beliefs are*

1. ***functional*** *- their existence and evaluation do not modify the structure of the object belief,*
2. ***metric preserving*** *- in reduced, first-order form, all structure is preserved. In particular, the partial order on their metric structure is preserved. A corollary to this point is that reduction is an interpretation preserving such order.*

No structure (particularly, multiplicative) is imposed on truth metrics. A multiplicative structure is a reasonable model for a probability theory-based realization, but it is not the only useful model.

Structures based upon a variety of other metrics have been shown as equally viable, especially in community settings. These new metrics include community respect (e.g., Advogato[4]), notions of good citizenship captured in the notion of karma [340], Trust Webs for public key technologies [49, 111, 123], and connectivity metrics for emergent Web communities [100, 333, 334].

2.12.4 Reasoning With Partial Information

Most of the reasoning that humans perform in their day-to-day lives is not deductive. Most judgments are made based upon experience and beliefs, not upon logic and claims. How are beliefs used to perform judgments in kind theory? The question is how to reason with partial information, especially with a semantics for truth structures as loose as stated in this chapter?

This problem is solved through the realization that knowledge, no matter how rigorous or first-hand, is necessarily inductive and prone to error. Hence, the cutoff point between that which is a claim and that which is a belief is completely relative.

The first-order manner in which such relativity is modeled is with a belief-to-claim cutoff [127, 275]. For each truth surety partial-ordered set *Poset* used by beliefs in a truth context Γ, a set of values *CC* (for claim cutoff) are chosen as belief-to-claim cutoffs. These values indicate the minimal values necessary to interpret a belief as a claim during judgment evaluation.

[4] http://www.advogato.org/

A set of values must be chosen because the truth values are in a poset and not necessarily a lattice. For consistency sake, if two values are comparable, the larger of the two is taken for the cutoff.

Example 9 (A Belief-to-Claim Example) *Assume that the current context contains a set of beliefs that reference truth values $V_0, \ldots, V_9$. These truth values are arranged in posets of the following forms:*

$$V_0 < V_1 < V_2 < V_3$$
$$V_4 < V_5 V_6 < V_7 V_8$$
$$V_9$$

Viewed as an ordered graph, the set of truth values has three components. Elements of different components are incomparable. Additionally, the second component does not form a lattice. The truth values V_5 and V_6 are (pairwise) incomparable, as are V_7 and V_8.

If the cutoff set was $CC = \{V_1, V_5\}$, then all beliefs tagged with truth values greater than or equal to these values are deemed claims during judgment evaluation. This set of transitioned truth values is exactly $\{V_1, V_2, V_3, V_5, V_7, V_8\}$.

Residual beliefs, those that are not adopted as claims, are used by judgment resolution algorithms to rank results in a user-centric manner. This ranking is discussed in Section 6.4.5.

2.13 Metalogical and Model Properties

2.13.1 Metalogical Properties

The primary three metalogical properties that are interesting with respect to a system of semantics (like a logic) are consistency, soundness, and completeness.

These metalogical properties are expressed with respect to kind theory formally as follows:

- Consistency of *KT* (a syntactic property):

  For any *wff* F, exactly one of $\Gamma \vdash F$, $\Gamma \vdash \neg F$, or $\Gamma \vdash ?F$ holds.

- Soundness of *KT*:

  For any *wff* F, if $\Gamma \vdash F$ holds, then $\Gamma \models F$ follows.

- Completeness of *KT*?:

For any *wff* F, if $\Gamma \models F$ holds, then $\Gamma \vdash F$ follows.

Let's review the definitions of $\vdash$ and $\models$ in kind theory so that these definitions can be compared to those used for standard metalogical analysis.

Recall that $\Gamma \vdash F$ means that the formula F is well-formed and derivable from the context Γ; it either exists in the context (i.e., $F \in \Gamma$) or can be constructed from the context via a finite number of applications of the rules of *KT* (i.e., it has a finite proof).

The sequent $\Gamma \models_A F$, on the other hand, means that some agent A (perhaps empty) claims that F holds and provides a proof of such; recall rules (Positive Claim*) and (Negative Claim*) and related rules.

The strong form of soundness and completeness [220, 310] are being used because (a) kind theory is defined constructively, and (b) it is assumed that the context necessarily contains the reflective definition of the logic, as discussed in Chapters 4 and 5. If *KT* were not extensible (i.e., it were a closed logic), then the weak forms of these definitions could be used.

Only claim-based truth structures are considered because, as discussed in Section 2.12.3, beliefs need not even be consistent let alone sound or complete. These definitions are not imprecise exactly because of the claim rules mentioned above; they only cover claim-based, and not belief-based, truth structures.

2.13.1.1 Consistency

In standard classical logics, consistency means that for any *wff* F, either $\Gamma \models F$ holds or $\Gamma \models \neg F$ holds, but not both. The definition of consistency here encompasses this one because the definition is extended to cover three-valued logics like that used for kind theory, but the meaning is consistent for two-valued logics.

Theorem 3 *If KT is based on a consistent foundational logic, then KT is consistent.*

Proof. The only extensions to a foundational logic that can impact consistency are the introduction of new rules (reflectively) via the statement of new functional kind with claims.

New rules are guaranteed consistent by the structure-preserving definition of Δ, and claims are guaranteed consistent by the definition of claim introduction covered in Section 2.12.2. Thus, *KT* is consistent. □

2.13.1.2 Soundness

Many mathematicians believe that, for a logic to be considered useful, it must be either suspected or proven sound. Sometimes a logic is used for years before a proof of soundness is found. In such a situation, some degree of informal assurance that the logic is sound must

exist, typically in the form of a proof-by-usefulness or a strong similarity to an existing logic that is known to be sound.

Proofs of soundness fall into two categories. The first is what is called the procedural method. This proof method calls for the examination of each construct of the theory considering how each impacts validity. These proofs are very long and tedious and error prone.

The second method is a proof-by-model. In this alternative, a model is constructed for the logic. Though this model is another logical system, it is one about which some metalogical properties are known. This is important because every metalogical property a model possesses, its theory inherits.

The soundness of kind theory is shown using the latter method.

Proof-by-Model. The proof-by-model approach is less a proof and more of an examination of the usage evidence. This methodology is inspired by the standard by-use procedure used in the examination of algebraic properties of theories in systems like OBJ3 and its siblings, offspring, and heirs.

The standard properties that are interesting with respect to such theories (algebras) are whether they are confluent and Church-Rosser. An algebra that has these two properties has the property (*). The proofs of these two properties are often quite long and tedious. For example, the proof that (one of) the lambda calculi is Church-Rosser is on the order of a dozen pages long [27, 28], and such a theory has fewer rules than most algebras specified in such systems.

Thus, often a new algebra is declared (*) if, after specifying and using the algebra in a tool like OBJ3 or Maude over a period of time, no evidence of inconsistency is found. Additionally, it has been noted that specifying an algebra with a well-understood model that is not (*) is considered a convoluted task, as discussed in several of the OBJ3 papers by Goguen *et al.*

In Chapter 4, *KT* is associated with a type theoretic model T_{Γ} which, in turn, has an algebraic model $A_{\mathcal{T}}$. Using $A_{\mathcal{T}}$ as an executable specification, a working system that (model-theoretically) covers the bulk of *KT* has been constructed in Maude. Extensive unit tests have been written for this algebra. The system has also been used to prove properties of, and about, some related domains (see Chapter 6). Thus, there exists a high degree of confidence that the model is sound, and therefore a high degree of confidence that the corresponding logic is sound as well.

2.13.1.3 Completeness

The completeness of a logic, on the other hand, is not necessary for its widespread adoption. Theories are used without knowledge of completeness; e.g., set theory variants [7], despite Cohen's work [77], and most theories with any nontrivial axiomatic system are incomplete [170].

Other theories were not known to be complete for an extended period of time but were used nonetheless; e.g., first-order predicate calculus, until shown to be complete by Gödel in his doctoral dissertation.

There is a debate in the deviant logic community as to what completeness means for a non-classical logic. Some even go so far as to claim that completeness results are not necessary at all in such logics [178].

The above definition is labeled with a "?" for exactly this reason. Since no consensus exists for the definition of completeness in such logics, no attempt is made here to define or prove any such result.

2.13.2 Model Properties

Models have a number of properties that are interesting as well. Foremost among these are computability and computational complexity. These properties for the model of kind theory are examined in Chapters 4 and 5.

Chapter 3

Theorems of Kind Theory

In this chapter a basic set of theorems are derived. The first set are all pairwise combinations of compositional application of the core functional operators. These theorems clarify kind theory's semantics and emphasize the structure-preserving foundations.

The second set are existence proofs involving basic operators. These theorems address the question of what happens when two operators share domains or codomains.

The third set are some of the theorems that have been proven that involve compositional kind.

3.0.3 Presentation Style

Diagrams are used in the presentation of the theorems as well. This visual notation, inspired by the commutative diagrams of category theory, is appealing because it is clean, intuitive, and communicates a great deal of information in a very small space. Calculational proofs are provided as well for the majority of the theorems. Not every theorem has a proof diagram.

3.1 Basic Functional Composition

First some basic lemmas of functional composition are presented.

Lemma 1 (Computable-Computable Interp) *The composition of a two computable kinds is a computable kind.*

Proof. The composition of computable functions is computable. □

Lemma 2 (Partial-Partial Interp) *The composition of two partial interpretation kinds is a partial interpretation kind.*

Proof. Partial interpretation preserves substructure on the left. Take two partial interpretation kinds $f < U \hookrightarrow V$ and $g < V \hookrightarrow W$. The composition $g \cdot f$ first applies f which preserves some

substructure of A, then g preserves some substructure of its result. Substructure preservation is transitive. The composition $g \cdot f$ preserves what part of the original substructure of U is preserved by f and g, therefore is itself a partial interpretation. □

If the substructures preserved by composed partial interpretations do not coincide, then all structure is forgotten by their composition.

Lemma 3 (Full-Full Interp) *The composition of a two full interpretation kinds is a full interpretation kind.*

Proof. Full interpretation preserves all structure; two full interpretations composed preserve all structure as well—apply one then the other. □

Lemma 4 (Computable Interp) *The composition of a computable kind $\dashrightarrow$ and a partial interpretation $\hookrightarrow$ (in any order) is a computable kind.*

Proof. Interpretations, partial or full, by their very definition, are computable kind. The composition of two computable kinds is computable by Lemma 1. □

Lemma 5 (Full-Partial Interp) *The composition of a full interpretation and a partial interpretation (in any order) is a partial interpretation.*

Proof. Full interpretation is a subkind of partial interpretation, thus this Lemma follows from Lemma 2. □

Lemma 6 (Canon Canon)

$$\frac{\Gamma \vdash [U] = V \quad \Gamma \vdash [V] = W}{\Gamma \vdash V = W}$$

Proof. The premises are that $[U] = V$ and $[V] = W$. But by the very definition of the functional kind *canonical*, the canonical form of a canonical form is itself. Since $[U] = V$, V is a canonical form, and necessarily $V = W$. □

3.2 Operator Composition

Six core operators have (partially) matching domains and codomains: $\hookrightarrow$, $\rightsquigarrow$, $[\,]$, $<_p$, $\subset_p$, and ":". The semantics for instance operator composition are identical to kind operator composition, so each of these theorems need only be considered once.

Since the realization operator ":" does not compose with itself, nor composes with $<_p$ on the right, the total number of pairwise combinations is 34 ($6 * 6 - 2$). The duality of the proofs nearly halves the number of theorems that must be considered. Thus there are, at most, twenty

basic theorems to review (5 diagonals + $\Sigma_{i=1}^{5}$ pairwise sets = 5 + 15 = 20). Duals will not be mentioned since they are trivial corollaries to other results.

Operator composition in the presence of kind composition is covered in Section 3.4. Here non-compositional operators are studied.

These theorems are presented in a very specific order that is imposed by the structure of kind theory. Each subkind of $\dashrightarrow$ is examined, walking down the inheritance hierarchy, from $\hookrightarrow$ to $\rightsquigarrow$ to $[\,]$. The connection between $<_p$ and $\hookrightarrow$ means that its theorems are detailed next, followed by $\subset_p$. The function kind ":" is last because it has the odd kind (INSTANCE $\rightarrow$ KIND).

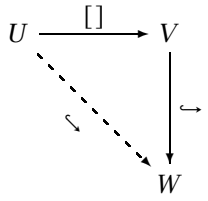
Due to the structure-preserving nature of inheritance, every theorem of $\rightarrow$ is true of all functional kinds since validity is maintained while moving down the inheritance hierarchy. The reason that more specialized theorems are examined is because the new child kind often have additional structure that leads to a stronger proof result.

3.2.1 Canonical-based Theorems

Theorem 4 (PartInterp Canon)

$$\frac{\Gamma \vdash [U] = V \quad \Gamma \vdash V \hookrightarrow W}{\Gamma \vdash U \hookrightarrow W}$$

Figure 3.1: The Theorem Diagram for (PartInterp Canon)



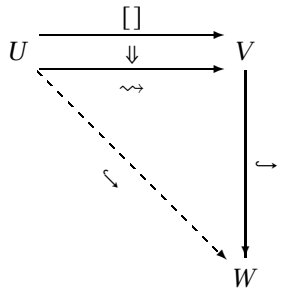
Proof.

$[U] = V$	(premise)
$V \hookrightarrow W$	(premise)
$U \rightsquigarrow V$	(by definition of $[\,]$)
$U \hookrightarrow W$	(Lemma 5)

□

Lets walk through this proof diagram in more detail so that the notational conventions used are clear. A solid arrow denotes a functional kind. The origin of the arrow indicates the related functional kind's domain, the terminal of the arrow points to its codomain. The arrow $\Downarrow$ when

Figure 3.2: The Proof Diagram for (PartInterp Canon)



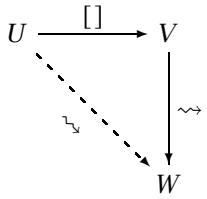
used between two arrows indicates that the origin arrow meta-logically implies the existence of the target arrow; the calculational proof provides the related justification. A non-solid, dashed arrow represents a functional kind whose existence is meta-logically implied by some other structure(s) in the diagram.

So, in proof diagram 3.2, the canonical arrow implies the full interpretation arrow because the canonical functional kind is a subkind of the full interpretation functional kind. The diagonal dashed arrow labeled with a partial interpretation exists because the functional composition of a full interpretation (the aforementioned implied left-to-right arrow) and partial interpretation (the right-hand, top-to-bottom arrow) is a partial interpretation.

Theorem 5 (FullInterp Canon)

$$\frac{\Gamma \vdash [U] = V \quad \Gamma \vdash V \rightsquigarrow W}{\Gamma \vdash U \rightsquigarrow W}$$

Figure 3.3: The Theorem Diagram for (FullInterp Canon)

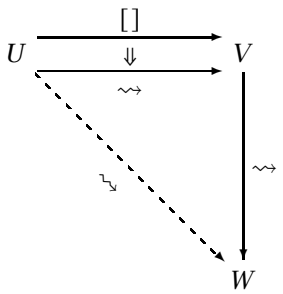


Proof.

$[U] = V$	(premise)
$V \rightsquigarrow W$	(premise)
$U \rightsquigarrow V$	(by definition of $[\,]$)
$U \rightsquigarrow W$	(Lemma 3)

□

Figure 3.4: The Proof Diagram for (Canon FullInterp)

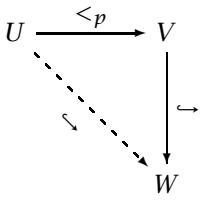


3.2.2 Inheritance-based Theorems

Theorem 6 (PartInterp Parent)

$$\frac{\Gamma \vdash U <_p V \quad \Gamma \vdash V \hookrightarrow W}{\Gamma \vdash U \hookrightarrow W}$$

Figure 3.5: The Theorem Diagram for (PartInterp Parent)

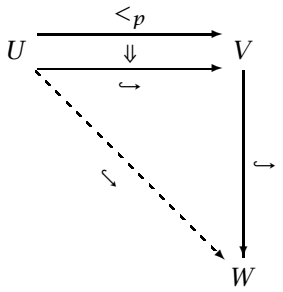


Proof.

$U <_p V$	(premise)
$V \hookrightarrow W$	(premise)
$U \hookrightarrow V$	(Parent Interp)
$U \hookrightarrow W$	(Lemma 2)

□

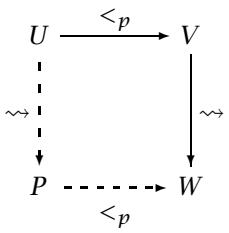
Figure 3.6: The Proof Diagram for (PartInterp Parent)



Theorem 7 (FullInterp Parent)

$$\frac{\Gamma, P \vdash U <_p V \quad \Gamma, P \vdash V \rightsquigarrow W}{\Gamma \vdash U \rightsquigarrow P} \qquad \frac{\Gamma, P \vdash U <_p V \quad \Gamma, P \vdash V \rightsquigarrow W}{\Gamma \vdash P <_p W}$$

Figure 3.7: The Theorem Diagram for (FullInterp Parent)

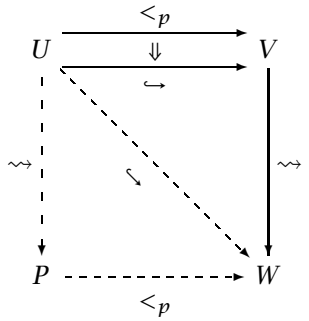


Proof. Since U's parent is V, and $\rightsquigarrow$ fully interprets V to W, then this interpretation also acts upon that substructure of U preserved by $<_p$. Call the object resulting from the full interpretation of that substructure of U that is P. Since full interpretation is structure-preserving, then necessarily P contains the substructure of U that is interpretable by $<_p$: $P <_p W$. □

Theorem 8 (Canon Parent)

$$\frac{\Gamma \vdash U <_p V \quad \Gamma \vdash [V] = W}{\Gamma \vdash U \hookrightarrow W}$$

Figure 3.8: The Proof Diagram for (FullInterp Parent)



Proof.

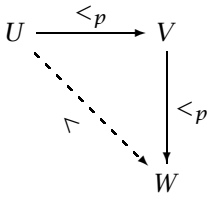
$U <_p V$	(premise)
$[V] = W$	(premise)
$V \rightsquigarrow W$	(by definition of $[\,]$)
$U \hookrightarrow V$	(Parent Interp)
$U \hookrightarrow W$	(Lemma 5)

□

Theorem 9 (Parent Parent)

$$\frac{\Gamma \vdash U <_p V \quad \Gamma \vdash V <_p W}{\Gamma \vdash U < W}$$

Figure 3.9: The Theorem Diagram for (Parent Parent)

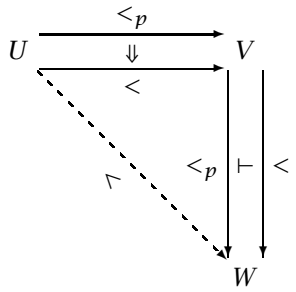


Proof.

$U <_p V$	(premise)
$V <_p W$	(premise)
$U < V$	(Parent Is-a)
$V < W$	(Parent Is-a)
$U < W$	(Is-a Trans)

□

Figure 3.10: The Proof Diagram for (Parent Parent)



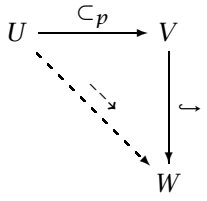
3.2.3 Inclusion-based Theorems

The first inclusion theorem is weak because a general $\hookrightarrow$ operator does not indicate what substructure is preserved.

Theorem 10 (PartInterp Part-of)

$$\frac{\Gamma \vdash U \subset_p V \quad \Gamma \vdash V \hookrightarrow W}{\Gamma \vdash U \dashrightarrow W}$$

Figure 3.11: The Theorem Diagram for (PartInterp Part-of)



Proof.

$U \subset_p V$	(premise)
$V \hookrightarrow W$	(premise)
$U \dashrightarrow V$	(by definition of $\subset_p$)
$V \dashrightarrow W$	(by definition of $\hookrightarrow$)
$U \dashrightarrow W$	(Lemma 1)

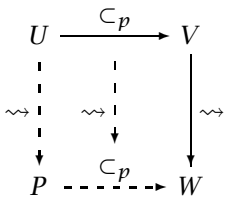
□

Full interpretation, on the other hand, is significantly stronger. Since all structure is preserved, quite a bit can be said about any functional composition with $\rightsquigarrow$.

Theorem 11 (FullInterp Part-of)

$$\frac{\Gamma, P \vdash U \subset_p V \quad \Gamma, P \vdash V \rightsquigarrow W}{\Gamma \vdash U \rightsquigarrow P} \qquad \frac{\Gamma, P \vdash U \subset_p V \quad \Gamma, P \vdash V \rightsquigarrow W}{\Gamma \vdash P \subset_p W}$$

Figure 3.12: The Theorem Diagram for (FullInterp Part-of)



Proof. Since V contains U, and $\rightsquigarrow$ fully interprets V to W, then this interpretation also acts upon U. Call the object resulting from the full interpretation P. Since full interpretation is structure-preserving, and $\subset_p$ is a component of that structure, then necessarily $P \subset_p W$. □

Corollary 12 (Part-of Canon)

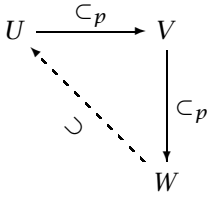
$$\frac{\Gamma, P \vdash U \subset_p V \quad \Gamma, P \vdash [V] = W}{\Gamma \vdash U \rightsquigarrow P} \qquad \frac{\Gamma, P \vdash U \subset_p V \quad \Gamma, P \vdash [V] = W}{\Gamma \vdash P \subset_p W}$$

Proof. This corollary is true because canonicality is a computable kind. □

Theorem 13 (Part-of Part-of)

$$\frac{\Gamma \vdash U \subset_p V \quad \Gamma \vdash V \subset_p W}{\Gamma \vdash W \supset U}$$

Figure 3.13: The Theorem Diagram for (Part-of Part-of)

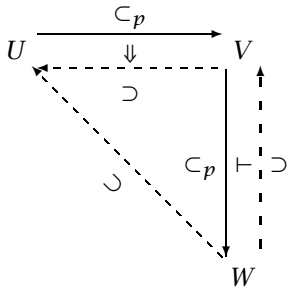


Proof.

$U \subset_p V$	(premise)
$V \subset_p W$	(premise)
$V \supset U$	(Part-of Has-a)
$W \supset V$	(Part-of Has-a)
$W \supset U$	(Has-a Trans)

□

Figure 3.14: The Proof Diagram for (Part-of Part-of)



The composition of inheritance and inclusion has no interesting result when the parent operator follows inclusion (e.g., $U \subset_p V \wedge V <_p W$). On reflection, this is not surprising because, while U is a part of V, it might have been one of the new pieces of structure that distinguishes V from W. Thus no fundamental relation between U and W need necessarily exist.

The same kind of reasoning holds true for the final functional pair, that of $\subset_p \cdot :.$ Just because K is a part of L does not coerce I into any relation with L or any realization of L for that matter. While it is *possible* that I is a part of some P that realizes L (e.g., $I \subset_p P \wedge P : L$), it is not *necessary*.

3.2.4 Realization-based Theorems

Finally, there are several realization-based theorems in this set.

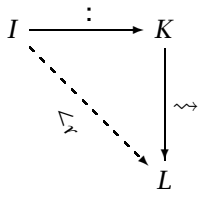
The first realization-related operation composition is $\hookrightarrow \cdot :.$ As discussed in Section 2.9.11, this particular composition is not identifiable with the current logic. Therefore no theorem exists for this composition.

The second realization composition is with full interpretation.

Theorem 14 (FullInterp Real)

$$\frac{\Gamma \vdash I : K \quad \Gamma \vdash K \rightsquigarrow L}{\Gamma \vdash I <_r L}$$

Figure 3.15: The Theorem Diagram for (FullInterp Real)



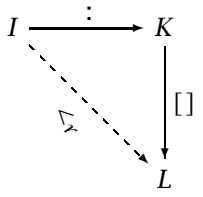
Proof. Since $\rightsquigarrow$ fully preserves structure, then all structure in K upon which the ":" operation depends is fully available, through interpretation, in L. This means that $I <_r L$ because K acts as the middle-man kind in the relational definition of $<_r$. □

Next, canonical forms of realizations are examined.

Corollary 15 (Canon Real)

$$\frac{\Gamma \vdash I : K \quad \Gamma \vdash [K] = L}{\Gamma \vdash I <_r L}$$

Figure 3.16: The Theorem Diagram for (Canon Real)



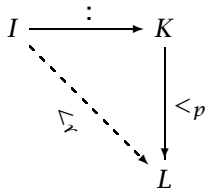
Proof. This corollary is true given canonicality is a computable kind. □

Finally, the relationship between realizations and inheritance is discussed.

Theorem 16 (Parent Real)

$$\frac{\Gamma \vdash I : K \quad \Gamma \vdash K <_p L}{\Gamma \vdash I <_r L}$$

Figure 3.17: The Theorem Diagram for (Parent Real)

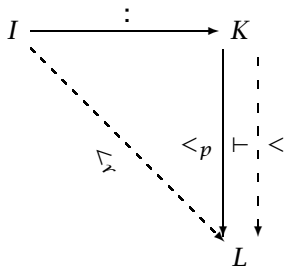


Proof.

$I : K$	(premise)
$K <_p L$	(premise)
$K < L$	(Parent Is-a)
$I <_r L$	(by definition of $<_r$)

□

Figure 3.18: The Proof Diagram for (Parent Real)



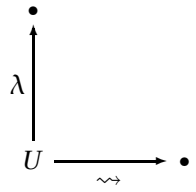
3.3 Existence Proofs

Now all direct functional compositions of basic operators have been considered, but what about situations where operators share domains or codomains? A few of these theorems are examined next.

3.3.1 Under Full Interpretation

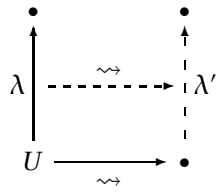
The majority of cases to consider are those situations involving full interpretations and canonical forms (which are full interpretations).

Figure 3.19: General Case of Full Interpretation: Premise



In all situations where there is a diagram of the form seen in Figure 3.19 where λ is some functional kind with domain U, full interpretation is guaranteed to preserve all structure. Thus the resulting contexts are always of the form seen in Figure 3.20. The full interpretation has

Figure 3.20: General Case of Full Interpretation: Result



a behavior and a data domain, as discussed in Section 2.7.4. The data domain is exactly the elements of U, and the behavior domain is exactly the functions on U represented here by λ.

3.3.2 Under Partial Interpretation

Likewise, partial interpretations share this diagrammatic representation with full interpretation, but in a forgetful sense. If λ is not part of the substructure preserved by a partial interpretation $\hookrightarrow$, then there is no horizontal arrow, as in the above diagram, mapping that λ to some new λ'. Likewise, if a given object in U is not part of the data domain of $\hookrightarrow$ it is not mapped in a structure preserving fashion.

As discussed in Definition 6, a partial interpretation $\hookrightarrow$ is a completed partial function. Those elements and maps (data and behavior) that are not part of the substructure on which $\hookrightarrow$ is defined are mapped to the null terminal object ω and are discarded and forgotten.

3.3.3 Under Canonicalization

Recall from Definition 13 the distinction between the canonicalization and the notion of full equivalence. Canonicalization guarantees not only that full structure on the left is preserved, but that the interpretation is invertible. This means that no new structure can exist in the canonical form that is identified within the current context.

Figure 3.21: General Case of Canonicalization

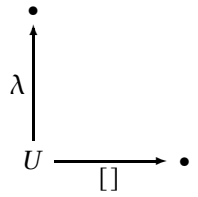
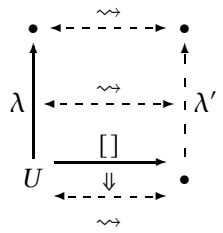


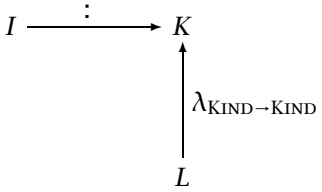
Figure 3.22: General Case of Canonicalization: Result



This means that any diagram of the form as seen in Figure 3.21 results in a structure of the form witnessed in Figure 3.22.

3.3.4 With Realization

Figure 3.23: Realization Structures



Consider compositional forms involving a realized kind. This entails structures of the form diagrammed in Figure 3.23.

As seen, for example, in Theorem 10, since what (sub)structure is preserved by interpretation is not known *a priori*, nothing can be deduced about those situations where $\lambda \in \{\rightsquigarrow, \hookrightarrow\}$.

On the other hand, when $\lambda = [\,]$, it can be guaranteed that, since the canonical kind K represents the equivalence class $[K]_{\equiv}$ for some equivalence $\equiv$, then instances of K (I in the diagram) are representatives of the equivalence class of realizations. That is to say, they are each a class representative under instance canonicalization. Thus the partial diagram completes in the following manner.

Figure 3.24: Completed Realization Structures

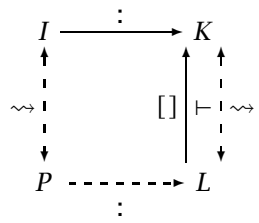
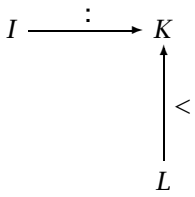


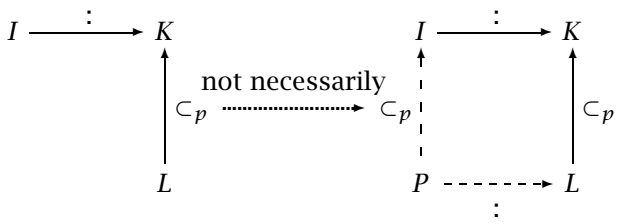
Figure 3.25: Realization with Inheritance



Inheritance poses the trickiest puzzle for this context structure. A diagram of the form seen in Figure 3.25 implies that a partial interpretation exists from K to L by (Parent PartInterp). If this partial interpretation covers the codomain of realization, then $I <_r L$. In this case inheritance is being used only as a structuring notion on equivalent kinds and perhaps those kinds have not yet been realized as an equivalence by an agent. In general, what substructure the interpretation is preserving is not known, thus nothing can be said about a relationship between the instance I and the kind L.

Inclusion has a much simpler result. Given a context of the form witnessed in Figure 3.26, it is clear that each realization of L (call it P) is contained in a realization of K (perhaps I).

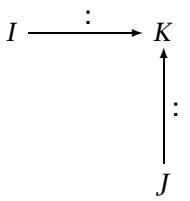
Figure 3.26: Realization with Inclusion



Of course, it is not *necessary* that I contains some realization of L because, for example, no realization need exist!

Finally, what about those common situations where two realizations share a kind? The

Figure 3.27: Shared Realization



$=$ Boolean function which compares the syntactic structure of two assets has already been defined, so I and J can be compared with $=$. Likewise, given the [] operation that must be defined on K, if $I = J$ can be trivially checked. If this is true, then the instances are (a) equivalent, (b) in the same equivalence class under their parent kind, and (c) one of them can be safely removed from the context. This is an example of a common context judgment that might be captured by a "kind wizard", as is briefly discussed in Chapter 6.

3.4 Compositional Theorems

Next a few theorems about compositional structures are proven.

Because the raw definitions of asset composition is being used, theorems are necessarily being examined in their *weakest* form. That is to say, if a new composition operator is defined via inheritance, extra semantics are added in the standard way for a functional kind, strengthening the operator's semantics as compared to its parent. This leads to a standard trade-off: the resulting increase in the operator's expressiveness, and thus strengthening of its theorems,

correspondingly decreases its applicability because of the restriction in its models.

In any case, because these are very weak theorems, they are not as expressive as one might hope. Examples are highlighted of such as different compositions are covered. When stronger, more expressive operators are defined that are closer to the problem domains, the corresponding theorems are strengthened to match the expectations for those domains.

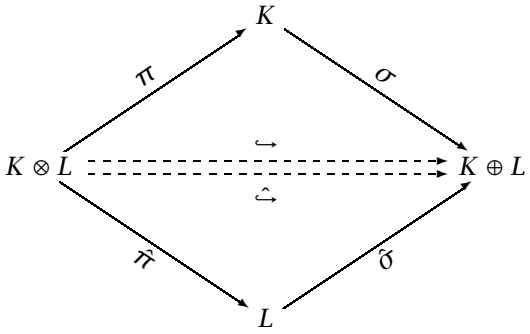
First, consider composition without any extra structure at all. For example, consider the case of taking a product kind, decomposing it, then reassembling its components in a new form. What properties does that new form have?

Theorem 17 (Refactoring)

$$\frac{\Gamma \vdash K \otimes L \quad \Gamma \vdash K \oplus L}{\Gamma \vdash (K \otimes L) \hookrightarrow (K \oplus L)}$$

Proof. Projection maps are partial interpretations by definition. Join maps are full interpretations by definition. Thus, the maps $\pi : K \otimes L \to K$ and $\hat{\pi} : K \otimes L \to L$ are both interpretable (total, computable) functions. The composition of a partial interpretation and a full interpretation is a partial interpretation by Lemma 5. Thus the result follows by two applications of CFK. □

Figure 3.28: Refactoring Proof Diagram



This theorem says that when you decompose some asset and put the parts back together, you are guaranteed to be able to transmute the old into the new, but possibly with some loss of structure and potentially weakened semantics. This kind of transformation is more common than one might think; examples include rewriting a paper and refactoring software.

Next, consider inheritance under composition. The first interest is in understanding how basic compositional forms impact inheritance relationships.

Since join maps are full interpretations, the use of coproducts with inheritance is nontrivial.

Theorem 18 (Coproducts under Inheritance)

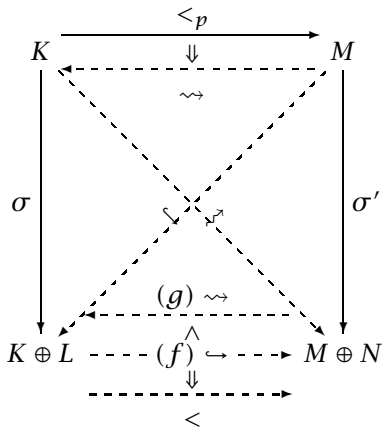
$$\frac{\Gamma \vdash K \oplus L \quad \Gamma \vdash M \oplus N \quad \Gamma \vdash K <_p M \quad \Gamma \vdash L <_p N}{\Gamma \vdash K \oplus L < M \oplus N}$$

Proof. Without loss of generality, since the diagram and decomposition are symmetric, only the top half of the diagram is discussed. Because $K <_p M$, by (Parent Interp) a full interpretation $\rightsquigarrow$ exists between M and K (i.e., $M \rightsquigarrow K$) and a partial interpretation exists from K to M ($K \hookrightarrow M$, which is exactly the parent relation in the first place). Due to Lemma 3, a full interpretation exists between M and $K \oplus L$. Likewise, due to Lemma 5, a partial interpretation exists between K and $M \oplus N$. Therefore, by the definition of coproduct, two unique maps exist: $f : K \oplus L \to M \oplus N$ and $g : M \oplus N \to K \oplus L$.

Now that it is known that these two unique maps exist, what properties do they have? The map f must be an interpretation that, when composed with the full interpretation σ, results in a partial interpretation. Thus f must be partial. In a similar vein, g must be full. Since there exist dual full/partial interpretations between two kinds, then by (Parent Interp*) $K \oplus L <_p M \oplus N$.

□

Figure 3.29: Coproducts under Inheritance Proof Diagram



What about inheritance with products? Projections are partial interpretations, this impacts how much can be said in the theorem.

Theorem 19 (Products under Inheritance)

$$\frac{\Gamma \vdash K \otimes L \quad \Gamma \vdash K <_p M \quad \Gamma \vdash L <_p N \quad \Gamma \vdash M \otimes N}{\Gamma \vdash K \otimes L \hookrightarrow M \otimes N}$$

This theorem is proved along the same lines as Theorem 18, so it need not be covered it in

detail here. What is striking to notice is that products weaken what can be said significantly. It can be stated that product assets whose components are related by inheritance have *some* preserved substructure, but what that substructure is cannot be divined via this logic.

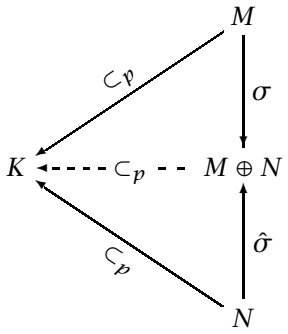
Consider containment under composition. If an asset has two parts, does it also have some composition of those parts?

Theorem 20 (Containment under Composition)

$$\frac{\Gamma \vdash M \subset_p K \quad \Gamma \vdash N \subset_p K \quad \Gamma \vdash M \oplus N}{\Gamma \vdash M \oplus N \subset_p K}$$

Proof. Given the assumptions, by the definition of coproduct, a unique map $f : M \oplus N \to K$ exists. Because joins are full interpretations, then they preserve all structure of their left-hand sides. Part of the structure of the joins in this decomposition is exactly the part-of maps to K. Thus, $M \oplus N \subset_p K$ as well. □

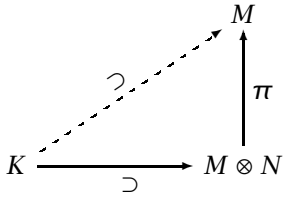
Figure 3.30: Containment under Composition



It is known that the dual of this theorem is true as well.

Theorem 21 (Containment Decomposes)

$$\frac{\Gamma \vdash K \supset M \otimes N}{\Gamma \vdash K \supset M}$$



This theorem says that if a compositional asset has a decomposable part, then it also contains the decomposition of that part.

And that concludes the look at compositional theorems.

Chapter 4

An Operational Model

While kind theory is specified as a formal system, it does not proscribe an operational model. In order to compute with kind theory, using it within software engineering tools and technologies, an implementation of the theory is necessary. One such model is defined in this chapter.

With regards to implementing a formal system (i.e., building a model of the logic), there are two options:

1. realize the logic within a logical framework, layering the formalism on a specific foundational logic, and computing within that framework, or
2. directly implement the logic within a programming language, representing core constructs with data structures, execute behaviors with directly realized computation, etc.

The first option was chosen for this work, leaving the second option for future work, as discussed in Chapter 8.

The second option, if executed well, will result in a very high-performance, highly-tuned implementation of kind theory. But realizing such an implementation should only happen after the metalogical aspects of the theory are fully comprehended, and how to layer it on other formalisms is deeply understood. These things can only happen within a real logical framework, as in the first option.

4.1 The Operational Process

The assets that are to be reasoned about are varied: program code, documents, web resources, etc. The first question to address is how to move from having these instances to reasoning about them?

First, some way to read instances into a computational system so that they can be formally represented is necessary. Some tool needs to parse instances into a kind theoretic representation, taking care to track the relationships between the literal assets and their formal de-

scription. A part of this parsing process is the use of explicit and implicit information within the source material for automatic classification (kinding) (see Sections 7.2.4). When an asset is kinded, it is automatically represented in canonical form (recalling Definition 13).

Representations are needed not only for instances, but also for kinds. Put another way, a "seralization" mechanism is needed for storing and retrieving collections of kind (domains, contexts, individual kind, etc.). XML, with its focus on the syntactic representation and manipulation of structured information, is the appropriate choices for such a mechanism.

Motivated by the core aspects of kind theory, performing structural reasoning on kinds is of core importance. Additionally, an efficient operational semantics for this reasoning is desirable. The operations of primary interest are exactly those primary operations of kind theory: decisions about equivalence and equality, inheritance, and inclusion.

The remaining fundamental operation of kind theory, interpretation, is realized by the underlying operational semantics of this formalism. In particular, (many) order sorted algebras with membership equational and rewriting logics are used. Thus, via the use of terminated and unterminated rewriting, there is sufficient computational capability to cover all aspects of interpretation.

This chapter describes a model that provides for the representation of both kinds and instances. This model is designed for efficiency and simplicity. Its representation is a type system with an algebraic model and operational semantics. This type system is called T_Γ.

In effect, T_Γ provides a minimalist representation of archetypal structures (the types which are the canonical representations of kinds in this model) and instances of those structures (realizations of the instances of kind theory). Its structure is based directly on the structure of kind theory; it is a model for kind theory by construction.

4.1.1 Model Incompleteness

The primary aspect of kind theory that explicitly not covered in this chapter and model is that of truth structures.

Each of the five substructures that make up truth structures must be accounted for: representing an agent, context, predicate, truth value, evidence, and degree.

On the surface, handling most of these is trivial, as each is represented by an instance in T_Γ of the appropriate type. Types for truth structures (`Tri` and `Boolean`) are explicitly defined in this chapter. Chapter 5 describes a reflective model that provides kinds (and thus, types) for the remaining substructures.

4.1.2 Context Representation

An aspect to consider with respect to encoding these truth substructures is the interplay between context representation and space complexity.

When an agent adds a truth structure to a context, the context relevant to that claim or belief is part of its structure. That context is often either the current context of the system or perhaps a subcontext of a previously constructed context, often a domain.

The context can be "snapshotted," added as a new instance to the current context, and that instance name is used as the reference within the truth structure. This ensures correct semantics, but is horrid from a space-complexity point of view.

Various database systems and file-systems provide a similar "snapshot" capability at much lower cost, so one of these technologies is a wise choice as a context-holding substrate. Such a mechanism will likely be used in the future when implementing kind theory natively.

This problem is explicitly tackled in the reflective model by using the notion of versioning within the core kind UNIVERSAL. Changes to a kind or instance imply changes to the version number, and all such changes are represented as partial interpretations. Essentially, this can be thought of as a revision control system for assets. In fact, this is how asset refinements are realized in the repository, as described in Chapter 6.

Using such a differential notion means that a space efficient representation of complex compositional constructs like a context comes at the cost of referential computation complexity. Referring to an "old" instance takes more than O(1) time because the instance has to be constructed on the fly from differentials. These aspects have not been studied in detail, nor is this solution considered in later when analyzing the properties of T_Γ and $A_\mathcal{T}$.

4.1.3 Chapter Outline

This chapter has four main parts. First, *KT* is described in full detail in the next three sections. Then the mapping between *KT* and T_Γ is discussed in Section 4.5. The algebra $A_\mathcal{T}$ that is a model for T_Γ is next discussed in Section 4.6. Finally, some analysis of these models is discussed in the final section of the chapter.

4.2 Constructs

T_Γ needs a number of basic constructs to represent a core set of ground kinds (recall Section 2.4.2 and the rule (Ground*)). These concepts are inspired primarily by research in knowledge representation [222], formal concept analysis and conceptual graph techniques [379], and ontology construction [362].

The basic constructs used are the following:

Ground – Ground concepts are entities that have no meaning other than "self." They are the fundamental representation of a concept, idea, or notion. There is no transliteration between grounds within T_Γ; that is to say, every ground is of exactly one, and only one type.

A small set of core ground concept types are defined within T_Γ. They include *integers*, *floating point numbers*, *Boolean and ternary logic values*, and *strings*.

New grounds can be added to T_Γ. Of course, new grounds are only valid if they do not conflict with existing grounds—they must introduce no *confusion*, in an algebraic sense, and they must preserve the *soundness* of the logic.

The use of the term *ground* is inspired by philosophical theories on ground and form (Platonic vs. Aristotelian points of view), some of which is inspired by readings such as Hofstadter [188].

Property – A *property* is a ground term that is associated with a *value*.

The term *property* was chosen over other alternatives like *attribute* or *slot* because (a) knowledge representation and library science researchers use this term, and (b) the alternatives have overloaded meaning, especially when applied to object-oriented systems or learning systems.

Value – A *value* is a list of ground terms that is associated with a property.

Slot – A *slot* is a declaration of a typed variable. The first part of a slot is a type name and the second part is a property.

The term *slot* is inspired by the notion of slots in frame-based systems in artificial intelligence [358].

Value Slot – A *value slot* is a slot-value pair. It is a typed variable and its current value.

Type – A *type* is a named collection of slots. A type can be thought of as a template or schema used to describe a new construct. Types have parent types, so there is a notion of inheritance. Types look like named records.

Instance – Finally, an *instance* is a specific instantiation of a type.

Now that the basic nomenclature of T_Γ has been reviewed, the specification of T_Γ is presented.

4.3 Specification Overview

T_Γ is defined in three stages:

1. A logic of terms and contexts is introduced called T_Γ. This logic is a constructive type system [327]. It happens to be a variant on the type system Cardelli discusses in [48].
2. It is shown that T_Γ, as an axiomatic proof system, is decidable.
3. The decidability of the proof system motivates the definition of an algebra $A_\mathcal{T}$ that is a model for T_Γ using many order-sorted algebra.

Chapter 5 defines the interpretations necessary to realize the semantics of kind theory using T_Γ. The notions that drive the definition of T_Γ are entirely due to the structure of kind theory as covered in Chapter 2.

4.3.1 Grounds

A *ground* is an entity that has exactly one of the following ground data types. This particular list is inspired by many sources including, but not restricted to, Common Lisp [346], Scheme, ADT theory, database theory, and knowledge theory [363].

The basic set of ground types are

- Boolean – The Boolean ground type.
- Integer – The integer ground type.
- Float – The float ground type.
- String – The string ground type.
- Tri – The ternary ground type.
- Universal – The universal ground type.

Name/reference types are not ground types as they are not fundamentally necessary due to the by-name equivalence of T_Γ. If explicitly represented by types, name/reference types would be represented as either subtypes of String or record types of some sort (e.g., encoding URLs with fields for protocol, realm, and name).

Within the logic, integers and floats have infinite precision and strings are of arbitrary length. Within the realization of the model all instances have finite length since there are finite resources.

Call the set of grounds *Basic*:

$$Basic = \{\mathsf{Bool}, \mathsf{Int}, \mathsf{Float}, \mathsf{String}, \mathsf{Tri}, \mathsf{Universal}\}$$

No operators are necessarily defined on ground elements in T_Γ. For example, when the length of a ground string is mentioned, this is only meaningful within the meta-language used to specify T_Γ.

4.3.2 Specification Constructs

Several supplementary constructs are used to define the constructs *type* and *instance*. These "helper constructs" are for specification purposes only; they cannot be used outside of the context of a type or instance definition or a judgment in T_Γ.

The supplementary specification constructs are *properties* that are ground strings, *values* that are lists of ground strings, *slots* that are ground-property pairs, and *value slots* that are slot-value pairs. Given the structure of these supplementary constructs, basic data structures like *pairs*, *triples*, and *lists* are needed.

4.3.3 Types

In the terminology of Goguen and Wolfram [167], types are the Platonic "ideal" of a construct: the permanent and unchanging notion of what it is to be a specific "kind of thing" from a property point-of-view. A type is a triple consisting of a *name*, a *set of parent type names*, and a *set of slots*.

In the language of classic type theory, a $F_{1<:}$-like type system is constructively defined [1, 48]. The type system is open—a user can define new named record types within the type system.

Types have by-name equivalence rather than structural equivalence. Two distinct constructs described in a constructive knowledge system like kind theory will, on occasion, coincidentally be structurally (or even syntactically) equivalent. However, their identities must remain distinct in all contexts but for those that explicitly declare equivalence. These implications force by-name semantics for equivalence.

4.3.4 Instances

Instances, on the other hand, are the Aristotelian notion of a "thing" or "entity." An instance is a triple consisting of a *unique name*, a *type*, and a *set of value slots*.

In the language of classic type theory, instances are named variable declarations of record types with by-name equivalence.

4.4 Formal Specification

This section contains the detailed formal specification of T_Γ. T_Γ's logic, called T_Γ, is defined via a type system. T_Γ is shown to be sound and complete, then the use of T_Γ as a proof system is shown to be decidable. The decidability of the proof system motivates the definition of an algebra $A_\mathcal{T}$ that is a model for T_Γ.

First, a set of logic rules that define T_Γ are introduced. These rules are listed in Tables 4.2 through 4.12.

An *environment* Γ is composed of a type context Γ_T and an instance context Γ_I. Sometimes this is written as $\Gamma \triangleq \Gamma_T \triangleright \Gamma_I$. The symbol $\varnothing$ is used to denote an empty context.

4.4.1 Syntax of the Type System

The syntax of T_Γ is summarized in Table 4.1. Curly braces are used to denote sets and tildes are used to denote lists. Ellipses are used to denote expanded sets or lists.

Table 4.1: Syntax of T_Γ

$A, B ::=$	types
$K \quad K \in Basic$	basic types
$M, N ::=$	terms
x	variable
$\{S\}$	slot set
$\{V\}$	valueslot set
$\{T\}$	type set
$\{T_0 \ldots T_n\}$	type set
(S, P)	slot
$\{(S, P)\}$	slot set
$\{(S_0, P_0) \ldots (S_n, P_n)\}$	expanded slot set
$(S, P = \widetilde{V})$	valueslot
$\{(S, P = \widetilde{V})\}$	valueslot set
$(S, P = V_0, \ldots, V_n)$	expanded valueslot
$\{(S_0, P_0 = V_{s(0)}, \ldots, V_{e(0)}) \ldots (S_n, P_n = V_{s(n)}, \ldots, V_{e(n)})\}$	expanded valueslot set
$\langle N, \{T\}, \{S\}\rangle$	type definition
$[I, T, \{V\}]$	instance definition
Γ, T, Γ'	context union

4.4.2 Basic Judgments on the Algebra

The basic judgments for contexts and environments are summarized in Table 4.2. Type contexts are sufficient environments to perform judgments exclusively on types. Judgments concerning instances necessitate the use of an instance context (e.g., $\Gamma_T \triangleright \Gamma_I$).

Table 4.2: Basic judgments on T_Γ

$\Gamma_T \vdash \diamond$	Γ_T is a well-formed type context.
$\Gamma_T \rhd \Gamma_I \vdash \diamond$	$\Gamma_T \rhd \Gamma_I$ is a well-formed environment.
$\Gamma_T \vdash N$	N is a well-formed type in Γ_T.
$\Gamma_T \vdash \langle N, \{T\}, \{S\}\rangle$	$\langle N, \{T\}, \{S\}\rangle$ is a well-formed type in Γ_T.
$\Gamma_I \vdash I$	I is an instance in Γ_I.
$\Gamma_I \vdash [I, T, \{V\}]$	$[I, T, \{V\}]$ is an instance in Γ_I.
$\Gamma_T \rhd \Gamma_I \vdash I$	I is a well-formed instance in $\Gamma_T \rhd \Gamma_I$.
$\Gamma_T \rhd \Gamma_I \vdash [I, T, \{V\}]$	$[I, T, \{V\}]$ is a well-formed instance in $\Gamma_T \rhd \Gamma_I$.
$\Gamma_T \rhd \Gamma_I \vdash I : T$	I is a well-formed instance of the well-formed type T in $\Gamma_T \rhd \Gamma_I$.
$\Gamma_T \vdash T <: U$	T and U are well-formed types in Γ_T and T is a subtype of U.
$\Gamma_T \rhd \Gamma_I \vdash T <: U$	T and U are well-formed types in $\Gamma_T \rhd \Gamma_I$ and T is a subtype of U.
$\Gamma_T \vdash N \subset M$	N and M are well-formed types in Γ_T and N is *part-of* M.
$\Gamma_T \rhd \Gamma_I \vdash I \subset J$	I and J are well-formed instances in $\Gamma_T \rhd \Gamma_I$ and I is *part-of* J.

4.4.2.1 Basic Rules

Table 4.3: Basic rules for T_Γ

Let the symbol T represent any basic type ($T \in \mathit{Basic}$).

$$\text{(Base Env)}\ \frac{}{\varnothing \vdash \diamond} \qquad \text{(Empty Inst Cont)}\ \frac{\Gamma_T \vdash \diamond}{\Gamma_T \rhd \varnothing \vdash \diamond} \qquad \text{(Basic Types)}\ \frac{\Gamma_T \vdash \diamond}{\Gamma_T \vdash T}$$

The basic rules for contexts are found in Table 4.3. Rule (Base Env) is the only rule that does not require any assumptions, thus it is the only axiom. It states that the empty environment is valid. Rule (Empty Inst Cont) states that the empty instance environment is well-formed as well. Finally, the rule (Basic Type) states that all well-formed type environments contain all the basic types. Thus, the minimal type environment is exactly *Basic*.

4.4.2.2 Subtyping

Table 4.4: Judgments for subtyping

$\Gamma_T \rhd \Gamma_I \vdash M : T$	M is a well-formed term of type T in $\Gamma_T \rhd \Gamma_I$.
$\Gamma_T \vdash T <: U$	T and U are well-formed types in Γ_T and T is a subtype of U.
$\Gamma_T \rhd \Gamma_I \vdash T <: U$	T and U are well-formed types in $\Gamma_T \rhd \Gamma_I$ and T is a subtype of U.

Next, judgments for subtyping are defined. The basic subtyping judgments for contexts and environments are summarized in Table 4.4.

Table 4.5: Subtyping rules for T_Γ

For (Bool Dep), $K \in \{\mathsf{Int}, \mathsf{Float}, \mathsf{String}, \mathsf{Tri}\}$.

$$\frac{\Gamma_T \vdash A}{\Gamma_T \vdash A <: A}\ \text{(Sub Refl)} \qquad \frac{\Gamma_T \vdash \langle N, \{T_0 \ldots T \ldots T_n\}, S\rangle}{\Gamma_T \vdash N <: T}\ \text{(Sub Def)}$$

$$\frac{\Gamma_T \vdash A <: B \quad \Gamma_T \vdash B <: C}{\Gamma_T \vdash A <: C}\ \text{(Sub Trans)} \qquad \frac{\Gamma_T \rhd \Gamma_I \vdash [I, T, \{V\}]}{\Gamma_T \rhd \Gamma_I \vdash I : T}\ \text{(Inst Type)}$$

$$\frac{\Gamma_T \rhd \Gamma_I \vdash I : A \quad \Gamma_T \vdash A <: B}{\Gamma_T \rhd \Gamma_I \vdash I : B}\ \text{(Inst Subsump)} \qquad \frac{\Gamma_T \vdash A}{\Gamma_T \vdash A <: \mathsf{Universal}}\ \text{(Sub Univ)}$$

$$\frac{\Gamma_T \vdash K}{\Gamma_T \vdash \mathsf{Bool}}\ \text{(Bool Dep)}$$

The subtyping rules are listed in Table 4.5. As one expects, subtyping is reflexive and transitive. Rule (Sub Def) states that types can have arbitrary numbers of unique supertypes and (Inst Type) states that instances have exactly one immediate supertype. All types are a subtype of Universal by rule (Sub Univ), and given rule (Inst Subsump), all instances are also of type Universal. The instance subsumption rule (Inst Subsump) states that if a construct I has a type A, and A is a subtype of B, then I also has type B. Thus subtyping behaves like set inclusion when type membership is seen as set membership.

Finally, rule (Bool Dep) states that the three basic types String, Int, and Float depend upon the existence of the basic type Bool. This rule is redundant given the current definition of *Basic*, but as seen later, when the redefinition of the basic context is permitted, this rule becomes important.

4.4.2.3 Type Inclusion

Table 4.6: Inclusion rules for T_Γ

$$\frac{\Gamma_T \vdash \langle N, \{T\}, \{(S_0, \widetilde{P_0}) \ldots (S, \widetilde{P} \ldots (S_m, \widetilde{P_m})\}\rangle}{\Gamma_T \vdash S \subset N}\ \text{(Inclusion)} \qquad \frac{\Gamma_T \vdash A \subset B \quad \Gamma_T \vdash B \subset C}{\Gamma_T \vdash A \subset C}\ \text{(Inclusion Trans)}$$

Rules for inclusion are listed in Table 4.6. Rule (Inclusion) states that if a type S is used

in a slot of a type definition N, (see rule (Val First Pair) in Table 4.11 and rule (Inst Intro) in Table 4.14), then that type T is a part of N. Additionally, type inclusion is transitive by (Inclusion Trans).

4.4.2.4 Basic Types and Instances

An empty or singleton initial type context containing the Universal type is not very useful. Thus, the initial type context is parameterized by a finite set of **basic types**. The basic types are models for standard structures like arithmetic over integers and floats, manipulation of strings, etc.

Definition 23 (Basic Types) *The set of basic types that have been identified as useful were mentioned in Section 4.3.1. Each type has a set of rules. The rules for the finite basic types,* Universal, Bool, Tri, *and* String *are detailed in Table 4.7 through Table 4.9.*

The rules for the infinite basic types Int *and* Float *are detailed in Table 4.10. Standard basic operations are defined on integers and floating point numbers, instances of which are represented in a canonical finite form.*

The supplementary type List is also necessary. The operations defined on List are construction and membership test.

Additionally, as mentioned earlier in rule (Bool Dep), the basic types Int, Float, String, and Tri depend upon the existence of the basic type Bool.

For each basic type introduced to Γ_T, a set of basic instances are defined. For some basic types like Bool, Tri, List and Set, the set of basic instances is finite.

Definition 24 (Instances of Basic Types) *The set of basic instances for* Bool *consists of* true *and* false*, as detailed in rules (Val True) and (Val False).*

For the basic type Tri*, the instances of* Bool *are used, plus the new instance* unknown*, as detailed in rule (Val Unknown).*

For String *and* List*, the set is the singletons* blank *and* nil *respectively, as captured in rules (Val Blank) and (Val Nil).*

Finally, for the basic types Int *and* Float*, the set of basic instances are countably infinite. The addition of* Int *introduces instances of the form* $[value, \mathsf{Int}, \varnothing]$*, and the addition of* Float *introduces instances of the form* $[value, \mathsf{Float}, \varnothing]$.

The symbol $\varnothing$ *is used as a valueslot placeholder for types with no slots.*

Definition 25 (Supertype Set) *Given* $\Gamma_T \vdash \langle N, \{T\}, \{S\}\rangle$*, the set of all types above* $\langle N, \{T\}, \{S\}\rangle$ *in* Γ_T*'s type set are called the* ***supertypes*** *of* N*. This set is called* $N_{<:}$.

Table 4.7: Rules for basic type Universal

$$\frac{\Gamma_T \vdash \diamond}{\Gamma_T \vdash \mathsf{Universal}} \text{ (Type Univ)}$$

Table 4.8: Rules for basic types Bool and Tri

The symbol BT denotes either the type Bool or the type Tri. All uses of the symbol BT in a rule must be replaced by the same type. Therefore each rule that uses this symbol is shorthand for two rules. $\mathcal{F} \in \{=, \neq, \vee, \wedge, +, \Longrightarrow\}$ is an operator on instances of type Bool or Tri.

$$\frac{\Gamma_T \rhd \Gamma_I \vdash \mathsf{Bool}}{\Gamma_T \rhd \Gamma_I \vdash [\mathit{true}, \mathsf{Bool}, \varnothing] : \mathsf{Bool}} \text{ (Val True)} \qquad \frac{\Gamma_T \rhd \Gamma_I \vdash \mathsf{Bool}}{\Gamma_T \rhd \Gamma_I \vdash [\mathit{false}, \mathsf{Bool}, \varnothing] : \mathsf{Bool}} \text{ (Val False)}$$

$$\frac{\Gamma_T \rhd \Gamma_I \vdash \mathsf{Tri}}{\Gamma_T \rhd \Gamma_I \vdash [\mathit{unknown}, \mathsf{Tri}, \varnothing] : \mathsf{Tri}} \text{ (Val Unknown)} \qquad \frac{\Gamma_T \vdash \mathsf{BT} \quad \Gamma_T \rhd \Gamma_I \vdash A : \mathsf{BT}}{\Gamma_T \rhd \Gamma_I \vdash (\neg A) : \mathsf{BT}} \text{ (Val Neg)}$$

$$\frac{\Gamma_T \vdash \mathsf{Bool}}{\Gamma_T \vdash \mathsf{Tri} <: \mathsf{Bool}} \text{ (Bool Sub)} \qquad \frac{\Gamma_T \vdash \mathsf{Tri}}{\Gamma_T \vdash \mathsf{Tri} <: \mathsf{Universal}} \text{ (Tri Sub)}$$

$$\frac{\Gamma_T \rhd \Gamma_I \vdash G : \mathsf{Bool} \quad \Gamma_T \rhd \Gamma_I \vdash A : \mathsf{Universal} \quad \Gamma_T \rhd \Gamma_I \vdash B : \mathsf{Universal}}{\Gamma_T \rhd \Gamma_I \vdash (\text{if } G \text{ then } A \text{ else } B \text{ fi}) : \mathsf{Universal}} \text{ (Val Cond)}$$

$$\frac{\Gamma_T \vdash \mathsf{BT} \quad \Gamma_T \rhd \Gamma_I \vdash A : \mathsf{BT} \quad \Gamma_T \rhd \Gamma_I \vdash B : \mathsf{BT}}{\Gamma_T \rhd \Gamma_I \vdash (A \mathcal{F} B) : \mathsf{Bool}} \text{ (Val Binary)}$$

Definition 26 (Slot Set) *Given* $\Gamma_T \vdash \langle N, \{T\}, \{(S, P)\}\rangle$*, call the set of all slots mentioned in* N*, and all of* N*'s parent types, the* ***slot set*** *of* N*. This set is denoted by* $N_\top$*.*

Definition 27 (ValueSlot Set) *Given* $\Gamma_T \rhd \Gamma_I \vdash [I, T, \{(S, P = \widetilde{V})\}]$*, call the set of all valueslots mentioned in* I *the* ***valueslot set*** *of* I*. This set is denoted by* $I_\top$*.*

4.4.2.5 Disjoint Types

Union types are recognized as useful constructs when it comes to general purpose specification languages. The rules of Table 4.13 define union types. Typically one defines rules for checking the type of, and accessing the values of, the left and right halves of a union type and instance. This method introduces four rules and a dependence upon the type Bool. Instead, the elegant

Table 4.9: Rules for basic type String

$\mathcal{F} \in \{=, \neq, <\}$ is an operator on instances of the type String.

$$\frac{\Gamma_T \vdash \mathsf{String}}{\Gamma_T \triangleright \Gamma_I \vdash [\lambda, \mathsf{String}, \varnothing] : \mathsf{String}} \text{(Val Blank)}$$

$$\frac{\Gamma_T \vdash \mathsf{String} \quad \Gamma_T \triangleright \Gamma_I \vdash A : \mathsf{String} \quad \Gamma_T \triangleright \Gamma_I \vdash B : \mathsf{String}}{\Gamma_T \triangleright \Gamma_I \vdash (A \; \mathcal{F} \; B) : \mathsf{Bool}} \text{(Val Bin String)}$$

$$\frac{\Gamma_T \vdash \mathsf{String} \quad \Gamma_T \triangleright \Gamma_I \vdash A : \mathsf{String} \quad \Gamma_T \triangleright \Gamma_I \vdash B : \mathsf{String}}{\Gamma_T \triangleright \Gamma_I \vdash (A + B) : \mathsf{String}} \text{(Val String Concat)}$$

case construct is used in rule (Val Case) to define exactly the same thing in a single rule. Note that the | operator separates the two options of the case statement.

4.4.2.6 Introduction Rules

The rules of contexts are listed in Table 4.14.

Since types in T_Γ can reference other types via slots, then they are necessarily recursive types. Typically, the use of recursive types introduces extra complexity to the logic and the proofs with and about the logic [95, 141]. Type introduction is defined in a novel manner that eliminates these complexities.

First, the base cases of empty type and instance contexts are captured in the rules (Empty Type Intro) and (Empty Inst Intro).

The primary rules for introduction are (Type Intro) and (Inst Intro). These two rules ensure that type and instance contexts remain well-formed and sound. The subscript function $e()$ is used to denote the index of the end of a list of values.

(Type Intro) states that, to introduce a new type N, N must be a new type name, N's parent types must be well-defined, N's slot types must be well-defined, and no slot introduced in N is used in any parent of N. The introduction of an entire well-formed type context into the current context is supported. As a result, this rule handles the introduction of a single type as well as a well-formed (possibly recursive) type set.

Likewise, (Inst Intro) denotes something very similar to (Type Intro). A new instance I must have an unused name, I's type must be well-defined, the slots of I's valueslot set must be a subset of the slot set of I's type, and every property mentioned in I's valueslots must be well-

Table 4.10: Rules for basic types Int and Float

$\mathcal{F} \in \{=, \neq, <, +, -, *, /\}$ is an operator on instances of the type Int. The variable i represents any integer and f represents any floating point number.

$$\frac{\text{(Val Int)}\quad \Gamma_T \vdash \mathsf{Int}}{\Gamma_T \rhd \Gamma_I \vdash [i, \mathsf{Int}, i] : \mathsf{Int}} \qquad \frac{\text{(Val Float)}\quad \Gamma_T \vdash \mathsf{Float}}{\Gamma_T \rhd \Gamma_I \vdash [f, \mathsf{Float}, f] : \mathsf{Float}}$$

$$\frac{\text{(Val Bin Int)}\quad \Gamma_T \vdash \mathsf{Int} \quad \Gamma_T \rhd \Gamma_I \vdash A : \mathsf{Int} \quad \Gamma_T \rhd \Gamma_I \vdash B : \mathsf{Int}}{\Gamma_T \rhd \Gamma_I \vdash (A\ \mathcal{F}\ B) : \mathsf{Int}}$$

$$\frac{\text{(Val Bin Float)}\quad \Gamma_T \vdash \mathsf{Float} \quad \Gamma_T \rhd \Gamma_I \vdash A : \mathsf{Float} \quad \Gamma_T \rhd \Gamma_I \vdash B : \mathsf{Float}}{\Gamma_T \rhd \Gamma_I \vdash (A\ \mathcal{F}\ B) : \mathsf{Float}}$$

defined. Since all collections of slots and valueslots are sets, multi-matching redundant entries is not possible.

The interpretation of (Inst Intro) is what makes T_Γ a recursive type system. Finally, rule (Context Intro) describes how well-formed contexts are unified to build new well-formed contexts.

4.4.2.7 Recursive Types

nil types and *nil* instances must be expressible with respect to T_Γ, even though no symbol in T_Γ exists to represent such a notion. Implicit in the definition of T_Γ are the use of the *nil* type and *nil* instance.

For the purpose of this discussion, assume that a type NilType exists that is the parent type of all types. Likewise, assume that an instance of name *nil* of type NilType exists. Since the type NilType and instance *nil* are not part of the syntax of T_Γ, there is no possibility of name collision with newly introduced constructs.

Definition 28 (Recursive Types) ***Simple Case:*** *Let T be a type. Assume T has a slot of the form (T, P); i.e., the type of the slot is T itself. T is called a* ***simple recursive type.***

Complex Case: *Let U and V be distinct types. Assume that a finite sequence of slot inclusions exists between U and V: U has a slot of the form (W, P), W has a slot of the form (W', P'), ..., and the final type W^* has a slot of the form (U, P^*). U and V are called* ***complex recursive types*** *because their slot type dependency forms a cycle.*

Definition 29 (Instances of Recursive Types) *Let T be a simple recursive type. Let I be an*

Table 4.11: Rules for basic types Pair and Triple

$$\frac{\Gamma_T \vdash A_1 \quad \Gamma_T \vdash A_2}{\Gamma_T \vdash A_1 \times A_2}\ \text{(Type Pair)} \qquad \frac{\Gamma_T \vdash A_1 \quad \Gamma_T \vdash A_2 \quad \Gamma_T \vdash A_3}{\Gamma_T \vdash A_1 \times A_2 \times A_3}\ \text{(Type Triple)}$$

$$\frac{\Gamma_T \rhd \Gamma_I \vdash M_1 : A_1 \quad \Gamma_T \rhd \Gamma_I \vdash M_2 : A_2}{\Gamma_T \rhd \Gamma_I \vdash \langle M_1; M_2 \rangle : A_1 \times A_2}\ \text{(Val Pair)}$$

$$\frac{\Gamma_T \rhd \Gamma_I \vdash M_1 : A_1 \quad \Gamma_T \rhd \Gamma_I \vdash M_2 : A_2 \quad \Gamma_T \rhd \Gamma_I \vdash M_3 : A_3}{\Gamma_T \rhd \Gamma_I \vdash \langle M_1; M_2; M_3 \rangle : A_1 \times A_2 \times A_3}\ \text{(Val Triple)}$$

$$\frac{\Gamma_T \rhd \Gamma_I \vdash M : A_1 \times A_2}{\Gamma_T \rhd \Gamma_I \vdash \mathit{first}\, M : A_1}\ \text{(Val First Pair)} \qquad \frac{\Gamma_T \rhd \Gamma_I \vdash M : A_1 \times A_2}{\Gamma_T \rhd \Gamma_I \vdash \mathit{second}\, M : A_2}\ \text{(Val Second Pair)}$$

$$\frac{\Gamma_T \rhd \Gamma_I \vdash M : A_1 \times A_2 \times A_3}{\Gamma_T \rhd \Gamma_I \vdash \mathit{first}\, M : A_1}\ \text{(Val First Triple)} \quad \frac{\Gamma_T \rhd \Gamma_I \vdash M : A_1 \times A_2 \times A_3}{\Gamma_T \rhd \Gamma_I \vdash \mathit{second}\, M : A_2}\ \text{(Val Second Triple)} \quad \frac{\Gamma_T \rhd \Gamma_I \vdash M : A_1 \times A_2 \times A_3}{\Gamma_T \rhd \Gamma_I \vdash \mathit{third}\, M : A_3}\ \text{(Val Third Triple)}$$

Table 4.12: Rules for basic type List

$$\frac{\Gamma_T \vdash \diamond}{\Gamma_T \vdash \mathsf{List}}\ \text{(List Type)} \qquad \frac{\Gamma_T \vdash \mathsf{List}}{\Gamma_T \rhd \Gamma_I \vdash [\mathit{nil}, \mathsf{List}, \varnothing] : \mathsf{List}}\ \text{(Val Nil)} \qquad \frac{\Gamma_T \rhd \Gamma_I \vdash L_0, \ldots, L, \ldots, L_n : \mathsf{List}}{\Gamma_T \rhd \Gamma_I \vdash L : \mathsf{List}}\ \text{(List Inclu)}$$

instance of T. A slot is not mentioned in an instance introduction if and only if the value of that slot is nil.

The existence of this implicit *nil* provides a fixed point for recursive types so that they are well-founded.

4.4.2.8 Examples

In the following examples, a variant of the syntax that is introduced by the algebra $A_{\mathcal{T}}$ in Section 4.6 is used. The syntax is basic enough that it is comprehensible without introduction. The string "`mt`" represents the empty set ($\varnothing$) as a kind of phonetic pun.

Example 10 (Types) ***Non-recursive types:*** *The set of basic types (**Basic**) are all non-recursive types. For example, `<Int, Universal, mt>` is the integer basic type. This type has no slots.*

Here are three more example non-recursive types:

Table 4.13: Union types

$$\frac{\Gamma_T \vdash A_1 \quad \Gamma_T \vdash A_2}{\Gamma_T \vdash A_1 + A_2}\ (\text{Type Union}) \qquad \frac{\Gamma_T \triangleright \Gamma_I \vdash M_1 : A_1 \quad \Gamma_T \vdash A_2}{\Gamma_T \vdash \mathit{inLeft}_{A_2} M_1 : A_1 + A_2}\ (\text{Val inLeft}) \qquad \frac{\Gamma_T \vdash A_1 \quad \Gamma_T \triangleright \Gamma_I \vdash M_2 : A_2}{\Gamma_T \vdash \mathit{inRight}_{A_1} M_2 : A_1 + A_2}\ (\text{Val inRight})$$

$$\frac{\Gamma_T \triangleright \Gamma_I \vdash M : A_1 + A_2 \quad \Gamma_T \triangleright \Gamma_I x_1 : A_1 \vdash N_1 : B \quad \Gamma_T \triangleright \Gamma_I x_2 : A_2 \vdash N_2 : B}{\Gamma_T \triangleright \Gamma_I \vdash (\text{case}_B\ M\ \text{of}\ x_1 : A_1\ \text{then}\ N_1 |\ x_2 : A_2\ \text{then}\ N_2) : B}\ (\text{Val Case})$$

```
< NamedThing ; Universal ; {(String, name)} >
< PhysicalEntity ; Universal ; {(Float, height) (Float, weight)} >
< Person ; {Universal, NamedThing, PhysicalEntity} ; mt >
```

No slots are specified in `Person`*'s definition, thus* `Person`*'s slot list is exactly the union of its parent types' slot lists:*

$$Person_{\top} = \{(\texttt{String}, \texttt{name}), (\texttt{Float}, \texttt{height}), (\texttt{Float}, \texttt{weight})\}$$

A simple recursive type: *A typical type of this variety is the type* `List`*:*

```
< List ; Universal ; { (Int, value) (List, next) } >
```

A complex recursive type: *The simplest possible, nontrivial example is as follows:*

```
< A ; Universal ; { (B, b) } >
< B ; Universal ; { (A, a) } >
```

Example 11 (Instances) ***Instances of non-recursive types:*** *An example instance of* `Person` *is as follows:*

```
[ Joe ; Person ;
 { (String, name) = Joe Kiniry
    (Float, height) = 6.05
    (Float, weight) = 89 } ]
```

Instances of a simple recursive type:

```
[ head ; List ; { (Int, value) = 5 (List, next) = middle } ]
[ middle ; List ; { (Int, value) = 3 (List, next) = last } ]
[ last ; List ; { (Int, value) = 0 } ]
```

Instances of complex recursive types:

Table 4.14: Introduction rules for T_Γ

The domain of a context Γ, denoted $dom(\Gamma)$, is the set of all grounds on which Γ is defined.

$$\frac{\Gamma_T \rhd \Gamma_I \vdash \diamond \quad \varnothing \vdash \diamond}{\Gamma_T, \varnothing \rhd \Gamma_I \vdash \diamond} \text{ (Empty Type Intro)} \qquad \frac{\Gamma_T \rhd \Gamma_I \vdash \diamond \quad \varnothing \vdash \diamond}{\Gamma_T \rhd \Gamma_I, \varnothing \vdash \diamond} \text{ (Empty Inst Intro)}$$

$$\frac{\begin{array}{c}\Gamma_T \rhd \Gamma_I \vdash P \quad {\Gamma_T}' \vdash \diamond \quad \Gamma_T, {\Gamma_T}' \vdash T_i \quad \Gamma_T, {\Gamma_T}' \vdash S_j \\ (S_j, P_j) \notin T_{i\top} \quad N \notin dom(\Gamma_T) \\ {\Gamma_T}' \vdash \langle N, \{T_0 \dots T_n\}, \{(S_0, P_0) \dots (S_m, P_m)\}\rangle\end{array}}{\Gamma_T, {\Gamma_T}' \vdash N} \quad (i = 0 \dots n, j = 0 \dots m) \text{ (Type Intro)}$$

$$\frac{\begin{array}{c}\Gamma_T \vdash \langle N, \{T\}, \{(S,P)\}\rangle \quad \Gamma_T \rhd \Gamma_I \vdash \diamond \quad I \notin dom(\Gamma_I) \\ (S,P) \in N_\top \quad \Gamma_T \rhd \Gamma_I, \Gamma_I' \vdash V_0^i \dots \Gamma_T \rhd \Gamma_I, \Gamma_I' \vdash V_{e(i)}^i \\ \Gamma_I' \vdash [I, N, \{(S_0, P_0 = V_0^0, \dots, V_{e(0)}^0) \dots (S_n, P_n = V_0^n, \dots, V_{e(n)}^n)\}]\end{array}}{\Gamma_T \rhd \Gamma_I, \Gamma_I' \vdash I} \quad (i = 0 \dots n) \text{ (Inst Intro)}$$

$$\frac{\begin{array}{c}\Gamma_T \rhd \Gamma_I \vdash \diamond \quad {\Gamma_T}' \rhd \Gamma_I' \vdash \diamond \quad \Gamma_T, {\Gamma_T}' \vdash \diamond \\ \Gamma_T, {\Gamma_T}' \rhd \Gamma_I \vdash \diamond \quad \Gamma_T, {\Gamma_T}' \rhd \Gamma_I' \vdash \diamond\end{array}}{\Gamma_T, {\Gamma_T}' \rhd \Gamma_I, \Gamma_I' \vdash \diamond} \text{ (Context Intro)}$$

```
*** No recursion.
[ A0 ; A ; { (B, b) = B0 } ]
[ B0 ; B ; mt ]

*** Simple loop.
[ A1 ; A ; { (B, b) = B1 } ]
[ B1 ; B ; { (A, a) = A1 } ]

*** More complex example.
[ A2 ; A ; { (B, b) = B2 } ]
[ B2 ; B ; { (A, a) = A3 } ]
[ A3 ; A ; { (B, b) = B1 } ]
```

That concludes a short set of examples of types. Next, T_Γ is examined from an operational point of view.

4.4.3 Operational Definition of Judgments

Next, all the judgments on T_Γ are operationally defined. Additionally, the operations mentioned in Definitions 25 through 27 are defined.

The goal here is to show not only that the proof system derived from T_Γ is *decidable*, but that it is *efficient* as well. This result motivates the definition of an algebra $A_\mathcal{T}$ in Section 4.6 that is a model for T_Γ.

4.4.3.1 Tuples, Sets, Lists, and Existential Operators

Tuples, sets, lists, and existential operators are used in the operational definitions that follow. An external logic (namely, first order predicate logic) is appealed to in determining the decidability of T_Γ. The use of these constructs and operators is examined in more detail when the decidability of T_Γ is analyzed in Section 4.7.

4.4.3.2 Summary of Operations

Each primary judgment discussed above is captured as an operation here with codomain Bool. By analyzing the decidability of these operations, a determination of the decidability of T_Γ as a proof system is made.

Definition 30 (Operations on Type System) *Define the following operations on T_Γ. Types listed*

on the left of a → are the domain of the operation; those on the right are its codomain.

Type Judgments

is-type? : *TypeContext Ground → Bool*

is-subtype-of? : *TypeContext Ground Ground → Bool*

is? : *TypeContext Ground Ground → Bool*

is-part-of? : *TypeContext Ground Ground → Bool*

type-of? : *TypeContext Set[Ground] → Set[Ground]*

slots-of? : *TypeContext Set[Ground] → Set[Slot]*

Instance Judgments

is-instance? : *Environment Ground → Bool*

is-a-type-of? : *Environment Ground Ground → Bool*

is? : *Environment Ground Ground → Bool*

is-part-of? : *Environment Ground Ground → Bool*

type-of? : *Environment Ground → Set[Ground]*

4.4.3.3 Type Judgments

Below are the operational definitions of the type judgments on T_Γ.

Definition 31 (is-type?)

$$\textit{is-type?}(\Gamma_T, G) = \begin{cases} \textit{true} & \textit{if } \exists \langle N, \{T\}, \{S\} \rangle \in \Gamma_T \textit{ such that } G = N, \\ \textit{false} & \textit{otherwise.} \end{cases}$$

This judgment is written as $\Gamma_T \vdash G$.

Definition 32 (is-subtype-of?)

$$\textit{is-subtype-of?}(\Gamma_T, G, G') = \begin{cases} \textit{true} & \textit{if } \exists \langle N, \{T\}, \{S\} \rangle \in \textit{type-of?}(\Gamma_T, G) \textit{ such that } G' = N, \\ \textit{false} & \textit{otherwise.} \end{cases}$$

The judgment is-subtype-of?(Γ_T, T, U) is written as $\Gamma_T \vdash T <: U$.

Definition 33 (is? for Types)

$$is?(\Gamma_T, G, G') = \begin{cases} true & \text{if } G = G' \wedge \textit{ is-type?}(\Gamma_T, G), \\ false & \text{otherwise.} \end{cases}$$

The operation *is?* is defined as by-name equality on types within a context. It has no equivalent judgment beyond equality on symbols.

Definition 34 (is-part-of? for Types)

$$\textit{is-part-of?}(\Gamma_T, G, G') = \begin{cases} true & \exists \langle N, \{T\}, \{S\} \rangle \in \Gamma_T \textit{ such that } G = N \wedge \\ & \exists \langle N', \{T'\}, \{S'\} \rangle \in \Gamma_T \textit{ such that } G' = N' \wedge \\ & [\exists (S, \widetilde{P}) \in S' \textit{ where } G = S] \vee \\ & \quad [\exists \langle N'', \{T''\}, \{S''\} \rangle \in \Gamma_T \textit{ such that} \\ & \quad\quad \langle N'', \{T''\}, \{S''\} \rangle \in \textit{type-of?}(\Gamma_T, N') \wedge \\ & \quad\quad \exists (S', \widetilde{P'}) \in S'' \textit{ where } G = S'] \\ false & \text{otherwise.} \end{cases}$$

The judgment is-part-of?(Γ_T, T, U) is written as $\Gamma_T \vdash T \subset U$.

4.4.3.4 Type Operations

Defining a few operations on type contexts are useful as well.

Definition 35 (type-of? for Types) *The term GS denotes an instance of the sort* `GroundSet`.

$$\textit{type-of?}(\Gamma_T, GS) = \begin{cases} error & \text{if } GS \notin dom(\Gamma_T) \\ Universal & \text{if } GS = \{Universal\} \\ \textit{type-of?}(\Gamma_T, T) \cup T_{<:} & \text{if } \exists \langle N, \{T\}, \{S\} \rangle \in \Gamma_T \textit{ such that} \\ & GS = \{N\} \wedge \{T\} \neq \{Universal\}, \\ T \cup T' \cup \textit{type-of?}(\Gamma_T, T) \cup \textit{type-of?}(\Gamma_T, T') & \text{if } GS = \{T, T'\}, \\ T \cup \textit{type-of?}(\Gamma_T, T) \cup \textit{type-of?}(\Gamma_T, TS) & \text{if } GS = \{T\} \cup TS \wedge TS \neq nil \end{cases}$$

Definition 36 (slots-of?)

$$\textit{slots-of?}(\Gamma_T, GS) = \begin{cases} \textit{error} & \textit{if } GS \notin \textit{dom}(\Gamma_T) \\ \varnothing & \textit{if } GS = \{\textit{Universal}\} \\ G_\top \cup \textit{slots-of?}(\Gamma_T, \textit{type-of?}(\Gamma_T, G)) & \textit{if } GS = \{G\} \wedge \exists \langle N, \{T\}, \{S\}\rangle \in \Gamma_T \\ & \textit{such that } G = N \wedge G \neq \textit{Universal} \\ \textit{slots-of?}(\Gamma_T, N) \cup \textit{slots-of?}(\Gamma_T, TS) & \textit{if } GS = \{T\} \cup TS \wedge TS \neq \textit{nil} \end{cases}$$

4.4.3.5 Instance Judgments

Recall that without a type context, an instance context cannot be defined since there are no types.

Definition 37 (is-instance?)

$$\textit{is-instance?}(\Gamma_T, \Gamma_I, G) = \begin{cases} \textit{true} & \textit{if } \exists \langle I, T, \{V\}\rangle \in \Gamma_I \textit{ such that } G = I, \\ \textit{false} & \textit{otherwise.} \end{cases}$$

The judgment is-instance?(Γ_T, Γ_I, I) is written $\Gamma_T \rhd \Gamma_I \vdash I$.

Definition 38 (is-a-type-of?)

$$\textit{is-a-type-of?}(\Gamma_T, \Gamma_I, G, G') = \begin{cases} \textit{true} & \textit{if } \exists \langle I, T, \{V\}\rangle \in \Gamma_I \textit{ such that } G = I \wedge \\ & \textit{is-type?}(\Gamma_T, G') \wedge (T = G' \vee G' \in \textit{type-of?}(\Gamma_T, T)), \\ \textit{false} & \textit{otherwise.} \end{cases}$$

The judgment is-a-type-of?$(\Gamma_T, \Gamma_I, I, T)$ is written as $\Gamma_T \rhd \Gamma_I \vdash I : T$.

Definition 39 (is? for Instances)

$$\textit{is?}(\Gamma_T, \Gamma_I, G, G') = \begin{cases} \textit{true} & \textit{if } G = G' \wedge \textit{is-instance?}(\Gamma_T, \Gamma_I, G), \\ \textit{false} & \textit{otherwise.} \end{cases}$$

The operation is? is defined as by-name equality on instances within an environment. Like the corresponding operation on types, its equivalent judgment is equality on symbols.

Definition 40 (is-part-of? for Instances)

$$\textit{is-part-of?}(\Gamma_T, \Gamma_I, G, G') = \begin{cases} \textit{true} & \textit{is-instance?}(\Gamma_T, \Gamma_I, G) \wedge \\ & \textit{is-instance?}(\Gamma_T, \Gamma_I, G') \wedge \\ & \exists (S, P = \widetilde{V}) \in \{V'\} \textit{ where } G \in \widetilde{V} \\ \textit{false} & \textit{otherwise.} \end{cases}$$

The judgment is-part-of?$(\Gamma_T, \Gamma_I, A, B)$ is written as $\Gamma_T \rhd \Gamma_I \vdash A \subset B$.

4.4.3.6 Environment Operations

Finally, the following operation on environments are defined.

Definition 41 (type-of? for Instances in Environments)

$$\textit{type-of?}(\Gamma_T \rhd \Gamma_I, G) = \begin{cases} T \cup \textit{type-of?}(\Gamma_T, T) & \textit{if } \exists \langle I, T, \{V\} \rangle \in \Gamma_I \textit{ such that } G = I, \\ \textit{error} & \textit{otherwise.} \end{cases}$$

4.5 Model Mapping

In order to show that T_Γ is a model for the structural aspects of KT, a mapping needs to be specified between the sentences of the two logics.

Recall that the ground kinds of a context are subjective. The ground kinds for this model are the ground types of T_Γ. Each ground kind that was not introduced as part of the basic definition of KT must be defined as part of this mapping. This set of kinds includes FLOAT, STRING, and

The mapping between T_Γ and KT is as follows:

- The top kind א is mapped to the universal type Universal. Note that א has no substructure, so this mapping has no substructure.
- The ground kinds are mapped directly to their corresponding ground types:
  - BOOLEAN is mapped to Boolean
  - INTEGER is mapped to Integer
  - FLOAT is mapped to Float
  - STRING is mapped to String
  - TRI is mapped to Tri

- Likewise, basic instances of ground kinds map to corresponding instances of basic types. E.g, $\perp$ maps to $[\mathit{false}, \mathsf{Bool}, \varnothing]$.
- Every kind K in T_Γ is mapped to a type named K in KT.
- Every instance I is mapped to an instance named I.
- Every context Γ in T_Γ is mapped to an environment $\Gamma_T \triangleright \Gamma_I$ (kinds to types and instances to instances, appropriately).
- Structural operators map directly as well:

Sentences of the form:	Map to:
$I : K$	$I : K$
$K <_p L$	$\langle K, \{L\}, \ldots \rangle$
$K < L$	$K <: L$
$K \subset_p L$	$\langle L, \{\ldots\}, \{\ldots, (K, \ldots), \ldots\} \rangle$
$K \supset L$	$L \subset K$

- Composition of kinds is mapped to a type merge. Since different kinds are guaranteed different names, and composition is guaranteed to retain all structure, merging the valueslot sets of the two types is a legitimate model.
- Decomposition of kinds is mapped to a type split in the same fashion. The partial interpretations map like valueslots to like valueslots, and all other valueslots are forgotten.
- Functional kind are mapped to a new kind FUNCTION whose parts indicate domain and codomain. E.g., $U \rightarrow V$ maps to $\langle \mathsf{Function}, \{\mathsf{Universal}\}, \{(Universal, FunctionDomain = U)(Universal, FunctionCodomain = V)\} \rangle$.
- A predicate type is introduced whose parent type is Function and whose codomain is Bool.
- Truth structures map to appropriate types. The claim type is defined as follows:

$$\langle \mathsf{Claim}, \{\mathsf{Predicate}\}, (\mathsf{Predicate}, claim)(\mathsf{Evidence}, evidence) \rangle$$

  Likewise, a belief type is defined as:

$$\langle \mathsf{Belief}, \{\mathsf{Predicate}\}, (\mathsf{Predicate}, claim)(\mathsf{Evidence}, evidence)(\mathsf{Poset}, conviction) \rangle$$

- Computable behaviors are mapped to sets of rewrite rules across equational algebras. Non-terminating behaviors are mapped to sets of rewrite rules in rewriting logic. Canonical forms are exactly canonical terms.

This completes the mapping of terms in KT to terms in T_Γ.

4.6 The Algebra

An algebra called $A_\mathcal{T}$ which is a model for T_Γ is next summarized. The algebra $A_\mathcal{T}$ is built constructively via a series of term algebras, each of which was written directly from the rules of T_Γ and the operational definitions of judgments on T_Γ presented in Section 4.4.3.

The algebra $A_\mathcal{T}$ is defined using many order sorted algebra with equational logic. The full algebra has been specified and rigorously tested in the Maude system [70, 72, 73] and is reproduced in Appendix A.

The construction of the mapping between T_Γ and $A_\mathcal{T}$ is tedious. It is summarized as follows:

- Each basic *type* is mapped to an algebraic *sort*:

type Universal	$\longrightarrow$	sort `Universal`
type Boolean	$\longrightarrow$	sort `Bool`
type Tri	$\longrightarrow$	sort `Tri`
type Integer	$\longrightarrow$	sort `Int`
type Float	$\longrightarrow$	sort `Float`
type String	$\longrightarrow$	sort `Id`

- A new sort `Ground` is introduced which encapsulates the semantics of all basic sorts for a given type context into a single sort and module. Parameterization of the initial type context (as discussed in Section 4.4.2.4) is accomplished by the introduction of basic sorts in this definition.

- The basic type hierarchy is directly mapped to a basic sort hierarchy.

- A parameterized sort `Set[]` is introduced that captures the basic semantics of sets.

- A parameterized sort for the type List is introduced called `List[]` that covers the semantics of lists specified in Section 4.4.2.4.

- Several sorts are introduced that map to the various constructs used to define types, instances, and contexts. These sorts include `Property`, `Value`, `Slot`, and `ValueSlot`. The parameterized sort `2Tuple[]` is used to define `Slot` and `ValueSlot`.

- New sorts `Type` and `Instance` are defined using the parameterized sort `2Tuple[]` again, with appropriate axioms, to capture the semantics of the types Type and Instance.

- New sorts `TypeContext` and `InstanceContext` are defined using the parameterized sort `Set[]`, with appropriate axioms, to capture the semantics of Γ_T and Γ_I in T_Γ.
- Finally, all of the above is unified into a single theory with appropriate axioms and supplementary operators such that this new theory, called `KindTypeTheory` or $A_\mathcal{T}$, is a model for T_Γ.

4.7 Analysis of the Type System

Now that type theoretic and algebraic models for kind theory have been defined, thee computational properties of the models can be considered.

T_Γ is a standard constructive type system. Consequently, it only checks if a term is well-formed. It is incapable of reporting when, how, or why a term is not well-formed.

An interactive, user-centric system must be able to provide this latter information. This is a secondary motivation for the operational definition of T_Γ operators covered in the last section. This algebra provides the basis for such "when, how, why" capability.

Much like how early compilers failed on invalid input, the current realization of the algebra fails with a small amount of error handling on bad input. Refining this system with the goal of more well-defined, integrated, user interactivity is a future goal discussed in Chapter 8.

Two properties of this model are of interest: if it is *computable* (typing is decidable and operations are computable) and, if so, what is its operational computational *complexity*.

4.7.1 Computability and Decidability

Theorem 22 *T_Γ and its models constitute a proof system that is computable; meaning all operations are computable and typing is decidable.*

Proof. Witness that every sub-operation used in the operational definition of $A_\mathcal{T}$ is computable by construction. The general reasons that this fact holds are: (a) the type system is constructive, (b) the context is finite, (c) substructures of the system have structure that assists in computation (e.g., the partial order on types, canonical forms, etc.), and (d) the system is only first-order and does not open up decidability problems by significantly restricting the expressiveness. Thus, the proof theory associated with the logic T_Γ is decidable. □

4.7.2 Complexity

The complexity of T_Γ is measured by analyzing the in-logic (object level) and extra-logical (meta-level) operations that are defined on this open constructive type system. If the operational

definition of T_Γ is space-efficient and time-efficient, then any reasonable implementation of it is as well.

The operational definition uses several specification constructs. In particular, several data structures are used (tuples, sets, and lists) as is first-order existential quantification. This analysis leads to the following worst-case complexity measures for these constructs.

- **Tuples:** Creation, access, and manipulation of tuples are trivially $O(1)$ operations in space and time.
- **Sets:** Sets can be represented by binary trees. Accordingly, they are space complexity $O(n)$, where n is the number of elements in the set. Element lookup, insertion, and deletion are all $O(\log n)$ operations. The remaining set operations are as follows:
  - *cardinality* – $O(1)$ since the size of the set can be tracked during insertion and deletion operations.
  - *difference* – $O(n \log n)$ since every element of one set must be looked up in the other.
  - *intersection* – $O(n \log n)$ for the same reason.
  - *union* – $O(n \log n)$ since all the elements of one set must be inserted into the other.
- **Lists:** Lists can be represented as linked lists. They have space complexity $O(n)$ where n is the number of elements in the list. Element lookup is order $O(n)$, insertion is $O(1)$ since the lists are not ordered, and deletion is $O(1)$, presuming the element to be deleted has already been identified.
- **Exists:** All of the existential operations are on sets, thus all terms using $\exists$ are order $O(\log n)$ where n is the number of elements in the set.

Thus, not only are all operations defined by polynomial time algorithms, but the worst of them, the introduction rules (Type Intro) and (Inst Intro), are $O(n \log n)$. This fulfills the last requirement, that of efficiency.

Chapter 5

A Reflective Model

In this chapter the core notions of kind theory are defined in kind theory. This includes the set-based structures of Chapter 2 (e.g., instance, kind, agent, etc.) as well the core functions and relations (e.g., $<$, $\equiv$, etc.). Additionally, a basic ontology for core mathematics notions and some general purpose abstract data types are provided, both of which are important to the reflective definition of kind theory.

The theoretical framework of the last few chapters provides the axiomatic foundation for kind theory, but some extra information is necessary for an actual realization.

Part of that information was the model described in the last chapter. This chapter augments that base by describing (a) a set of standard ground kind, (b) a set of supplementary kinds used to define the universal properties of all kinds via the kind UNIVERSAL, and (c) a rewriting system that translates kind theory (*KT*) into T_Γ so that efficient reasoning about structures can be performed.

Finally, predicate kind and related base constructs (e.g., claims, beliefs, and guards) have several special uses in kind theory, particularly from the collaborative viewpoint. Basic definitions of all these constructs are provided as well as description of how they are represented within this specific model.

Essentially, a metalevel definition of kind theory is specified. Such a definition is used in two ways. First, it can be used within logical frameworks that expose a metalevel, providing a template for a metalevel realization of the theory. This is exactly how it is used within the logical framework chosen. Second, it can be used as a basic design structure for a realization of kind theory directly in a programming language, as discussed at the beginning of Chapter 4.

Complete coverage is not attempted in this chapter. The reflective definition is still a work in progress. The intent here is to (a) provide evidence for the comprehensiveness of kind theory through both a mathematical and a realized example, (b) provide an initial snapshot for a kind theory representation of the theory and model of kind theory. This second aspect assists in the consideration of alternative models so that new models and systems can be reliably constructed

and compared.

5.1 Core Notions

All of the notions from Section 2.1 must be formally accounted for within this theoretical framework beyond the set-based definitions of the foundations chapter.

A **construct** is a piece of knowledge or information that can be referenced. Since this definition is quite basic, the formal kind definition of construct is correspondingly uncomplicated.

Definition 42 (CONSTRUCT)

$$\text{CONSTRUCT} <_p \text{UNIVERSAL}$$

The notion of a construct adds nothing new to the base definition of UNIVERSAL since the latter includes referential information (see below).

Next, the notion of a reusable asset is defined.

Definition 43 (ASSET) *An **asset** is a construct that has reuse value. Thus, it has been, or can be, reused by other constructs.*

$$\text{ASSET} <_p \text{CONSTRUCT}$$

There are numerous examples of where aspects of the context can be made explicit in the core notion definitions. For example, the sentence $\text{REUSELIST} \subset_p \text{ASSET}$ might be added, where REUSELIST is a list of reuse instances defined as $\text{REUSELIST} <_p \text{LIST} \oplus \text{CONSTRUCT}$. While this imparts a certain representational clarity, it complicates the theory with unnecessary constructs. This information is implicitly available in the current context in the form of other sequents, e.g., every term of the form $I : K$.

Definition 44 (KIND) *The most reflective term, the kind* KIND *has the following definition:*

$$\text{KIND} <_p \text{ASSET}$$

Here too, terms like $\text{REALIZATIONS} \subset_p \text{KIND}$ and $\text{REALIZATIONS} <_p \text{LIST} \oplus \text{INSTANCE}$ can be defined to capture the list of all instances that realize this kind. But again, this unnecessarily grows the set of default kinds and largely complicates the model.

Instances are the other half (along with the above kind KIND) of kind theory's construct duality: everything is a classifier or a thing. Since instances and kinds both inherit from assets, and no equivalence relation is defined on them, then they are necessarily distinct.

Definition 45 (INSTANCE) *The kind* INSTANCE *is defined as*

$$\text{INSTANCE} <_p \text{ASSET}$$

Some basic notions are necessary for defining collections of kind. They are called CONTEXT and DOMAIN.

Definition 46 (CONTEXT) *A* CONTEXT *is a working set of kinds and instances.*

$$\text{CONTEXT} <_p \text{SET}[\text{KIND} \circ \text{INSTANCE}]$$

The kind SET is discussed in Section 5.2.4 and the semantics of parameterization (the bracket notation) later in this chapter in Section 5.2.2.

Definition 47 (DOMAIN) *A* DOMAIN *is a specialized partition of a context.*

$$\text{DOMAIN} <_p \text{CONTEXT}$$

All kinds are either behavior or data, depending upon the context. Therefore, kind that characterize this division are necessary.

Definition 48 (BEHAVIOR) BEHAVIOR *is a kind that indicates that a construct has at least one interpretation. Thus,*

$$\text{BEHAVIOR} <_p \text{KIND}$$

$$\text{BEHAVIORINTERP} <_p \text{INTERPRETATION}$$

$$\text{BEHAVIORINTERP} \subset_p \text{BEHAVIOR}$$

See below for the definition of INTERPRETATION.

Definition 49 (DATA) *The kind* DATA *denotes all things that do not have behavior.*

$$\text{DATA} <_p \text{KIND}$$

Definition 50 (AGENT) *An agent is an instance that can interpret. The kind* AGENT *is defined as*

$$\text{AGENT} <_p (\text{INSTANCE} \circ \text{BEHAVIOR})$$

5.1.1 Basic Functional Kind

Functional kinds deserve special attention. As previously discussed, functional kinds are classified into four primary kind: functions (the root of the inheritance hierarchy), partial interpretations, full interpretations, and computable functions.

First, the base notion of function is defined. A function is a fundamental behavioral construct. The notion of an entity that performs some mapping, interpreting input and producing output, is fundamental to mathematics, computation, and life.

Definition 51 (FUNCTION) *The kind* FUNCTION *is defined as follows. Since it is a topmost kind,* FUNCTION $<_p$ UNIVERSAL. *Additionally,*

$$\begin{array}{cc} \text{FUNCTIONDOMAIN} <_p \text{KIND} & \text{FUNCTIONCODOMAIN} <_p \text{KIND} \\ \text{FUNCTIONDOMAIN} \subset_p \text{FUNCTION} & \text{FUNCTIONCODOMAIN} \subset_p \text{FUNCTION} \end{array}$$

The only necessary aspects of a function are a well-defined ***domain*** *and* ***codomain****.*

Most people call the target of a function its **range**. The sentence

$$[\text{FUNCTIONCODOMAIN}] = \text{RANGE}$$

is appropriate for many users uncomfortable with the term codomain.

Next, the notion of a relation is defined.

Definition 52 (RELATION) *A relation is a special kind of function whose output is the set* $\{\top, \bot, ?\}$. *Thus,* RELATION $<_p$ FUNCTION *and*

$$\begin{array}{cc} \text{RELATIONCODOMAIN} \subset_p \text{RELATION} & \text{RELATIONCODOMAIN} <_p \text{FUNCTIONCODOMAIN} \\ \text{RELATIONCODOMAIN} <_p \text{TRI} & \text{TRI} <_p \text{SET}[\text{TERNARYTRUTHVALUES}] \\ \top, \bot, ? \subset_p \text{TRI} & \top, \bot, ? : \text{TERNARYTRUTHVALUES} \end{array}$$

Next, a notion of computation is needed to differentiation computable functions from uncomputable ones.

Definition 53 (COMPUTATION) *Any computation is realized by the functional kind* COMPUTATION *which captures the notion of a computable function.*

$$\text{COMPUTATION} <_p \text{FUNCTION}$$

Finally, the notion of interpretation must be defined. Interpretation is the keystone to understanding and manipulating kinds since it is the core structure-preserving function. The generic

notion of an interpretation is handled first, then it is refined to partial and full specializations.

Definition 54 (INTERPRETATION) *An **interpretation** is a function that preserves structure. Thus,* INTERPRETATION $<_p$ FUNCTION *and*

INTERPCONTEXT $<_p$ CONTEXT	INTERPCONTEXT $\subset_p$ INTERPRETATION
INTERPAGENT $<_p$ AGENT	INTERPAGENT $\subset_p$ INTERPRETATION
CLASSIFICATIONFUNCTION $<_p$ COMPUTATION	CLASSIFICATIONFUNCTION $\subset_p$ INTERPRETATION
DATAINTERPRETATION $<_p$ FUNCTION	DATAINTERPRETATION $\subset_p$ INTERPRETATION
BEHAVIORINTERPRETATION $<_p$ FUNCTION	BEHAVIORINTERPRETATION $\subset_p$ INTERPRETATION

Interpretation has a more subtle functional kind than one might suspect. General interpretations are not necessarily computable functions, but they are interpretable by some agent. This puts them in that fuzzy area somewhere in between that which is computable and that which is capable of being evaluated.

This kind is currently called ORACULARFUNCTION. This name was chosen to pay tribute to the oracles of computing theory and the devils of distributed systems. It is meant to connote the idea of an oracle capable of answering queries, right or wrong, regardless of the properties of the behavioral form itself. An alternative term is HALTINGFUNCTION because if the focus is on the fact that the construct always provides an answer, then its behavior necessarily halts, no more and no less.

Now interpretation is specialized to its partial and full versions.

Definition 55 (PARTIALINTERPRETATION) *A* PARTIALINTERPRETATION *is a partial interpretive context. Thus it has (a) a **computable** function that classifies context elements as either data or behavior, and (b) a pair of functions, one for data and one for behavior, that preserve part of the structure of their domains.*

$$\text{PARTIALINTERPRETATION} <_p \text{INTERPRETATION}$$

Recall that the classifier and data/behavior functions were already defined in the general definition of INTERPRETATION, so there is no need to repeat them here.

An interpretation has several equivalent realizations, often represented as sets of rules in some formal rewriting system.

Definition 56 (FULLINTERPRETATION) *A* FULLINTERPRETATION *is a full interpretive context. Like a partial interpretation, it has a **computable** classification function as well as a pair of functions for operating on the classified data and behavior in such a manner that fully preserves*

structure.

$$\text{FULLINTERPRETATION} <_p \text{INTERPRETATION}$$

Note that the substructure of PARTIALINTERPRETATION is sufficient to fulfill the structural requirements of FULLINTERPRETATION, only the functions now have a different set of axioms associated with them.

5.1.2 Core Functional Kind

Lastly, the core functional kind of kind theory must be reflectively defined. This provides a reflective fixed-point for the inheritance relation and opens up the possibility for the refinement of core functional concepts like inheritance.

This process is not detailed here, except to note the names of these core functional kind as well as make some comments on this process. Each core functional kind's kind is provided in the set of lambda terms below. The functional kind used in each rule is a COMPUTATION because each rule is decidable and computable. The subscripts of the terms denote their members' kind. The full set of logical operators is the union of all the λ_is.

$$\begin{aligned}
\lambda_{(\text{KIND}\to\text{KIND})\otimes(\text{INSTANCE}\to\text{INSTANCE})} &= \{\subset_p, [\,], \to, \dashrightarrow, \rightsquigarrow, \hookrightarrow\} \\
\lambda_{\text{KIND}\to\text{KIND}} &= \{<_p\} \\
\lambda_{\text{INSTANCE}\to\text{KIND}} &= \{:\} \\
\lambda_{\text{INSTANCE}\otimes\text{KIND}\to\text{BOOLEAN}} &= \{<_r\} \\
\lambda_{[(\text{KIND}\otimes\text{KIND})\to\text{BOOLEAN}]\otimes[(\text{INSTANCE}\otimes\text{INSTANCE})\to\text{BOOLEAN}]} &= \{\circ, \otimes, \oplus, \supset, \equiv, =, \eqcirc, \lessdot\} \\
\lambda_{\text{KIND}\otimes\text{KIND}\to\text{BOOLEAN}} &= \{<\} \\
\lambda_{\text{KIND}\to\text{BOOLEAN}} &= \{\nabla\} \\
\lambda_{(\text{KIND}\otimes\text{KIND}\to\text{KIND})\otimes(\text{INSTANCE}\otimes\text{INSTANCE}\to\text{INSTANCE})} &= \{\cdot, \times, +\} \\
\lambda_{(\text{AGENT}\otimes\text{PREDICATE}\otimes\text{EVIDENCE})\to\text{TRI}} &= \{\gamma\} \\
\lambda_{(\text{AGENT}\otimes\text{PREDICATE}\otimes\text{EVIDENCE})\to(\text{TRI}\otimes\text{POSET})} &= \{\beta\}
\end{aligned}$$

For example,

$$\text{PARENT} <_p \text{KIND} \dashrightarrow \text{KIND}$$

defines the functional kind (at the metalevel) of the functional kind (at the object level) PARENT, which represents the basic inheritance function $<_p$.

The basic inference rules of kind theory associated with these core functional kind, as detailed in Section 2.9, are directly translated into the metalevel as well. Each is represented as an operation on a context. If a rule is reversible, then two functional kind are added to the context, one for each direction of the rule's application.

For example, the rule (Parent Is-a) is represented by an instance of the functional kind:

$$\text{ParentIsa} : \text{CONTEXT} \dashrightarrow \text{CONTEXT}$$

The actual realization of the rule only happens when a model for kind theory is available with an infrastructure for computation, like that described in the last chapter ($A_{\mathcal{T}}$). Therefore, the semantics of ParentIsa at the metalevel are described by a set of rewrite rules at the metalevel within Maude.

When a new notion of inheritance needs to be added to the kind context: (a) this ground inheritance kind is used as a parent in an inheritance relation: NEWINHERITANCE $<_p$ PARENT, and (b) the extra semantics of this new notion of inheritance are defined through the introduction of new functional kinds that operate on contexts and involve NEWINHERITANCE.

The members of the full set of core functional kind are named as follows: INHERITANCE, PARENT, INCLUSION, PARTOF, GENERALCOMPOSITION, DECOMPOSITION, COMPOSITION, EQUIVALENCE, EQUALITY, and REALIZATION.

5.2 Some Fundamental Kind

Several fundamental kind domains for core areas of mathematics and computer science are now introduced.

5.2.1 Mathematical Structures

As mentioned at the beginning of this chapter, at this time only the vocabulary of these structures is introduced. The full formal definitions for some of the core notions of mathematics are not provided.

This is an example of the refinement process used when manipulating kind. First a term is chosen and placed within the kind hierarchy. Next, perhaps a natural language definition is bound to the kind. At some stage the kind is axiomatized. Over time, a formal semantics might get bound to the kind. For example, a theory might be described generically but formally as an institution [158].

First, define some elements of mathematics with barely any structure at all. Assume that these kinds have the widely used, standard, loose definitions that are attributed to them in mathematics.

Definition 57 (Basic Elements of Mathematics)

$$\textsc{VocabularyOfMathematics} <_p \textsc{Domain} \qquad \textsc{MathematicalTerm} <_p \textsc{Kind}$$
$$\textsc{Definition} <_p \textsc{MathematicalTerm} \qquad \textsc{Fact} <_p \textsc{MathematicalTerm}$$
$$\textsc{Axiom} <_p \textsc{MathematicalTerm} \qquad \textsc{Theorem} <_p \textsc{MathematicalTerm}$$
$$\textsc{Lemma} <_p \textsc{Theorem} \qquad \textsc{Corollary} <_p \textsc{Theorem}$$
$$\textsc{Conjecture} <_p \textsc{Theorem} \qquad \textsc{Proof} <_p \textsc{MathematicalTerm}$$
$$\textsc{Predicate} <_p \textsc{Function} \qquad \textsc{Proposition} <_p \textsc{Predicate}$$
$$\textsc{Term} <_p \textsc{MathematicalTerm} \qquad \textsc{Sentence} <_p \textsc{Term}$$
$$\textsc{Sentence} <_p \textsc{List}[\textsc{Term}] \qquad \textsc{Formula} <_p \textsc{Sentence}$$

Definition 58 (Theory) *A theory is a domain of mathematics.*

$$\textsc{Theory} <_p \textsc{Domain}$$
$$\textsc{Set}[\textsc{MathematicalTerm}] \subset_p \textsc{Theory}$$

Definition 59 (Logic) *A logic is an axiomatic theory of mathematics with certain metalogical properties.*

$$\textsc{Logic} <_p \textsc{Theory}$$
$$\textsc{Set}[\textsc{Axiom}] \subset_p \textsc{Logic}$$
$$\textsc{MetalogicalClassifier} <_p \textsc{Kind}$$
$$\textsc{Set}[\textsc{MetalogicalClassifier}] \subset_p \textsc{Logic}$$
$$\textsc{Consistent} <_p \textsc{MetalogicalClassifier}$$
$$\textsc{Complete} <_p \textsc{MetalogicalClassifier}$$

Mathematical systems are often described as algebras or calculi. Of course, formal definitions for exactly what constitutes either of these classifiers is still debated within mathematical circles. Thus, only the terms are introduced and their structure is not characterized.

Definition 60 (Algebra and Calculus) *An algebra and a calculus are theories of mathematics with structural properties described in their appropriate kinds.*

$$\textsc{Algebra} <_p \textsc{Theory} \quad \textsc{Calculus} <_p \textsc{Theory}$$

5.2.2 Parameterization

A core construct in mathematics, programming languages, and many other related domains is parameterization. Thus, parameterization and its typical functional syntax is well-understood and comprehended by many communities. This motivates introducing to kind theory a parameterization construct whose syntax inherits from one of the original parameterized languages, OBJ3 [165].

Example 12 (An Example of Parameterization) *Sets are parameterized over the kind of their elements. So the term* SET[KIND] *is written for a special kind of refinement function* f : SET $\oplus$ KIND $\dashrightarrow$ SET[KIND]*, where the brackets in the codomain are a part of the symbol denoting the compositional kind.*

Definition 61 (PARAMETERIZATION)

$$\text{PARAMETERIZATION} <_p \text{COCOMPOSITION}$$

$$\text{PARAMETERIZATIONREWRITE} : \text{FULLINTERPRETATION}$$

$$\text{PARAMETERIZATIONREWRITE} \subset_p \text{PARAMETERIZATION}$$

The rewrite PARAMETERIZATIONREWRITE realizes the syntactic transformation of this bracketed notation to the core (co)compositional form that is unfamiliar to most people. Since all structure is preserved in this rewrite, it is a full interpretation.

Henceforth, the bracketed parameterized notation is used freely.

5.2.3 Truth

Next the basic elements of semantics are considered. Since kind theory heavily utilizes both classical and non-classical (deviant) logics, examples of both are reflectively defined.

5.2.3.1 Classical Logic

Two-valued truth values and logic are represented by the kinds BOOLEANTRUTHVALUES and BOOLEAN, respectively.

The truth values TRUE, FALSE, and UNKNOWN are all basic kinds as well. They do not have any structure at this point in time because they are used solely as constants in the current domains relating to truth and truth structures.

Definition 62 (BOOLEANTRUTHVALUES) BOOLEANTRUTHVALUES *are either* ***true*** *or* ***false.***

$$\text{BOOLEANTRUTHVALUES} <_p \text{DATA}$$
$$\top : \text{BOOLEANTRUTHVALUES} \qquad \bot : \text{BOOLEANTRUTHVALUES}$$

Definition 63 (BOOLEAN) *The kind* BOOLEAN *represents a* ***Boolean logic****. Thus,*

$$\text{BOOLEAN} <_p \text{LOGIC}$$
$$\text{BOOLEANTRUTHVALUES} \lessdot \text{BOOLEAN} \quad \textit{(a direct data mapping)}$$
$$\text{BOOLEANTRUTHVALUES} \subset_p \text{BOOLEAN} \quad \textit{(by partial equiv)}$$
$$\nvDash : \text{BOOLEAN} \quad \textit{(set theoretically)}$$
$$\mathbf{2} : \text{BOOLEAN} \quad \textit{(categorically)}$$

Since this is the first time that such a mix of kinds and realizations is being demonstrated, especially with regards to so many formal systems, some comments on this definition of BOOLEAN and related structures are necessary.

The first line states that the parent of the kind named BOOLEAN is LOGIC. Accordingly, Boolean logic is a logic and all the properties of general logics are inherited by Boolean logic. Within this current context, recalling Definition 59, that means that Boolean logic has a set of axioms and a metalogical classification, in particular, that it is consistent and complete.

The next two lines point out the relationship between the earlier defined BOOLEANTRUTHVALUES, which is a collection of data instances that represent Boolean truth values, and Boolean logic. In particular, all Boolean logics have constants that syntactically and semantically represent true and false, thus a direct data mapping exists between the constants of BOOLEANTRUTHVALUES from Definition 62 and those constants of the Boolean logic.

The third line of the definition is a logical inference of the definition of partial equivalence. More specifically, when a set of Boolean truth values (BOOLEANTRUTHVALUES) and an arbitrary Boolean logic (BOOLEAN) are canonically represented, it becomes clear that the truth values are part of the structure of the logic.

The last two lines are the most interesting of the bunch. Both point out instances of Boolean logics within two different mathematical formalisms.

The first line says that the two-valued set ($\nvDash$) within a standard realization of set theory (say **ZFC**) fully represents a Boolean logic. The second says that the category with two objects ($\mathbf{2}$) is equally capable. These two little lines summarize a great deal going on behind the scenes mathematically.

In particular, each operation of Boolean logic (which is not yet enumerated in this example, but a typical set would include at least negation and implication) can be mapped to either: (a)

a set theoretic operation on $\nvDash$, or (b) a category theoretic operation on **2**. Each of these is a standard exercise for mathematics undergraduates in foundational mathematics courses.

This example emphasizes the multidisciplinary nature of realizations within kind theory, even within a metamathematical domain.

5.2.3.2 Three-Valued Logic

Three-valued truth values and logic are represented by the kinds TERNARYTRUTHVALUES and TRI, respectively.

Definition 64 (TERNARYTRUTHVALUES) TERNARYTRUTHVALUES *are either* ***true, false,*** *or* ***unknown****.*

$$\text{TERNARYTRUTHVALUES} <_p \text{DATA} \qquad ? : \text{TERNARYTRUTHVALUES}$$
$$\top : \text{TERNARYTRUTHVALUES} \qquad \bot : \text{TERNARYTRUTHVALUES}$$

Definition 65 (TRI) *The kind* TRI *represents a* ***ternary logic****. Thus,*

$$\text{TRI} <_p \text{LOGIC}$$
$$\text{TERNARYTRUTHVALUES} \lessdot \text{TRI} \quad \textit{(a direct data mapping)}$$
$$\mathsf{Tri} : \text{TRI} \quad \textit{(within } T_\Gamma\textit{)}$$
$$\mathbf{3} : \text{TRI} \quad \textit{(categorically)}$$

The foundations of this ternary logic kind follow the same lines as the Boolean logic discussed above.

5.2.3.3 Inter-logic Reasoning

As an example of how reasoning about theories with kind theory, consider the following theorem, stated as an inheritance relation between two kinds.

Theorem 23

$$\text{TRI} <_p \text{BOOLEAN}$$

Proof. (A sketch) Map the unknown realization to the null terminal (ω) and the remainder of the structure is a direct mapping. □

This interpretation implies that the only logics that realize TRI are those that maintain validity (i.e., have identical truth tables) when moving from a classical to a three-valued logic. This constraint eliminates several useful three valued logics [178, 358].

The relationship between Boolean logic and classical logics is emphasized by the introduction of a kind CLASSICAL that is an full equivalence synonym for BOOLEAN. Recall that choosing which kind is canonical is entirely up to an agent's point of view. The computer scientist sticks with BOOLEAN, while the logician probably opts for CLASSICAL.

5.2.4 Abstract Data Types

Several structures are universally witnessed, especially in mathematics and within computing structures. Basic specifications for the kind SET, BAG, and LIST are all provided, all of which are specializations of the kind COLLECTION for the purposes of this discussion.

Definition 66 (COLLECTION) *A **collection** is any structure that aggregates assets. It is a **parameterized kind** realized as a function on kind. See Definition 61 for more details.*

$$\text{COLLECTION}[\text{ASSET}] <_p \text{FUNCTION}$$

The kinds SET, BAG, and LIST are specialized by imposing enough additional semantics to realize the the standard algebraic/axiomatic theories of each kind.

5.2.4.1 Set

Definition 67 (SET) *Sets are widely used and understood constructs of mathematics and are realized by a number of standard abstract data types. Below is a partial specification of the kind* SET.

$\text{ELEMENT} <_p \text{ASSET}$	$\text{SET} <_p \text{COLLECTION}[\text{ELEMENT}]$
$\text{BASICSETTHEORY} <_p \text{THEORY}$	$\text{SET} <_p \text{BASICSETTHEORY}$
$\text{ADDFUNCTION} <_p \text{COMPOSITION}$	$\text{REMOVEFUNCTION} <_p \text{DECOMPOSITION}$
$\text{INCLUSIONFUNCTION} <_p \text{INCLUSION}$	$\text{ADDFUNCTION} \subset_p \text{BASICSETTHEORY}$
$\text{REMOVEFUNCTION} \subset_p \text{BASICSETTHEORY}$	$\text{INCLUSIONFUNCTION} \subset_p \text{BASICSETTHEORY}$
$\texttt{Set} : \text{BASICSETTHEORY}$ *(algebraically)*	$\boldsymbol{SET} : \text{BASICSETTHEORY}$ *(categorically)*

`Set` is the algebraic theory of sets (see below). ***SET*** is the category of infinite sets. These are certainly not the only two realizations of the kind SET.

The realizations and resulting interpretations for this definition are straightforward. For example, the operation `__` in the algebraic theory that defines `Set` (see Appendix A.2 and the next definition for details) maps to the kind ADDFUNCTION.

The algebra `Set` realizes the semantics of basic set theory. A variant of it is provided as part of many algebraic specification software systems.

The specific realization is specified in parameterized many order sorted algebra with Full Maude [104]. Only the signature of the algebra here; the full equational semantics are found in the Maude distribution[1].

This algebra is significantly more expressive than that which is needed to realize BASICSETTHEORY. Extra operators are mapped to the null terminal ω, as defined in the definition of the kind INTERPRETATION earlier.

Definition 68 (The Algebraic Theory of Sets) *This algebra is specified with Full Maude:*

```
(fmod SET[X :: TRIV] is
  protecting MACHINE-INT .

  sorts Set[X] Set?[X] .
  subsorts Elt.X < Set[X] < Set?[X] .

  op err : -> Set?[X] [ctor] .
  op mt : -> Set[X] [ctor] .
  op __ : Set[X] Set[X] -> Set[X] [ctor assoc comm id: mt] .
  op _in_ : Elt.X Set[X] -> Bool .
  op _in_ : Set[X] Set[X] -> Bool .
  op delete : Elt.X Set[X] -> Set[X] .
  op |_| : Set[X] -> MachineInt .
  op _-_ : Set[X] Set[X] -> Set[X] . *** difference
  op _&_ : Set[X] Set[X] -> Set[X] [ctor assoc comm] . *** intersection
  ...
endfm)
```

5.2.4.2 Bag

Next, the classic abstract data type bag is examined.

Definition 69 (BAG) *Bags are sets with duplication. Thus,*

$$\text{BAG} <_p \text{SET} \qquad \text{BASICBAGTHEORY} <_p \text{BASICSETTHEORY}$$

$$\text{BAGTOSETREWRITE} : \text{PARTIALINTERPRETATION} \qquad \text{SETTOBAGREWRITE} : \text{FULLINTERPRETATION}$$

$$\text{BAGTOSETREWRITE} \subset_p \text{BAG} \qquad \text{SETTOBAGREWRITE} \subset_p \text{BAG}$$

$$\text{BAGADDFUNCTION} <_p \text{ADDFUNCTION} \qquad \text{BAGADDFUNCTION} \subset_p \text{BASICBAGTHEORY}$$

$$\texttt{Bag} : \text{BASICBAGTHEORY}$$

`Bag` is the algebraic theory of bags. Its specification includes a refined addition function (__) that can deal with element duplication, otherwise it is identical to `Set`. The refine-

[1] http://maude.csl.sri.com/

ments associated with this specialization are specified and performed by the interpretations BAGTOSETREWRITE and SETTOBAGREWRITE.

5.2.4.3 List

Finally, lists are considered.

Definition 70 (LIST) *Lists are ordered bags. Thus,*

$$\text{LIST} <_p \text{BAG}$$

$$\text{BASICLISTTHEORY} <_p \text{BASICBAGTHEORY}$$

$$\text{LISTTOBAGREWRITE} : \text{PARTIALINTERPRETATION}$$

$$\text{BAGTOLISTREWRITE} : \text{FULLINTERPRETATION}$$

$$\text{LISTTOBAGREWRITE} \subset_p \text{LIST}$$

$$\text{BAGTOLISTREWRITE} \subset_p \text{LIST}$$

$$\text{LISTADDFUNCTION} <_p \text{BAGADDFUNCTION}$$

$$\text{LISTREMOVEFUNCTION} <_p \text{REMOVEFUNCTION}$$

$$\text{LISTINCLUSIONFUNCTION} <_p \text{INCLUSIONFUNCTION}$$

$$\text{LISTADDFUNCTION} \subset_p \text{BASICLISTTHEORY}$$

$$\text{LISTREMOVEFUNCTION} \subset_p \text{BASICLISTTHEORY}$$

$$\text{LISTINCLUSIONFUNCTION} \subset_p \text{BASICLISTTHEORY}$$

$$\texttt{List} : \text{BASICLISTTHEORY}$$

The refinements in this definition are captured by the above rewrites as well. In particular, to convert from a list to a bag, the ordered nature of lists must be "forgotten."

5.2.4.4 Discussion

[86]

From a typical programming languages point of view (i.e., ADTs and/or type systems), the notions of bag and list are not traditionally directly related to sets, so why is it that BAG and LIST are subkinds of SET? The distinction holds because, from a kind theory point of view, if you understand a bag or a list, then you can understand a set, but only if you understand the refinement interpretation(s) that throw away notions of order and duplication.

Note also the connection between understanding such a kind relationship and the equivalence class of each kind. That is to say, if two kind are equivalent, then by Theorem 1 a mutual

subkinding relationship exists. Hence, by rule (Parent Interp), an interpretation can exist between the kind. That is not to say that such an interpretation does exist in the current context. In fact, if one does not exist, then the agent cannot understand the subkind relationship, but existence is not necessary for soundness. The lack of such an interpretation in context is an issue of refinement, not one of existence.

5.3 A Realization of Kind Theory

There are several recourses to define the core functions and relations of kind theory.

The refinement process for knowledge creation sometimes goes through a series of stages, each progressively more formal. In providing a reflective definition for kind theory, and as an example of its use, any one of these stages can be targeted. Examples ranging from the very informal to the very formal are equally representable.

For example, in Section 5.1 the definitions of some of the basic notions of kind theory were provided in a relatively informal manner. New terms were defined as kinds and their interrelationships were documented. Some of these informal definitions, particularly those of Section 5.2.1, relied on relatively formal external references (e.g., the notion of structure preserving, the definition of a logic, etc.), but no formal connection was provided between the kind theory construct and the external formal system.

Section 5.2.3 provides a full formal binding between assets and an external formal system, in this case T_{Γ} and $A_{\mathcal{T}}$. This full connection also exists in the discussion of abstract data types in Section 1.3.6.

5.3.1 Remaining Reflective Definitions

The process of defining the remainder of kind theory reflectively is a procedural exercise. Each definition, axiom (inference rule), theorem, term, etc. of the preceding and following chapters is converted verbatim into kind theory. Portions of this structure have already been defined using a Web-based architecture called the Jiki, which is discussed in Chapter 6. This process is not detailed because it is very lengthy and unrevealing.

Likewise, the realization of kind theory discussed in this chapter as well as Chapter 4 has been partially completed as well. The majority of this work was written in Full Maude. Bits and pieces were also written in normal Maude, OBJ3, and several programming languages including Eiffel, Java, and OCaml for experimentation purposes. This work is discussed in more detail in Chapter 8, and the full algebra is specified in Appendix A.

5.3.2 Type and Instance Realizations

A detailed realization of kind theory using type theory and order sorted algebra have been developed.

A type theory was developed exclusively to reason about asset well-formedness and perform efficient evaluations of the core judgments of kind theory. Thus, each inference rule and axiom of kind theory has a direct realization in T_Γ that is computable, deterministic, and computationally efficient.

The type theory is, in turn, realized by an algebra. This algebra has been specified in the metalogical algebraic rewriting framework Maude. By defining this model, not only is an executable specification of the logic realized, but reasoning about the model is possible with the algebraic tools provide with and for Maude. For example, the confluence of the algebra's rewrite rules [262] can be analyzed to determine whether the algebra is Church-Rosser [298]. This analysis has not yet been started.

5.3.3 Context Change Operators

Context change operators have not yet been discussed. Both generic realizations of the Δ operator, as described in Section 2.10, as well as the introduction of new inference rules via inheritance from core reflective functional kind such as INHERITANCE must be considered.

Currently, context change operators are realized in a manual fashion. If the computational framework used to realize a kind system has reflective capabilities, then operations can be defined at the metalevel, and exposed at the object-level, for correct manipulation of context. The logical framework Maude is used in exactly this fashion. Additionally, consideration is made in Chapter 8 for using languages like ML, CLOS, and Smalltalk for realizations of kind systems exactly because they fully expose metalevel operations.

If the computational framework does not provide reflective capabilities (as in OBJ3), then the basic context manipulation interface must be used and conformance is now an onus on the agent. For example, adding a new inference rule to kind theory while using OBJ3 entails (a) defining a new theory that inherits from some base theory, and (b) introducing a new set of equational rewrites that realize the new rule's semantics.

All the while, the agent (the person or system making these manual additions to the OBJ3 context) must guarantee that the new algebra maintains structure per the semantics of the Δ operator. Within an algebraic framework this is straightforward in practice because the kind theoretic semantics map directly to the algebraic semantics of theory inheritance (the "no *junk* and no *confusion*" rule). Validating such is another matter entirely, and most (if not all) algebraic frameworks provide no mechanism for this.

5.3.4 Inter-theory Interpretations

As discussed in Chapter 4, T_Γ is a full realization of the structural core of kind theory. Recall that the name *KT* is used to denote the whole of kind theory. $A_\mathcal{T}$ is an algebra that models T_Γ.

Thus, to define a model for *KT*, the following set of interpretations are defined:

$$KTtoT_\Gamma <_p \textsc{PartialInterpretation} \quad T_\Gamma toKT <_p \textsc{FullInterpretation}$$
$$T_\Gamma toA_\mathcal{T} <_p \textsc{FullInterpretation} \quad A_\mathcal{T} toT_\Gamma <_p \textsc{FullInterpretation}$$

Each interpretation maps the key constructs from one domain to another. For example, $KT \rightsquigarrow T_\Gamma$ maps kinds to types, subkinding to subtyping, containment judgments to the is-part-of? type judgment, etc.

Each basic kind is mapped to a ground type. Such a mapping for a kind *K* is called a canonical type realization of *K*. Each mapping of a kind via a type to a construct in the algebra $A_\mathcal{T}$ is called a canonical instance realization.

Thus, the canonical type realization of kind BOOLEAN is the type `Boolean` (see Section 4.3.1). Its canonical instance realization is the `Bool` sort in the Maude realization of OSA.

Likewise, the canonical type realization of kind TRI is the type `Tri`. The canonical instance realization is the `Tri` algebraic sort.

5.3.5 Core Supplementary Kind

A set of core supplementary kinds also must be defined to provide a useful realization of kind theory. These are not ground kind, but are used in the definition of the universal properties of kind via a realization of the kind UNIVERSAL.

A simplified syntax is used to present the definitions of these kind. An example suffices to explain this syntax. Recall the discussion in Section 2.10.2.2 where it was pointed out that introducing a new kind meant (reflectively) also introducing a new instance of the kind KIND.

Example 13 (Kind Definition Simplified Syntax) *A sentence of the form*

$$\begin{aligned} \textsc{KindName} \triangleq\ & \langle field_0 = value_0, \\ & parentkind = \textsc{ParentKind}, \\ & \ldots \\ & field_n = value_n \rangle \end{aligned}$$

means that a new kind named KINDNAME *is defined whose parent kind is defined via the field* parentkind. KINDNAME *is also defined by a realization of the kind* KIND. *Within this realization,*

each named field is realized by the given data, each representing a simplified instance of the appropriate kind.

Lets look at two real supplementary kinds in detail to get a firmer grasp on this syntax which is very similar to that which is realized in the algebra $A_{\mathcal{T}}$.

Definition 71 (The Kind CREATOR) *A creator is an agent that creates new assets. It is defined as follows.*

$$\begin{aligned}\text{CREATOR} \triangleq\ & \langle name = \texttt{Creator},\\ & uri = \texttt{kind://Foundation/Creator},\\ & parentkind = \text{UNIVERSAL},\\ & version = \texttt{0.0},\\ & creator = \texttt{kinirycs.caltech.edu},\\ & language = \texttt{English},\\ & definition = \texttt{"The creator of an asset, be it a person or a system."},\\ & obligation = \top,\\ & comment = \texttt{"Every asset has a creator."}\rangle\end{aligned}$$

The kind CREATOR is defined as having a parent kind of UNIVERSAL and nine substructures, each of which has a specific value. This kind is used to define which agent initially adds an asset to a context within $A_{\mathcal{T}}$.

Each asset also can have some natural language text associated with it, as seen above. Consequently, the language in which the text is written is an important property. As a result, a supplementary kind LANGUAGE is introduced.

Definition 72 (The Kind LANGUAGE) *The kind that represents a natural language is defined as follows.*

$$\begin{aligned}\text{LANGUAGE} \triangleq\ & \langle name = \texttt{Language},\\ & uri = \texttt{kind://Foundation/Language},\\ & parentkind = \text{UNIVERSAL},\\ & version = \texttt{0.0},\\ & creator = \texttt{kinirycs.caltech.edu},\\ & language = \texttt{English},\\ & definition = \texttt{"The spoken/written language of an asset."},\\ & obligation = \top,\\ & comment = \texttt{"Every asset has a language."}\rangle\end{aligned}$$

5.3.6 Universal Properties

The basic universal properties adopted are derived from the Dublin Core metadata standard [369]. They are as follows:

- $name : Kind \rightarrow String$ - The label assigned to the data element.
- $uri : Kind \rightarrow String$ - The unique identifier assigned to the data element.
- $parentkind : Kind \rightarrow Kind$ - Indicates the parent kind of this asset.
- $version : Kind \rightarrow Float$ - The version of the data element.
- $creator : Kind \rightarrow Creator$ - The agent authorized to register the data element; i.e., the person or system that originally registered the kind.
- $language : Kind \rightarrow Language$ - The language in which the data element is specified.
- $definition : Kind \rightarrow String$ - A statement that clearly represents the concept and essential nature of the data element.
- $obligation : Kind \rightarrow Boolean$ - Indicates if the data element is required to always or sometimes be present (contain a value) in an instance.
- $comment : Kind \rightarrow String$ - A remark concerning the application of the data element.

These core kinds are used to define the universal kind, the root of the kind hierarchy, UNIVERSAL.

Definition 73 (The Kind UNIVERSAL) *The realization of the kind* UNIVERSAL *includes parts that are exactly the functional kinds itemized above.*

Thus, each and every kind defined in the realization of the theory has a name, a URI, a version, etc. This core set of universal properties is used as a metadata schema within the reuse repository. This aspect is discussed in detail in Chapter 6.

This completes a formal description of kind theory. Over the next few chapters applications of kind theory to several problems within software engineering with reuse are considered.

Chapter 6

Software Engineering Examples

The initial domain for the application of kind theory is global, collaborative software engineering. The desire is to build tools and develop processes so that a developer can reap the benefits of the formalization.

This chapter considers several examples from the domain of software engineering. The intent is to show that kind theory provides an abstraction that augments and incorporates many existing tools and techniques and provides a foundation for the advancement of software engineering.

The point of this chapter is not to dig down to the details; many instances are provided without details on their corresponding kinds. Instead, it is hoped that the general flavor of the application of the theory and what it can help accomplish through a broad set of examples from the domain of software can be conveyed.

Some of this chapter uses a different notation than proceeding chapters. To emphasize the fact that this theory has been partially realized by an actual, working system, examples are presented using the notation of that system rather than the notation of kind theory.

This notation is fairly straightforward and still under development, so it is not detailed in full here. In general, as described briefly in Section 4.6, and in detail in Chapter 5, each core concept of kind theory is mapped directly to a structure of the algebra. That structure is abstracted with a terse syntax that inherits a great deal from the syntax of algebraic systems like OBJ3 and Maude. Any reader who has familiarity with those or similar systems should have no problem understanding the examples.

6.1 Programming Constructs

The primary types of assets which must be dealt with are those present in programming languages. Functions, modules, control structures, variables, etc. are the *lingua franca* of the software engineer.

One programming construct, the loop, is examined in detail here. This gives a feel for how programming constructs can be represented, manipulated, and reasoned about using kind theory.

6.1.1 Loops

Consider a loop in any standard programming language. Loops come in many syntactic forms. For example, in the C programming language there are three primary loop constructs: `for`, `while`, and `do`. Three loops using these constructs are shown in Example 14.

Example 14 (Three Equivalent C Loop Examples) *Consider the following three loops, each of which is written in the C programming language.*

```
for(int x = 0 ; x < 10 ; x++)
  f(x);
```

```
int x = 0;
while (x < 10) {
  f(x);
  x++;
}
```

```
int x = 0;
do {
  f(x);
  x++;
} while (x < 10);
```

The `f(x)` in the three examples is the same function.

Fact 1 (Equivalence of Loops) *The three loops shown in Example 14 are semantically equivalent.*

Fundamentally, all loop constructs are equivalent to each other at some abstraction level[1]; they are each syntactic variations on a common theme. That theme is specified by the kind LOOP.

Example 15 (Loop) *The kind* LOOP *is a part of the core* COMPUTATIONALSTRUCTURES *domain.*

```
kind Loop is-kind-of Universal .
  op start : -> Program .
  op increment : -> Program .
  op guard : -> Predicate .
  op body : -> Program .
endk
```

This specification defines a new kind called LOOP. It states that the parent kind of LOOP is UNIVERSAL. Four kinds are defined in a *part-of* relationship with LOOP. All four are functional kinds, as denoted by the prefix `op` which indicates that these kinds are implicit subkinds of

[1] Particularly if one is familiar with monads.

the kind OPERATION. Each operation's domain and codomain kinds is indicated by the infix -> operator. Thus, all four have empty domains (thus they are constants) and the indicated codomains.

Each of these contained kind are documented using the UNIVERSAL realization's DEFINITION kind (from Chapter 5) with natural language. This documentation (not shown here) states that `start` describes the start state of the loop, `increment` describes a state change on the loop index, `body` is the program executed on each loop iteration, and `guard` is a predicate tested on each iteration to determine if the loop terminates.

6.1.2 Formal Loops

Programs of a form other than `x++` are legitimate `increment` programs. Moving a pointer down a list, decrementing a monotonically decreasing counter, or shortening a string buffer are all legitimate `increment` realizations. The key distinction of `increment` is that it enforces forward progress for the loop. Such a notion is formalized in a refinement of the LOOP kind called FORMALLOOP.

To define a FORMALLOOP, a notion of loop specification is needed.

Definition 74 (LOOPSPECIFICATION) *Kinds similar to these are part of the* CORESPECIFICATION *and* THEORYOFCOMPUTATION *domains.*

```
kind LoopSpecification is-kind-of Specification .
  op variant : -> Function .
  op invariant : -> Predicate .
endk

kind FormalLoop is-kind-of Loop .
                is-kind-of LoopSpecification .
endk
```

LOOPSPECIFICATION kind have two constant parts, a `variant` function and an `invariant` predicate. FORMALLOOP is defined as a subkind of both LOOP and LOOPSPECIFICATION. All instances of FORMALLOOP have all the elements of both LOOP and LOOPSPECIFICATION (e.g., `start`, `increment`, etc.).

6.1.3 Correctness

Another viewpoint on an instance of FORMALLOOP is one of *correctness*. Is the loop correct, given its specification? Because kind theory is a full logical framework, notions of specifications and proofs and how they relate to one another can be represented so that uncomplex proofs, like those of loop correctness, are automatable.

Recall the discussion of mathematical structures in Section 5.2.1. It is important to remember that proof constructs are the original formal reusable entities. A lemma is a contextual, stored intermediate result in a proof tree; a corollary is a specialization of a theorem; a conjecture is a statement that has no proof in the current context.

Consider the following program that computes an integer division by two[2].

Example 16 (Correctness Proof Example: Original Program) *Here is a program that divides an integer number by two.*

```
program divideByTwo(N : Integer): Integer is
  -- Ed. note: The result of a function is stored in the variable
  -- "Result" which is automatically defined and kinded in context,
  -- much like the Result keyword in Eiffel.
  X : Integer
  X := N
  Result := 0
  while (1 < X) do
    X := X - 2
    Result := Result + 1
  end
end
```

Interpretation into kind resulting in the following.

Example 17 (Correctness Proof Example: Interpretation) *Each construct in the example is presented as a realization of a kind. The details of each of these kinds are not specified here; their use and definitions are clear from context. The instance names in this example are human-readable placeholders for what is otherwise uninterpretable URIs (recall the* `uri` *identifier of the realization of the* UNIVERSAL *kind from Section 5.3.6).*

```
instance variableX is-of-kind Variable is
  eq name = "X" .
  eq kind = Integer .
endi

instance variableResult is-of-kind Variable is
  eq name = "Result" .
  eq kind = Integer .
endi

instance loopGuard is-of-kind Guard is
  eq value = 1 < X .
endi

instance lineOne is-of-kind Program is
  eq value = X := X - 2
endi

instance lineTwo is-of-kind Program is
  eq value = Result := Result + 1
endi

instance loopBody is-of-kind Program is
  eq value = lineOne ; lineTwo
endi

instance loop is-of-kind Loop is
  eq init = variableX.value := variableN.value ;
```

[2]This example is based upon exercise 22 from [161].

```
            variableResult.value := 0 .
  eq increment = lineOne .
  eq guard = loopGuard .
  eq body = lineTwo .
endi

instance divideByTwoProgram is-of-kind Function[N : Integer]: Integer is
  eq value = variableX ; variableResult ; loop .
endi
```

Thus, the program has been represented with kind theory. This means several things:

1. There is sufficient information so that this particular specification can be executed. This is not a necessary condition, but happens to be the case for this example.
2. All the elements of this specification can be reused and refined in other kind. The instance called lineTwo is generalizable to an instance of some sort of INTEGERINCREMENT kind:

   ```
   kind IntegerIncrement is-of-kind Function[X : Integer]: Integer is
     op result = X + 1 .
   endk
   ```

   which are realized in any number of syntactic forms (e.g., `x = x + 1`, `x++`, `++x`, `x = 1 + x`, etc.).

To check program correctness, the program first needs a specification.

Example 18 (Correctness Proof Example: Specification of DIVIDEBYTWO) *This time a specification is written of the program directly into $A_{\mathcal{T}}$. A standard algebraic semantics is used for the program as in Goguen and Malcolm [161].*

```
instance preC is-of-kind Precondition[S : State] is
  eq pre = S [[N]] = n ; N >= 0 = true .
endi

instance postC is-of-kind Postcondition[S : State] is
  eq post = (S[[N]] is (S[[2 * Result + X]]) and (S[[0 <= X]]) and (S[[X <= 1]]) .
endi

instance inv1 is-of-kind LoopInvariant[S : State] is
  eq predicate = (S[[2 * Result + X = N]]) .
endi

instance var is-of-kind LoopVariantFunction is
  eq kind = Integer .
  eq variantfunction = X .
  eq bound = 1 .
endi
```

Now, a kind is needed that explains what proofs of correctness for loops are. Additionally, in this example, the correctness of the loop is the correctness of the program.

Example 19 (Correctness Proof Example: SIMPLELOOPPROOF) *To define a proof structure, the various well-known parts of the proof are described, interpreted with a full interpretation, then an automated proof with the algebraic realization is attempted, which happens to be a part of a logical framework and basic theorem proving system.*

```
kind SimpleLoopProof is-of-kind ProofStructure[L : FormalLoop]: ProofTree is
  -- first prove that invariant is an invariant
  eq invariantcheck = red invariant(S ; body) .
  -- next, prove that the loop is safe.
  eq safety = red pre(S) implies invariant(S ; init) and
                 invariant(S) and not(guard) implies post(S) and
                 invariantcheck .
  -- next, show that it has progress (i.e., variant functions is decreasing).
  eq progress = red variant(S ; body) < variant(S) .
  -- finally, show that it terminates.
  eq termination = red init ; guard ; variant .
  eq run = invariantcheck ; safety ; progress ; termination .
endk
```

Essentially, SIMPLELOOPPROOF explains how to rewrite a FORMALLOOP into the common structure of a formal proof of (total) correctness for loops. To execute this proof structure, the `run` equation is evaluated. This will result either in a proof tree (in the kind theory system, a rewrite execution sequence and a result of *true*), or a partial failure (which may or may not be useful).

The important point of this example is that with kind one can specify how to do transformations of these knowledge abstractions between different domains and then "execute" those transformations, if possible and appropriate, to obtain new knowledge.

6.1.4 Optimization

The next example examines a programmer's optimizations of loops. Consider the loop-based program from Example 16.

Example 20 (Correctness Proof Example: A Faster Divider) *If there is division operator, then the original program is radically simplified.*

```
program fastDivideByTwo(N : Integer): Integer is
  Result := N / 2
end

inst fastDivideByTwoProgram is-of-kind divideByTwoProgram is
```

```
  ...
endi
```

Fact 2 (Behavioral Equivalence of Dividers) FASTDIVIDEBYTWO *and* DIVIDEBYTWO *are behaviorally equivalent.*

Both instances fulfill the same behavioral specification since `fastDivideByTwoProgram is-of-kind divideByTwoProgram`, so how are the two differentiated? The critical difference between the two instances is their *computational complexity*.

The realization of kind theory has a module that provides an ontology on, and judgments about, computational complexity. Here is an excerpt necessary for this discussion:

Example 21 (Correctness Proof Example: The Computational Complexity Domain) *This domain provides kind used to describe and reason about the time and/or space complexities of algorithms.*

```
kind ComplexityMeasure is-of-kind Function[N : Integer], Measure is
  op measure : ComplexityPolynomial .
  op _>_ : ComplexityMeasure ComplexityMeasure -> Boolean .
  op _=_ : ComplexityMeasure ComplexityMeasure -> Boolean .
  op _is-of-complexity_ : Program ComplexityMeasure -> Boolean .
  ...
endk
```

Thus, each algorithm can be associated with a formal specification of its (polynomial) complexity. In particular:

Example 22 (Correctness Proof Example: Computational Complexity Specifics) *Complexity measures are realized for the divider examples.*

```
inst ConstantTime is-of-kind ComplexityMeasure is
  eq measure = O(1) .
endi

inst LinearTime is-of-kind ComplexityMeasure is
  eq measure = O(N) .
endi


...
  divideByTwoProgram is-of-complexity LinearTime .
  fastDivideByTwoProgram is-of-complexity ConstantTime .
...
  --- compare two programs
  red divideByTwoProgram > fastDivideByTwoProgram .
  ==> false.
  red divideByTwoProgram < fastDivideByTwoProgram .
  ==> true.
```

Thus, algorithms that are functionally/behaviorally identical can be compared with other metrics. In particular, the results of a realization search for an instance that matches a particular kind composition can be ordered by computational complexity. Section 6.4.5 goes into the use of such substructures for search in more detail.

6.2 Specifications

Formal languages are those that have a logical structure. Specification languages are formal languages that are used to *specify* properties of constructs within their domain. From a kind-centric metamathematical perspective, the *theory* of a formal language is its domain, a *model* of a specification is its realization.

In this section some of the classic specification constructs for software are examined while building toward an example using the Java programming language.

6.2.1 Documentation as Specification

Documentation, in this example, is a set of unstructured natural language terms. The only constraints that are put on documentation are (a) that it has content (it is of nonzero length) and (b) it is human readable.

A representation of a system with kind is a specification, and is thus a type of documentation. Such a formal and verbose set of information is too formal, detailed, and overwhelming for most users. A more distilled version of the data tuned to a particular audience is necessary.

Tools which have grown in a Literate Programming tradition [65, 218] are being used more today because of the rise of languages that have integrated LP-ish tools like Java, Python, and Perl. These tools transform documented source code into system documentation in a number of formats. Such a transformation is accomplished with a custom-built program that must parse annotated source code and transform the data into the appropriate output format.

These documentation transformations can also be expressed with kind theory using rewriting rules. Consider the following Eiffel-inspired example.

Example 23 (The Contract/Short Form) *This kind represents a method in the Eiffel* short form *(or* contract form *as it is now know [112]). A full implementation of a method is shortened to its specification, documentation, and method signature.*

```
kind ShortForm is-of-kind PartInterp[F : FormalMethod] is
  op textualize : F -> NaturalLanguageDocumentation .
  eq textualize(nil) = nil .
  eq newline = String.newline .
  eq textualize(F) = "/**" newline
```

```
                    F.summary newline
                    " * @pre " F.precondition.toString newline
                    " * @post " F.postcondition.toString newline
                    " **/" newline
                    signature newline .
endk
```

This functional kind is a partial interpretation that "forgets" the implementation of a formally specified method. The rewrite equation *textualize* is used to transform the specification into structured natural language documentation; in this example, a Javadoc-like header.

Representing asset transformations with kind theory has several benefits. First, a specification is provided of existing transformations. Reason can be performed about whether the implementations are correct, how they might interact, and other similar goals. Second, the specifications are executable within the kind system, using it as a form of formal prototyping. Finally, the results of transformation are new formal assets, so representing, manipulating, and reasoning about them as any other asset is possible.

6.2.2 Assertions

Assertions are predicates used as testing and debugging triggers. Assertions are defined by slightly augmenting the predicate kind.

```
kind Assertion is-kind-of Predicate .
  op failureAction : -> Action .
endk
```

Most systems that provide assertion support fail when the assertion is violated via a *halt* instruction of some kind, either as a process failure or an exception. Because systems realize the semantics of assertion failures differently, an explicit binding for this is needed in the ASSERTION kind.

Such variance not only occurs across languages or operating system, but is even occasionally available within a single package. The IDebug framework [206] provides several failure modes for assertions. Future work integrating IDebug and kind theory is discussed in Chapter 8.

6.2.3 Class Invariants

An invariant in the domain of OO languages is an assertion checkable during *stable states*. The definition of stable states varies among languages. A class invariant is an invariant that is bound to a class. The scope of the invariant is the scope of the class, and the checkable stable state is the entry and exit actions for the exposed access points of the class (e.g., public methods for Java classes).

The general structure of class invariants is captured by the following kind:

```
kind Invariant is-kind-of Assertion .
  -- empty
endk
```

```
kind ClassInvariant is-kind-of Invariant .
  op boundClass : -> Class .
endk
```

6.2.4 Pre/Post-Conditions

Pre-conditions and post-conditions are constructs at the core of the specification and testing of correct software. Originally gaining attention in Hoare logic [187, 358], they were applied for software construction rigorously by Knuth [215, 216, 217] and practically by Meyer [200, 265, 266].

The kind specification of these constructs is straightforward.

```
kind MethodSpecifier is SpecificationElement .
  op boundMethod : -> Method .
  op label : -> String .
  ...
endk

kind PreCondition is Assertion .
                  is MethodSpecifier .
endk

kind PostCondition is Assertion .
                   is MethodSpecifier .
endk
```

The standard failure action of a pre-condition or post-condition is a *halt* instruction often accompanied by some failure details. Since these assets are assertions, they can have any failure action. Alternatively, their kind can be refined on a per-language or per-system basis to reflect the native failure actions of their enclosing executable context.

6.2.5 Temporal Operators

Temporal operators and systems like UNITY [56] are also used because most software is concurrent and/or distributed. Many temporal logic variants exist in the literature; see [126, 178, 318] and especially [358] for good coverage. The focus here is on those operators that are used in UNITY.

To specify a basic temporal logic the following kind are necessary. The derived operators *leadsto* and *follows* are not specified as their complicated definitions would only obfuscate the intent of this material.

```
kind State is Multiset .

-- Fundamental Operators

kind Initially is Predicate .
  eq domain = Predicate * System .
endk

kind Next is Predicate .
  eq domain = Predicate * System * Predicate .
endk

-- This could be rewritten as:
-- kind Initially is Predicate[Predicate * System] .

kind Transient is Predicate .
  eq domain = Predicate * System .
endk

-- Derived Operators

kind Stable is Predicate .
  eq domain = Predicate * System .
endk

kind StableToNext is FullInterpretation .
  var p : Predicate .
  var x : System .
  eq domain = Stable .
  eq codomain = Next .
  eq stable.p.x = (p.next.p).x .
endk

kind Invariant is Predicate .
  var p : Predicate .
  var x : System .
  eq domain = Predicate * System .
  eq invariant.p.x = initially.p.x and stable.p.x .
endk

-- This could be rewritten this as:
-- kind StableToNext is FullInterpretation : Stable -> Next .

kind StableEquivNext is Equivalence .
  eq domain = Stable .
```

```
    eq codomain = Next .
    eq interpretation = StableToNext .
  endk
```

This specification highlights the primary ways that logical operators can be derived.

The kind INVARIANT is defined as a predicate, but its semantics are defined in terms of the INITIALLY and STABLE kind. Thus, in this context, there is no relationship between INVARIANT and INITIALLY and STABLE but for an inclusion relation. While this does not capture all of the structure that is inherent in INVARIANT, it suffices for most uses.

Note that this INVARIANT is different from the identically named kind that was discussed in Section 6.2.3, as logical contexts effectively induce namespaces on their assets.

STABLE, on the other hand, is defined as a predicate equivalent to NEXT under specific circumstances. Thus, STABLE has all of the information inherent in the INVARIANT example; i.e., the inclusion relation, the ability to evaluate, dependency, etc. The context containing STABLE has additional information about this kind, in particular, its full interpretation to NEXT.

This interpretation broadens the scope of reasoning in this context, better supports future canonicalizations of the temporal logic domain, and offers a foundation for cross-domain structural correspondence, perhaps between a temporal logic-based formalism like UNITY and a more general logical framework.

6.2.6 Program Code

Program code is another kind of specification from several different points of view. IDEs and compilers typically break down code with ASTs, thus have a fixed level of granularity. Kind-based representation of such constructs has no such fixed form, thus provides something akin to granularity *a la carte*.

Program code has the following characteristics:

- it is a *document* encoded in some format (e.g., ASCII). It has a core set of basic properties; e.g., it has an author, an owner, a creation date, a format, etc.
- it is a *specification* of information processing.
- it is a *set of instructions* which, in the proper context, can be executed. For some code, this context is an interpreter (and execution is immediate); for other code, some intermediate processing (compilation) must take place. The latter form can be viewed as a lazy-executable. Ergo, code is an *executable*, a *compilable*, an *interpretable*, or any combination of the three.

- it is a *piece of Intellectual Property*. The program may encode some knowledge that has inherent value or identity beyond the program code itself—i.e., it is the standard reference code for a patent or a trade secret.

To capture these aspects the following kind is defined.

```
kind Code is-kind-of Specification,
                     Executable,
                     Compilable,
                     Interpretable,
                     Property .
```

It is not explicitly stated that the kind CODE is of kind DOCUMENT since `is-kind-of` is a transitive operator and it was already specified that SPECIFICATION < DOCUMENT.

6.2.7 Specification Languages

Another documentation form is visual modeling languages. UML [90, 91, 92] and BON [364] are two variants. In these iconographic languages, programs are represented by diagrams. Different diagrams describe different aspects of the system—static structure, dynamic behavior, etc. Often, modeling tools permit the manipulation of the diagrams which, in turn, change the associated source code. Thus, in some of these visual representations, the diagram-code relationship is a bijection.

6.2.7.1 Comments on UML

Various tools exist in this domain, (e.g., Paradigm Plus [353], Rational ROSE, Together, etc.), but the vast majority of them have no theoretical foundation, nor do they generate data that is easily reused.

This state of affairs is result of the fact that most "populist" specification languages, UML in particular, have little or no formal semantics. Only with the slow adoption of OCL do UML-based specifications have any semantics [192, 366]. No formal relation can be constructed between design and implementation artifacts.

6.2.7.2 Extended BON

As a counterpoint to these interpretation-centric limitations of UML, new research is being conducted on specification languages with full reversibility and seamlessness. The BON specification language has been extended with a set of new constructs inherited from work in code standard specifications [205, 209]. This new language is called *Extended BON*. A binding lan-

guage is also being developed that lets a developer denote the interpretations between an Extended BON specification and Eiffel code.

The semantics of Extended BON and this binding language are specified in kind theory. Thus, because bindings are full interpretations, any change to specification is automatically reflected in the complementary code, and *vice versa*.

Extended BON is discussed in Section 7.1.11.2. This work is also available for download via the EBON[3] SourceForge project.

6.3 Programming Language Semantics

Programming languages are at the center of computation and software engineering with reuse. While the syntax of a language can be dealt with using kind theory (given the structure of most languages' grammars and their connection with freely generated algebras), representing the semantics of programming languages, on the other hand, is a difficult task in any formalism.

If the formal representation of the syntax and semantics of a small set of languages were available in kind theory, then automatic, derivable translation between those languages, similar to the hand-built Fortran to C (f2c) and Java to C compilers (e.g., Toba, Harissa, and j2c), are possible. Not only could such translation be automated, but the interpretations and realizations would be provably correct.

These sorts of transformations are also at the core of most of the fundamental problems in the compositional systems community. The construction of programs that are primarily built from components are often wholly centered on the glue code [296]. Were a unified formal representation of component language semantics available, in some cases, much of this glue code could be automatically generated (or, optimally, eliminated entirely) in a provably correct manner.

6.3.1 Java

Several tools have been written in Java that help build correct software [206, 211]. A kind representation is defined of a very small part of Java below. This Java kind domain will help formalize the current tools' semantics as well as show uses for interpretations at the programming language level. The focus is on the transformation of program specifications, written as source code annotations, into executable test code.

Assume the basic terminology of Java is defined with kind, e.g., classes, methods, documentation, expressions, variables, etc. The basic Java kind domain is augmented with language-

[3] http://ebon.sf.net/

independent constructs as defined in the code standard by Kiniry and Zimmerman [212]. Primarily, these extensions consist of new code properties with formal semantics. Examples include properties to denote constructs for concurrency semantics and others for the specification of side-effects. These extensions are called *semantic properties*, and are discussed in much more detail in Chapter 7. They are only mentioned briefly here.

Semantic properties are realized by Javadoc tags [199]. Java constructs are tagged with the original (`author`, `version`, etc.) and the new Javadoc tags. The original tags have a translation, from code to documentation, via documentation processing tools like Javadoc and Polardoc. This translation, from the syntax of Java comments to the syntax of HTML documents, is captured programmatically in these tools' code, but can also be captured formally in kind interpretations.

Each augmented tag has a specification in one of a number of formalisms. Additionally, each is realized by a Java implementation in the JPP tool [211]. This relationship between specification and implementation is not only for organization or documentation purposes; the goal is that the conformance of the implementation to the specification be checkable within the native formalism of the tag.

The challenge inherent in this goal is the interpretation of basic Java semantics into the native formalism of each tag. Since only enough structure for correct reasoning need be preserved in these interpretations, a full interpretation of Java semantics is unnecessary.

The secondary utility in defining an evolving, collaborative, formal kind domain for a programming language like Java is to facilitate the refinement of the libraries and language itself. The Java language specification is ambiguous [12]. Likewise, the core Java libraries are ill-specified and contain numerous errors due to the evolving nature of the language and the design of the language itself. Providing a kind representation of the core libraries would help designers and users understand and correct the libraries and control the evolution in a more well-founded and collaborative manner. This infrastructure is discussed in more detail in Chapter 8.

6.3.2 An Example Implementation

Consider the following piece of code directly copied from the IDebug package [206].

```
/**
 * <p> Returns a boolean indicating whether any debugging facilities are
 * turned off for a particular thread. </p>
 *
 * @concurrency GUARDED
 * @require (thread != null) Parameters must be valid.
 * @modify (QUERY)
```

```
 * @param thread we are checking the debugging condition of this thread.
 * @return a boolean indicating whether any debugging facilities are
 * turned off for the specified thread.
 * @review kiniry Are the isOff() methods necessary at all?
 **/

public synchronized boolean isOff(Thread thread)
{
  return (!isOn(thread));
}
```

This asset is an aggregation of the following assets:

- the program code P = "return (!isOn(thread));"
- the method definition

  ISOFF : METHOD = "public synchronized boolean isOff(Thread thread)"

- the documentation (call it DOC) “Returns a boolean indicating whether any debugging facilities are turned off for a particular thread.”
- the assertion (in the form of a precondition) A = "(thread != null)" which has default action and documentation “Parameters must be valid.” (a instance of DOCUMENTATION)
- and the explicit semantic properties, aggregated as the metadata set MD, comprised of
  - (concurrency, GUARDED)
  - (modifies, QUERY)
  - (param, thread, "we are checking the debugging condition of this thread.")
  - (return, "a boolean indicating whether any debugging facilities are turned off for the specified thread.")
  - (review, kiniry, Are the isOff() methods necessary at all?)

The aggregate comment header is the composition of the textual and tags-based documentation instances, $header = DOC \circ MD$. The method definition is the composition of its constituent elements, called its def: a signature, a name, and some metadata to capture keyword semantics. Finally, the whole is the composition of these two composite assets. Such a composition results in an instance specification such as

```
instance idebug.Debug.isOff is DOC composed-with MD .
```

which results in a resolved instance of

```
instance idebug.Debug.isOff is-kind-of FormalMethod .
  eq documentation = header .
  eq definition = def .
endi
```

The generic composition operator `composed-with` is resolved with kind matching to the only valid composition operator defined in context. In this example, the resolved is that which assigned each instance to the appropriate kinded field of the FORMALMETHOD kind. Finally, this composition is realized by concatenation when the instance is interpreted to ASCII program code. In an AST-based representation, the realization of composition would be tree grafting.

The JPP tool automatically translates these annotations into program code, as described in detail elsewhere [211]. Other tools provide similar features, though all have significant problems with specification conformance [119]. The plan is to construct a kind theory of knowledge for such specification systems by inference from the JPP realization and related tools. The work on the Extended BON tool suite also covers some of this territory. Chapter 8 discusses this future work in more detail.

6.3.3 The Lambda Calculus

The untyped lambda calculus has been realized in order sorted algebra by Goguen as the theory `LAMBDA` as a course exercise [154] and in rewriting logic in the Maude tutorial [71].

The basic kind specification of the untyped lambda calculus is as follows. A semantics of a typed lambda calculus is not stated since its model is still under intense investigation [26, 27, 28].

```
kind LambdaApplication is-kind-of Function : Expression -> Expression .
  -- Written as "F.A" or "F A" .
endk

kind LambdaAbstraction is-kind-of Function : Variable * Expression -> Expression .
  -- Written as "lambda x . M[x]" .
endk

kind LambdaCalculus is-kind-of Theory
                    is LambdaApplication composed-with LambdaAbstraction .
```

This kind domain (the LAMBDACALCULUS theory) can be mapped to an OSA realization of the untyped lambda calculus via the theory `LAMBDA` using a small set of straightforward interpretations:

```
kind LambdaCalculusToLambdaOBJ3 is-kind-of FullInterpretation : LambdaCalculus -> OBJ3 .
  -- Maps terms in LambdaCalculus to terms in the LAMBDA OBJ3 theory.
endk
```

Because (untyped) lambda calculi are inherently non-terminating, it seems more natural to specify their semantics in rewriting logic rather than a equational logic system like OBJ3. This work is also discussed by Stehr and Meseguer [347].

6.4 Asset Storage and Search

Exposing kind theory exclusively through a theorem proving interface is certainly not an end goal. Since the intent is to understand and foster collaborative open reuse, then constructing a technology to support such is a next step. The Jiki, as an open Web architecture, is a first step in that direction.

6.4.1 Background

An *open architecture* is an architecture that is designed and implemented to support modification, extension, and reconfiguration. Such reconfiguration might occur at install-time, use-time, or run-time. An *open Web architecture* is a Web-centric open architecture that supports either the modification of a Web of documents or the Web software itself.

Wiki[4] is an open Web architecture in the first sense; it permits users to manipulate a Web of documents with a standard Web browser.

The Jiki[5] is a distributed component-based Open Source[6] Wiki-ish server designed and built by members of the Distributed Coalition[7] (DC). It is an open Web architecture in both senses—it is a Wiki server that merges the Web and object systems object models to support arbitrary content types and reconfiguration.

6.4.2 Use of the Jiki for Open Collaborative Reuse

For the first several years of the Jiki's lifespan the software and Web site have been used to facilitate the interactions of several hundred individuals interested in the domain of *distributed objects*.

In 1996, several collaborators, most importantly Ron Resnick and Mark Baker, and I founded a mailing list called *DistObj*. The intent of this mailing list was to help the group of computer professionals interested in this then relatively new field that synthesized distributed computing and object-oriented software.

[4] http://c2.com/cgi/wiki?WikiWikiWeb
[5] http://www.jiki.org/
[6] http://www.opensource.org/
[7] http://www.distributedcoalition.org/

In 1998 at OOPSLA, several members of this list met during a Bird-of-Feather session and decided to implement the Jiki. The goals were threefold: (1) it was to provide a kernel codebase on which DistObj members could experiment with new tools and technologies related to distributed objects, (2) it was an exercise in the use of distributed objects itself, and (3) it would be used to house the knowledge of this community of researchers and developers. Thus, in a sense, the Jiki was a reflective experiment in software for both technical and social reasons.

Over the past four years, the members of this community have archived nearly 100 MB of knowledge in the form of mailing list discussions, Jiki Web pages, source code, documentation, specification, and more. This data is currently organized in a structured filesystem, and the intent is to import this data into a database when the new version of the Jiki is completed.

The Jiki source code was released in 1998. Over a dozen child projects have appeared on the net over the past few years, incorporating all kinds of new features and technologies into the core architecture. Many of the authors of these child projects have provided patches and updates to the architecture to support a host of new features. Several of these additions are incorporated into the new release of the Jiki which will be available in mid-2002.

All source code is documented using semantic properties. The Jass tool is used to automatically transform contracts into test code.

6.4.3 Architecture

The Jiki is a distributed component-based Web architecture. It is written in Java and its components are JavaBeans. Jiki components communicate with a variety of mechanisms including direct method invocation, remote method invocation, and HTTP.

Interface components are broken down into several categories: getters, editors, previewers, putters, and miscellaneous. Additionally, several subsystems, exposed as components or APIs, complete the architecture. These include subsystems for asynchronous events, exceptions, persistence, resources, translators, and utilities.

Assets stored within the repository are called *resources*. A resource represents either a static or a dynamic entity. Static entities are documents stored in a variety of formats. Dynamic entities are computational processes that expose interfaces via Web technologies like URLs, HTTP, etc. Examples of dynamic resources are servlets, CGI programs, Web services, and Jiki components.

Static assets are archived via a persistence subsystem. The basic persistence subsystem stores resources is a well-organized hierarchy of files within the filesystem. Alternative persistence systems have been written by contributors and include facilities that use various database products.

Static resources are encoded in a variety of formats. Basic documents are written in the *Jiki Data Format* (JDF), a presentation markup language similar to that which the original Wiki supports. The architecture also supports documents encoded in HTML, XHTML, XML, "classic" Wiki format, and others.

Translators are used to convert these various storage formats into formats that are viewable and editable via a Web interface. For example, a document encoded in the JDF, when viewed as a Web page, must be translated from JDF into HTML.

Because the Jiki is a Wiki, all resources are editable via a Web interface. The components listed above (getters, editors, etc.) are the primary means by which resources are managed. Getters read resources from the persistence system, putters store changed resources via the persistence system, etc.

The remaining subsystems are used for development purposes (e.g., exceptions, utilities, etc.) or are slated for incorporation in the new Jiki release (events).

6.4.4 Kind System Integration

The current architecture is not yet integrated with the kind system. The intent is to plug the kind system into it once the new version that supports the necessary features for this integration has been released.

These necessary features include the following: documents must be stored in a database for efficient search and manipulation, resources must be encoded in a single standard fashion so that component metadata is uniform, revision control of resources must be available for efficient context manipulation, and user authentication must be available so that agents can be identified and tracked.

With regards to the current kind system based upon T_{Γ} and $A_{\mathcal{T}}$, the intent is to model the Jiki's core resource schema on the Universal ground type using the universal properties discussed in Section 5.3.6. A collaborator named Zhiming Kuang implemented a prototype several years ago for a class project using IBM's XML4J API by writing a DTD modeled on Universal.

The "name" and "uri" substructures map directly to the name and URL elements of the resource API. The "parentkind" substructure is used in conjunction with "language" to track document encoding, and both are embedded in the universal resource schema. This means that document kinds map to schema-based markup types. The "version" substructure maps directly to the revision number of the new RCS back-end, thus interpretations between instances (resources) are realized by version tracking within RCS. All other universal elements are DTD substructures and are used for metadata and search purposes.

A user can add truth structures to their context through a feedback mechanism that exists on every Web page. This mechanism is similar to that which is planned in the KDE, as discussed in Section 6.5. Common domain-specific predicates used as truth structures are part of their appropriate kind domains. Sentences such as “I trust assets from this creator” and “I agree with this agent” are examples from the domain of reusable software.

For any given Web page, clicking on a specific “comment on this asset” page results in a new page that lists many of these standard statements. The user can state the truth value, surety, and evidence (via a URI) associated with each and every truth structure they care to add to their context. Figure 6.1 shows a mock-up of such a user interface.

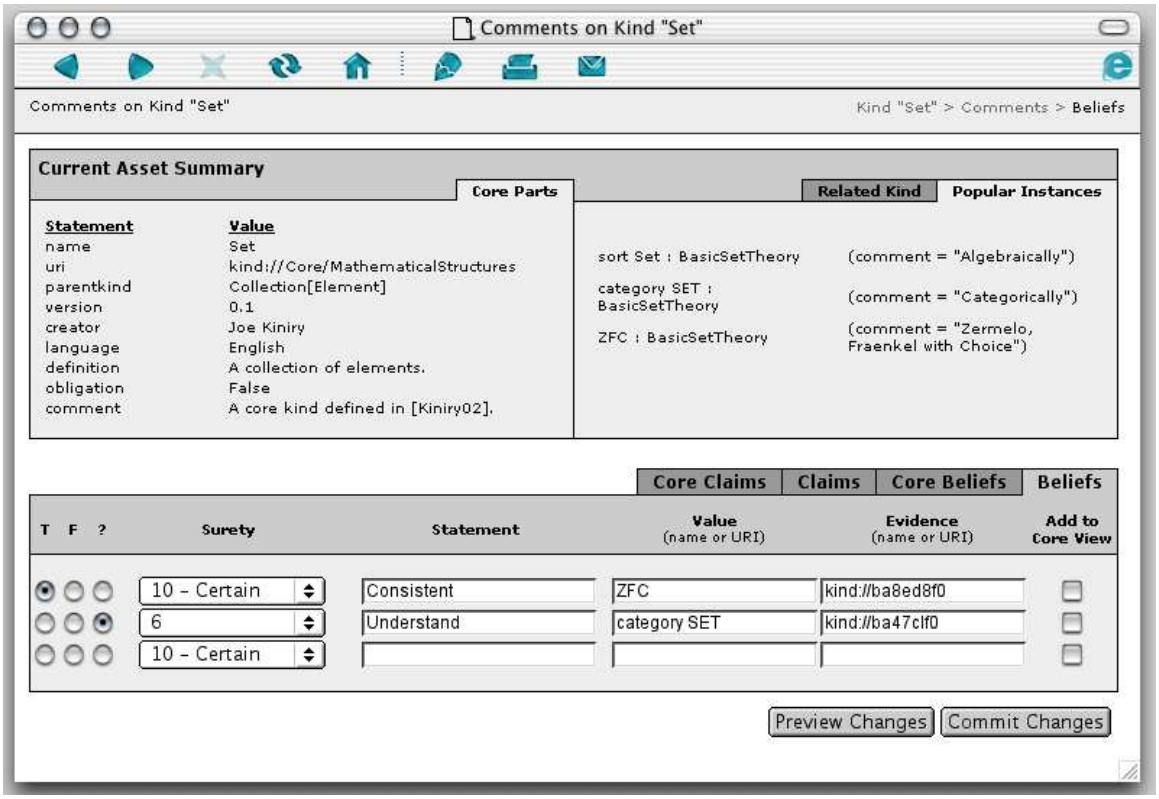


Figure 6.1: Stating Beliefs via a Web Interface

Likewise, to make claims about an asset, a similar page is used, as seen in Figure 6.2

6.4.5 Asset Search

Asset search is realized by resource search within the Jiki. A basic search mechanism is in place now with plans on a much more capable search system in the future.

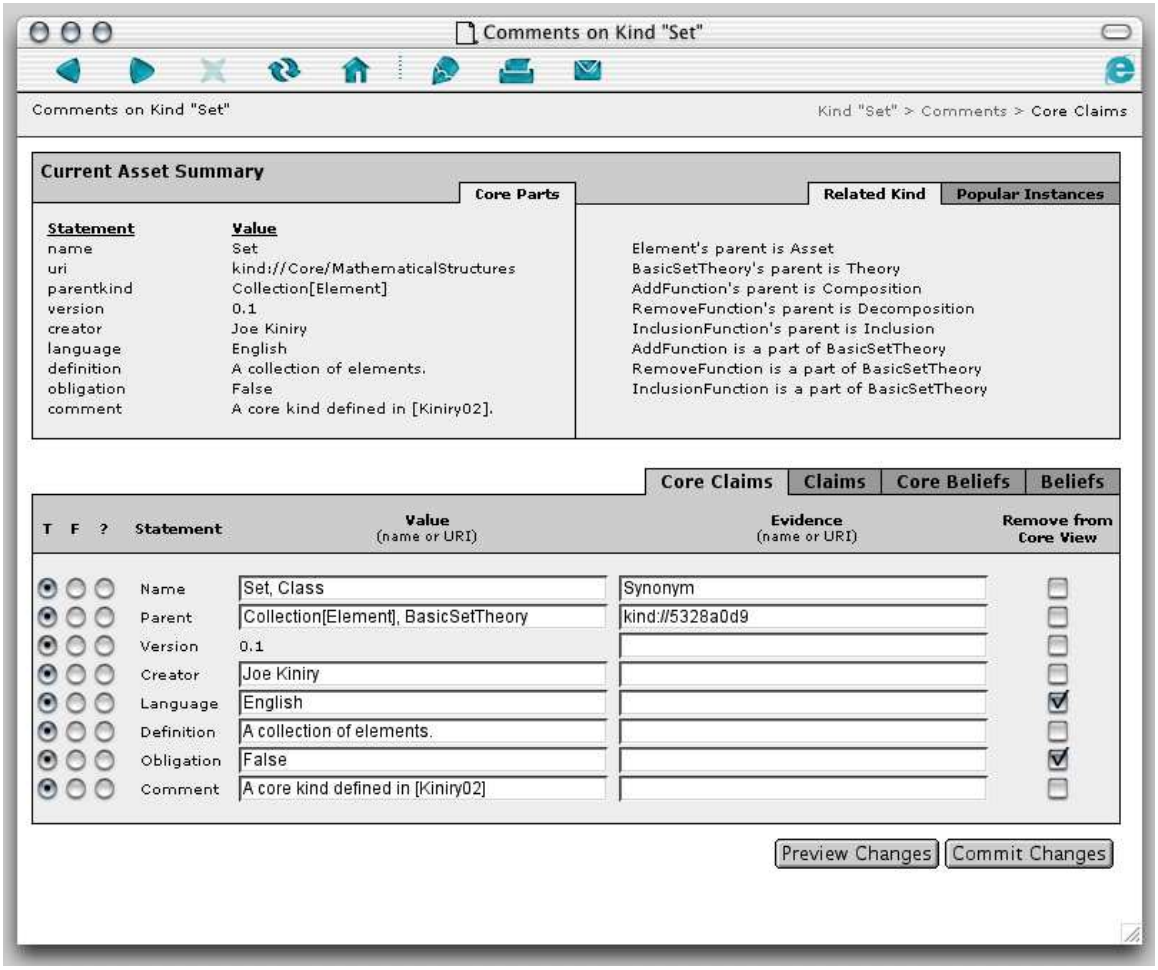


Figure 6.2: Stating Claims via a Web Interface

6.4.5.1 Current Interface

The current search interface is based upon htdig, an open source Web search indexing engine. Searches are entirely keyword-based and have little connection with the planned kind system integration.

This interface will be retained after integration because it is the most simple, efficient, first-line search technique for masses of structured or unstructured data.

Augmenting its search capability after kind system integration through the incorporation of a natural language domain for kind theory is intended, as discussed in Section 6.2.1. Such an addition would open up possibilities for synonym- and related topics-based searches.

6.4.5.2 Kind System Integration

Once integration is complete, searches using the kind system have two practical and theoretical interfaces.

Practical. Asset search is the primary operation of the Jiki. Thus, a search interface that is practical, intuitive, comprehensive, and efficient is needed, regardless of the underlying formalism representing assets and search's operational semantics. The most popular existing search engines have been examined to determine which feature(s) fit these criterion as well as the theory's capabilities.

Searches of the form "find pages like" look to be the most appropriate basic search semantics on which to base searches. Such an interface is provided via a single link on every page that triggers such a search. There is no need to expose any query language or theoretical interface.

Theoretical. On the theoretical side, there is a question of how to pose this search query.

If the foundational logic is first-order, then the query can be encoded in a classical judgment involving an existential quantifier on assets. That is to say, a typical query using a first-order (intuitionistic) quantifier would be of the form $\Gamma \vdash \exists A \mid A \; op \; U$ where op is the key operator of the query (perhaps *is-a* or *full equivalence*) and U is the partial specification being matched against (the key page in "find pages like").

This search query is a partial equivalence judgment. An agent A wishes to find all assets that match a partial specification S in a context Γ under an equivalence relation E. Normally such a problem is not only terrifically complex, but computationally undecidable. However, using kind theory the algorithm is decidable.

First, S is rewritten into canonical form given A, E, and Γ. While the context Γ is finite, S could possibly match a composition of assets that does not currently exist in Γ. Thus, the search

problem is necessarily non-terminating. Possible search heuristics are discussed in Chapter 8, but currently partial equivalences are searched for in a breadth-first fashion.

Each asset T in Γ is tested against S to determine if $S \lessdot_E T$. If the equivalence relation is valid then T is added to the result set. As discussed in earlier chapters, no specific algorithm is defined for ordering the result set. Obviously agent beliefs play an important role in such presentation, but it is beyond the current scope of this work to define such heuristics.

Since equivalence tests are structural comparisons, (recall rule Partial Equiv*), they are constant time operations. Thus, each composition-level pass searching for an asset that matches a partial specification is a linear operation on the size of the context.

The above base-case algorithm for the algebra and reflective model with equational logic can be specified, but the recursive compositional next steps cannot be because of the unbounded nature of the computation. Instead, rewriting logic is used to capture the operational semantics of such.

After the base pass through the context, each legal pairwise composition of assets is tested for partial equivalence. This means that all non-base case search steps are exponential in size, so search strategies smarter than a brute force technique must be derived in the future.

The infrastructure for such strategies is already in place in kind theory. It is hypothesized that the feedback cycle based upon community truth structures will provide a search scaffolding of commonly used compositions in the form of standard data structures, causally or associatively related constructs, and domain specific idioms. It is believed that this search scaffolding provides the kernel for deeper compositional searches and will focus attention on very specific compositional subdomains for knowledge search and evolution. Searches utilizing this community knowledge in the form of residual truth structures as tactics is called augmented existential judgments. Some of the future possibilities for such searches and the formal foundations for them are discussed in Chapter 8.

6.5 Knowledgeable Development Environments

To take initial steps toward the vision of the future of software development, the incorporation of a kind system into today's software development environments is under investigation. Some ideas about these environments are outlined here, making sure to detail how kind theory is used.

These ideas are an extension of the early work in knowledge-based representations in advanced IDEs, best represented by work like Rich and Walters on the Programmer's Apprentice [321].

6.5.1 A Usage Example

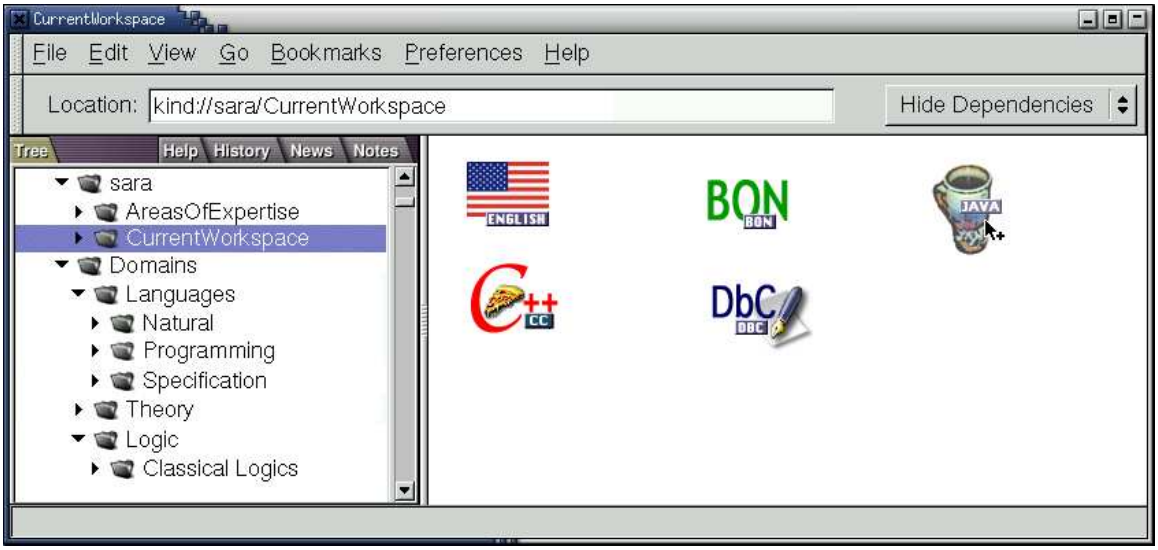


Figure 6.3: Building a Context

Consider a software engineer named Sara who is working on a banking project called *WizBank*. The first step that Sara takes as a casual user of a kind system is the identification of the relevant areas of knowledge with which she is familiar. Sara creates a new knowledge workspace (a kind context) and names it *WizBank Workspace*. This workspace represents her knowledge *context*. She then drag-and-drops a set of icons that represent *domains* from a domain toolbar onto her workspace. This action is realized theoretically via the inclusion of specific kind into a named context. See Figure 6.3 for a mock-up user interface that depicts such a scenario.

Possible domains of knowledge in the area of software engineering include specific programming language subdomains (syntax, semantics, common tools, etc.), design techniques and languages (such as the domain analysis methods of the SEI [16] and languages like Catalysis [103], respectively), and formal specification techniques like Design by Contract [200] (DbC) and temporal logic [126]. An example rendering of the domain covering programming languages is seen in Figure 6.4.

For the project that Sara is working on, the design documentation is written in BON [364] and English, and all the programming is done with Java and C++. Consequently, Sara chooses the appropriate kind domains to cover these four areas. Sara also decides that she is going to provide light formal specification of her classes with Design by Contract because she recently read OOSC [263]. Sara did not write any of the kind in these domains, they are provided to her automatically, accordingly she needs no knowledge of kind theory to use the kind system.

As discussed in Chapter 2, a *domain* is a named set of kind and a set of instances that

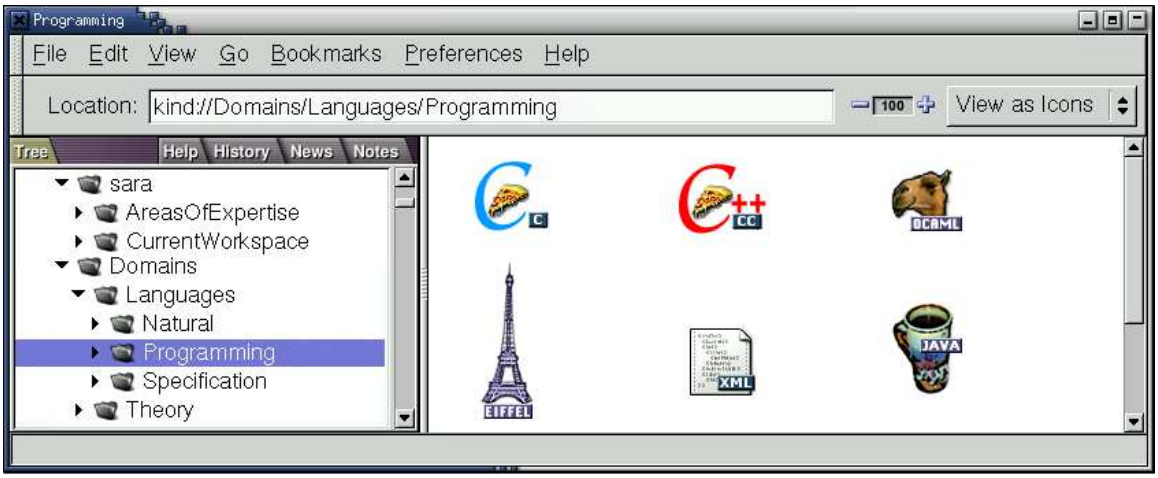


Figure 6.4: An Example Domain

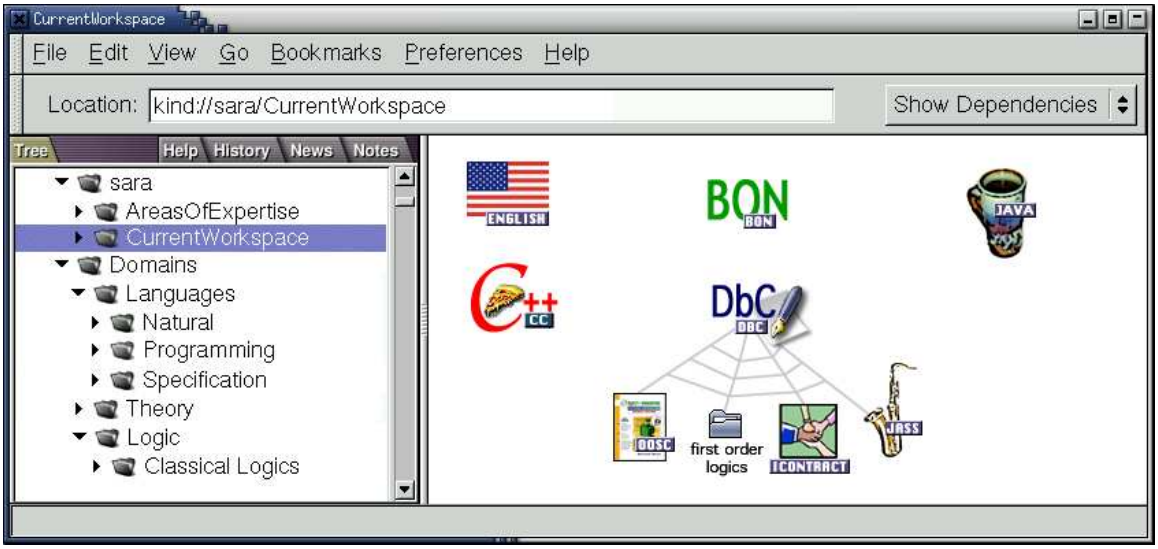


Figure 6.5: A Sketch of Automatic Inclusion

realize those kind. Thus, each of the domains that Sara has chosen contains numerous kinds that describe the primary concepts of the domain as well as references to kind instances (system tools and other resources, in this example) that realize these kind. For example, when she chose to put BON in her context, the system automatically made the various BON manipulation tools available to her, the Extended BON Tool-Suite for example. Likewise, when she chose to use DbC, the kind system automatically put the tools Jass [288] and iContract [221] in her context since they realize DbC for Java, as seen in Figure 6.5.

Additionally, domains have interdependencies that are realized by references between kind within different domains. Any kinds or domains that are necessary to realize a domain that has been explicitly chosen for Sara's context must be included as well. For example, DbC relies upon some basic first-order logic. Consequently, the domain of first-order logic is included in her context automatically.

Given that these additions were not directly added by Sara, they need not be exposed to her unless she asked for a "verbose" explanation of her context. This helps with knowledge and information hiding because otherwise her context could rapidly grow to an unmanageable state.

Each domain describes a comprehensive specific body of knowledge, but a user might not know everything in that domain. Likewise, Sara's level of familiarity will have a large range across different concepts. For example, most software engineers know the syntax of their favorite programming language, but in fact understand very little of its formal semantics. Therefore, Sara states to the system her general level of familiarity with each asset. Such data, which is characterized as a *belief* about the comprehension level of an *agent* (Sara) about an asset, is part of the context. Different levels of familiarity and dependency can be represented by color intensity or other similar techniques.

When Sara learned Java she was taught that Java interfaces are *public types*. Her instructor emphasized this fact so much that she does not even think of the word *interface* when she considers the concept. Thus, in Sara's context, the conceptual asset JAVAINTERFACE is renamed to PUBLICTYPE. This action is accomplished by renaming the kind, which is presented as if she renaming a file. Behind the scenes, a new kind PUBLICTYPE is defined as the target of a canonicality relation on JAVAINTERFACE in Sara's context.

6.5.2 Contexts

The kind system (realized in Maude) provides a standard read-eval-print loop. The primary interaction in such systems is through context operations, like those of Sections 2.10.1 and 2.10.2, and judgments on contexts. As a reminder, an example context operation is "add kind K to the

current kind context C". A typical query (judgment) is "does a construct I realize a kind K"?

In the realization $A_{\mathcal{T}}$, as discussed generically in Section 2.4.1, a kind context is a set of kind. The context is a working set of knowledge for a person or software client using the system. Such a set is identified by determining relevance to the current agent's domain of investigation.

Example 24 (Defining a Context) *For example, suppose Sara is programming servlets for a Java application server and is knowledgeable about Design by Contract (DbC), XML, and Design Patterns. Her context includes sets of kind that have previously been specified for several problem domains including the Java language, application server concepts, formal specification of software with DbC, the basics of XML, and basic notions from the Design Patterns community.*

Any useful realization of kind theory is one that contains a large number of these domain-specific kind ontologies. A part of this work has been identifying these domains and providing some basic ontologies that a community can extend. This point is discussed more in Chapter 8's discussion of future work.

As discussed in Chapter 2, the primary operations available on kind contexts include adding and removing kind to and from a context. Judgments include queries to see if a kind is in the current context, discovering kind that are not in the context but match a given (partial) specification, etc.

Example 25 (Defining a Context: Continued) *In this example, Sara tells the system, "I'm programming in Java today on an XML-based application server and I am knowledgeable of DbC and Patterns". This command is either in English or, more likely, is in the form of adding existing kind contexts (called "Java 1.2 Language Definition", "Design By Contract", "Basic XML", etc.) to her kind context.*

Turning back to the original running example, the LOOP kind and its brethren, the appropriate kind context is one that includes the loop-related kinds that have been specified and all the kinds on which they depend. This recursive dependency structure resolves into a fairly large and comprehensive set of kinds that rolls back to first principles, the ground kind of kind theory.

6.5.3 Consistency

A problem with contexts that brings their use immediately into question is one of consistency. A context is a logical model of a set of problem domains, but often different domains have conflicting ways of looking at a problem or perhaps have conflicting terminology. Thus, in building a kind context, the discovery of a state where the introduction of a new kind induces an inconsistency in the context is a frequent occurrence.

Conflict identification and resolution is a key feature of any realization of kind theory. When a new kind is added to the kind context, conflict detection and resolution algorithms detect these consistency problems. Recall that the fundamentals of these algorithms were already covered in Section 2.12. If the conflict cannot be automatically resolved, the user is shown the conflict and given a number of resolution options. The user then chooses one of the options or adds a new kind to resolve the conflict. The system responds appropriately, providing a new conflict-free kind context on which to build.

An example of an automatically resolvable kind conflict is one of name clashes that happen when using multiple domains. For example, consider a user who is working with the earlier mentioned DbC domain and Microsoft's Common Language Runtime (CLR) [89]. In CLR, Microsoft has redefined the word *contract*, a widely recognized term in the Computer Science community [184]. A *contract* in CLR is what the rest of the world calls a *signature* [265].

One way to resolve a name conflict in a kind context is to rename both CONTRACT kinds by using full module syntactic expansion. Thus, the CLR module's CONTRACT becomes CLR.CONTRACT and likewise, the DbC module's CONTRACT becomes DBC.CONTRACT.

Example 26 (Contract is Signature) *Resolving conflicts in a semantically valid, nonautomatic manner requires discipline and forward-thinking.*

```
domain Resolution is
  extends CLR .
  extends DbC .
  extends Core_OO_Notions .
  ...
  rename Signature to CLR.Contract .
  ...
enddom
```

Of course, the wise user sees the longer term, more subtle semantic issue with this conflict. Instead of using the automatic name rewrite to resolve the conflict at the domain level, she introduces a new relation that states that the CLR's notion of contract is *semantically* equivalent to the kind SIGNATURE. This is accomplished using the canonicality relation and defining two new interpretations. This pair of interpretations realizes a full interpretation bijection between `CLR.Contract` and `Signature`. Such a commonly occurring relationship can be expressed with visual metaphors, so as not to expose the theory to the user.

The manner in which various assets central to software engineering are represented, manipulated, and reasoned about using kind theory has been reviewed. The ability to deal with this range of assets is evidence of the breadth of application of kind theory to existing software

engineering problems. As a counterpoint, in the next chapter one way in which kind theory can extend the field of software engineering with reusable components in several new ways is examined.

Chapter 7

Semantic Properties, Components, and Composition

In this chapter several new concepts that are enabled through the application of kind theory to address problems in software engineering with reusable components are reviewed. The core ideas of this chapter are semantic properties, semantic components, and semantic composition[1].

7.1 Semantic Properties

Semantic properties are domain-specific specification constructs that augment an existing language with richer semantics. These properties are utilized in system analysis, design, implementation, testing, and maintenance via documentation and source-code analysis and transformation tools.

Semantic properties are themselves specified at two levels: with natural language, and formally using kind theory and other formalisms. The refinement relationships between these specification levels, as well as between a semantic property's use and its realization in program code via tools, is specified with kind theory.

7.1.1 Background

Ad hoc constructs and local conventions have been used to annotate program code since the invention of programming languages. The purpose of these annotations is to convey extra programmer knowledge to other system developers and future maintainers. These comments usually fall into that gray region between completely unstructured natural language and formal specification.

[1] Semantic properties are not directly related to the properties construct of Chapter 4.

Invariably, such program comments rapidly exhibit "bit rot". Over time, unless well maintained by documentation specialists, rigorous process, or other extra-mile development efforts, these comments become out-of-date. They are the focus for the common mantra: an incorrect comment is worse than no comment at all.

Recently, with the adoption and popularization of lightweight documentation tools in the literate programming tradition [214, 219], an ecology of semistructured comments is flourishing. The rapid adoption and popularity of Java primed interest in semistructured comment use via the Javadoc tool. Other similar code-to-documentation transformation tools have since followed in volume including Jakarta's Alexandria, Doxygen, and Apple's HeaderDoc. SourceForge[2] reports thirty-six projects with the term "Javadoc" in the project summary. FreshMeat[3] reports another thirty-five, with some overlap.

While most of these systems are significantly less complicated than Knuth's original CWEB, they share two key features.

First, they are easy to learn, since they necessitate only a small change in convention and process. Rather than forcing the programmer to learn a new language, tool, or imposing some other significant barrier to use, these tools reward the programmer for documenting her code.

Second, a culture of documentation is engendered. Prompted by the example of vendors like Sun, programmers enjoy the creation and use of the attractive automatically generated documentation in a Web page format. This documentation-centric style is only strengthened by the exhibitionist nature of the Web. Having the most complete documentation is now a point of pride in some Open Source projects.

The primary problem with these systems, and the documentation and code written using them, is that even semistructured comments have no semantics. Programmers are attempting to state (sometimes quite complex) knowledge, but are not given the language and tools with which to communicate this knowledge. And since the vast majority of developers are unwilling to learn a new, especially formal, language with which to convey such information, a happy-medium of informal formality must be found.

That compromise, the delicate balance between informality and formality, is exemplified in what are called semantic properties. Semantic properties demonstrate the lightest-weight aspect of the Knowledgeable Software Engineering program which is discussed in Chapter 8.

Semantic properties are domain-independent documentation constructs with intuitive formal semantics that are mapped into the semantic domain of their application. Semantic properties are used as if they were normal semistructured documentation. But, rather than being ignored by compilers and development environments as comments typically are, they have

[2] http://www.sf.net/
[3] http://www.freshmeat.net/

the attention of augmented versions of such tools. Semantic properties embed a tremendous amount of concise information wherever they are used without imposing the often insurmountable overhead seen in the introduction of new languages and formalisms for similar purposes.

7.1.2 Evolution and Current Use

The original inspiration for semantic properties came from three sources: the use of tags (e.g., `author` and `param`) in the Javadoc system, the use of annotations and pragmas in languages like Java and C for code transformation and guided compilation, and indexing clauses in Eiffel. All of these systems have a built-in property-value mechanism, one at the documentation level (Java) and one within the language syntax itself (Eiffel), that is used to specify semistructured information.

In Java, tags are the basic realization of semantic properties. They are used for documentation and formal specification, as discussed in more detail in Section 7.1.11.1. Tags are not part of the language specification; in fact, they are entirely ignored by all Java compilers.

Some annotations and pragmas are used by external, supplementary tools. An example of such are the formal tags used for some Design by Contract tools for Java (e.g., Jass [198]). These tools transform contracts in documentation into test code that is executed at run time. Similar constructs are realized in Eiffel with first-class keywords like *require*, *ensure*, and *invariant*.

Eiffel provides first-class support for properties via *indexing clauses*. An Eiffel class can contain arbitrary property-value pairs inside of its `indexing` block. This information is used by a variety of tools for source code search, organization, and documentation.

7.1.3 Documentation Semantics

Recently, Sun has started to extend the semantics of these basic properties with respect to language semantics, particularly with regards to inheritance. If a class C inherits from a parent class P, and P's method m has some documentation, but C's overridden or effective (in the case where P and/or m is *abstract*) version of m does not, then JDK 1.4's Javadoc *inherits* $C.m$'s documentation for $P.m$, generating the appropriate comments in Javadoc's output.

This explicit change in a tool's behavior is evidence of an implicit change in the semantics of the documentation. While straightforward and useful in this example, the meaning of such inheritance is undocumented and often unclear.

The situation in Eiffel happens to be more stable and less confusing. The semantics of properties, as realized by indexing clauses and formal program annotation via contracts, are defined in the language standard [264].

Even so, no mechanism exists in either system for extending these semistructured comments with new semantics beyond plug-ins for documentation (e.g., doclets in Java and translators in EiffelStudio). Additionally, the semantics of current annotations are entirely implicitly specified within a particular language or tool. No general purpose formalism exists to express their extra-model semantics, independent of the particular programming language with which they are used.

7.1.4 Properties and Their Classification

Properties were originally defined elsewhere[4] thirty-five semantic properties. All semantic properties are listed in Table 7.1. Their full informal specifications are found in Appendix B.

Table 7.1: The Full Set of Semantic Properties

Meta-Information:	**Contracts**	**Versioning**
author	ensure	version
bon	generate	deprecated
bug	invariant	since
copyright	modifies	**Documentation**
description	require	design
history	**Concurrency**	equivalent
license	concurrency	example
title	**Usage**	see
Dependencies	param	**Miscellaneous**
references	return	guard
use	exception	values
Inheritance	**Pending Work**	time-complexity
hides	idea	space-complexity
overrides	review	
	todo	

To derive the core set of semantic properties, the existing realizations that are used in two languages for many years were abstracted and unified. First, the set of predefined Javadoc tags, the standard Eiffel indexing clauses, and the set of basic formal specification constructs found to be useful were collected together. After that set was made self-consistent (that is, duplicates were removed, semantics were weakened across domains for the generalization, etc.), it was declared the core set of semantic property kind.

These properties were then classified according to their general use and intent. The classifications used are *meta-information*, *pending work*, *contracts*, *concurrency*, *usage information*, *versioning*, *inheritance*, *documentation*, *dependencies*, and *miscellaneous*. These classifications

[4]The original specification of these properties was in the Infospheres Java Coding Standard (http://www.infospheres.caltech.edu/). That standard has since been refined and broadened. More recent versions are available at KindSoftware (http://www.kindsoftware.com/).

are represented via kind theory's inheritance operators (see Section 7.1.8).

Many of these semantic properties are used solely for documentation purposes. The `title` property documents the title of the project with which a file is associated; the `description` property provides a brief summary of the contents of a file. These are called informal semantic properties.

Another set of properties are used for specifying non-programmatic semantics. By "non-programmatic" it is meant that the properties have semantics, but they are not, or cannot, be expressed in program code. For example, labeling a construct with a `copyright` or `license` property specifies a legal semantics. Tagging a method with a `bug` property specifies that the method has some erroneous behavior that is described in detail in an associated bug report. These properties are called semiformal because they have a semantics, but outside of the domain of software.

Finally, the remainder of the properties specify structure that is testable, checkable, or verifiable, either with static analysis or run-time testing. Basic examples of such properties are `require` and `ensure` tags for preconditions and postconditions, `modifies` tags for tracking side-effects, and the `concurrency` and `generate` tags for concurrency. These properties are called formal because they are realized by a formal semantics.

The KindSoftware coding standard [209] summarizes the evolving set of semantic properties. Each property has a syntax, a correct usage domain, and a natural language summary.

7.1.5 Context

Each property has a legal scope of use, called its context. Contexts are defined in a coarse, language-independent fashion using inclusion operators in kind theory. Contexts are comprised of *files*, *modules*, *features*, and *variables*.

Files are exactly that: data files in which program code resides. The scope of a file encompasses everything contained in that file.

A module is some large-scale program unit. Modules are realized by an explicit module- or class-like structure. Examples of modules are classes in object-oriented systems, modules in languages of the Modula and ML families, packages in the Ada lineage, etc. Other concepts used as modules include units, protocols, interfaces, etc.

Features are the entry point for computation. Features are often named, have parameters, and return values. Functions and procedures in structured languages are features, as are methods in object-oriented languages and functions in functional systems.

Finally, variables are program variables, attributes, constants, enumerations, etc. Because few languages enforce any access principles for variables, variables can vary in semantics con-

siderably.

Each property listed in the tables in Appendix B has a legal context. The context *All* means that the property can be used at the file, module, feature, or variable level. Additional contexts can be defined, extending the semantics of contexts for new programming language constructs as necessary.

7.1.6 Visibility

In programming languages that have a notion of visibility, a property's visibility is equivalent to the visibility of the context in which it is used, augmented by domain-specific visibility options expressed in kind theory.

Typical basic notions of visibility include *public*, *private*, *children* (for systems with inheritance), and *module* (e.g., Java's *package* visibility). More complex notions of visibility are exhibited by C++'s notion of *friend* and Eiffel's class-based feature scoping.

Explicit visibilities for semantic properties are used to refine the notion of specification visibility for organizational, social, and formal reasons. For example, a subgroup of a large development team might choose to expose some documentation for, and specification of, their work only to specific other groups for testing, political, or legal reasons. On the social front, new members of a team might not have yet learned specific tools or formalisms used in semantic properties, so using visibility to hide those properties will help avoid information overload.

Lastly, some formal specification, especially when viewed in conjunction with standard test strategies (e.g., whitebox, greybox, blackbox, unit testing, scenario-based testing), has distinct levels of visibility. For example, testing the postcondition of a private feature is only reasonable and permissible if the testing agent is responsible for that private feature.

7.1.7 Inheritance

Semantic properties also have a well-defined notion of property inheritance. Once again, new and complicated extra-language semantics are not to be forced upon the software engineer. Therefore, property inheritance semantics match those of the source language in which the properties are used. The earlier discussion of basic comments for Java methods (a feature property context) is an example of such property inheritance.

These kinds of inheritance come in two basic forms: replacement and augmentation.

The replacement form of inheritance means that the parent property is completely replaced by the child property. An example of such semantics are feature overriding in Java and the associated documentation semantics thereof.

Augmentation, on the other hand, means that the child's properties are a composition of all its parents' properties. These kinds of composition come in several forms. The most familiar is the standard substitution principle-based type semantics [245] in many object-oriented systems, and the Hoare logic/Dijkstra calculus-based semantics of contract refinement [266].

These formal notions are expressible using kind theory because it is embedded in a complete logical framework. For example, automatic reasoning about the legitimacy of specification refinement much like Findler and Felleisen discuss in [119] is possible.

Each of the aspects discussed (classification, context, visibility, and inheritance) are represented formally using kind theory.

The benefits of such a general formal semantics are numerous. Knowledgeable Development Environments (KDEs) can use the existence of semantic properties to help guide the development process (discussed in Section 6.5). Transformation tools can use the properties to generate test code (as already discussed in Section 6.3.1). Static analysis tools can check the validity of specifications against the program code.

7.1.8 Semantics of Semantic Properties

Because there are thirty-five semantic properties, and because the derivation of their formal specification is still underway, only representative examples of the semantics of semantic properties are detailed here.

7.1.8.1 Core Kinds

The kind SEMANTIC_PROPERTY captures all the basic structure of semantic properties. Several substructures are important to this discussion. A CONTEXT denotes the legitimate contexts in which a property can be used. DESCRIPTION contains a natural language description of the property (as in the tables in Appendix B). SYNTAX is a placeholder for the specification of the syntax of the property (dependent upon the current model in context).

7.1.8.2 Informal and Semiformal Properties

The only aspects of informal and semiformal properties that are represented with kind theory are their classification and structure.

For example, the *author* property has scope *module* and is classified as a piece of meta-information. Thus,

```
kind Meta-Information_Semantic_Property is-kind-of Semantic_Property .
```

```
instance Author is-kind-of Meta-Information_Semantic_Property is
  eq Context = "Module" .
  eq Syntax = "author" String [ Email ] .
endi
```

Interpretations of the *author* property are straightforward and purely syntactic. If a module in a specification has an *author* property, then the corresponding module in the implementation must have the author property as well.

Additionally, the domain of application might impose some small syntactic changes as well. For example, within Eiffel, an *author* property is specified in an indexing clause. The mandatory *String* and optional *Email* substructures must be enclosed within double quotes. In Java, on the other hand, neither of these substructures need be changed, but the keyword bound to the property needs an ‘’ prefix (e.g., “`author`”).

7.1.8.3 Formal Properties

Formal properties, on the other hand, can have significantly more complex model semantics. For example, the *concurrency* property specifies the concurrency semantics of a module or feature. Its extra-model semantics are, if a module is tagged as having a particular concurrency, then all features within that module are tagged with the same property.

But once realized within a particular implementation, the semantic burden grows considerably. For example, within the Java domain, the default semantics of all features (and consequently all modules) is *CONCURRENT*. That is to say, all methods are reentrant, and are supposed to be thread-safe, by default. Methods tagged with the `synchronized` keyword have *GUARDED* concurrency. All other semantics that can be specified with the *concurrency* property are not checkable by default and therefore can only be specified and checked via truth structures.

Thus, because of Java’s expressive concurrency semantics, the ability to validate a realization’s concurrency semantics against its specification is fairly straightforward.

In a language like Eiffel, on the other hand, the problem is not as straightforward. Eiffel has no default concurrency mechanisms, therefore any realization of concurrency semantics is going to be based upon the use of an external library. Thus, standard code patterns and practices are the only possible hooks for checkable semantics.

7.1.9 Process Changes with Semantic Properties

Rigorous use of semantic properties makes for more complete and correct documentation and program code. The first stage of incorporating their use is the explicit adoption of a coding standard that emphasizes semantic properties.

A set of conventions have been defined and used in several projects and courses. The following is an short distillation of the related coding standard, focusing on necessary, scoped semantic property use [209]. These conditions can be specified using kind theory as well, as discussed briefly in Section 7.1.10.

7.1.9.1 File Scope

Every file has a file scoped comment containing at least the following information:

1. The title of the project with which this module is associated using the `title` property.
2. A short description of the file's contents using the `description` property.
3. Any copyright information relevant to the project using the `copyright` property.
4. The license(s) under which the file is protected using the `license` property.
5. The primary authors of the file using the `author` property.
6. The RCS *$Revision$* tag (or similar) using a `version` property.

7.1.9.2 Module Scope

A module scoped comment describes the purpose of the module, guaranteed invariants, usage instructions, and/or usage examples. The beginning of the comment contains a description of the module in exactly one sentence using the `bon` property. Also included in the class comment are any reminders or disclaimers about required or desired improvements.

Each module comment block must include exactly one `version` property and at least one `author` property. The `author` properties often list everyone who has written more than a few lines of the class. Each `author` property should list exactly one author. The `version` property's version is a project-specific string (e.g., "1.0b5c7") followed by a RCS *$Date$* tag (or equivalent).

7.1.9.3 Feature Scope

Use natural language and local documentation conventions to describe the nature, purpose, specification, effects, algorithmic notes, usage instructions, reminders, etc. for each and every feature. The beginning of a method comment should be a summary of the method in exactly one sentence using the `bon` property. Every parameter, return value, and feature side-effects should also be documented with the appropriate semantic properties.

7.1.9.4 Variable Scope

Many variable-scoped semantic properties are implicit within the structure of the program code given the semantics of the programming language in use. So at the minimum, variables should be commented with those semantic properties that are not implicitly specified by program structure.

For example, the `hides` property, when used, is an explicit documentation of an implicit feature of the language. The most important semantic property to use on variables is the `values` property, each of which should also be documented as a class invariant.

7.1.10 Tool Support

These semantic properties have been used for the last five years. While an explicit adopted coding standard, positive feedback via tools and peers, and course grade and monetary rewards go a long way toward raising the bar for documentation and specification quality, these social aspects are not enough. Process does help, regular code reviews and pair programming in particular, but tool support is critical to maintaining quality specification coverage, completeness, and consistency.

7.1.10.1 Basic Tools

Templates were the first step taken. Raw documentation and code templates have been used in programming environments ranging from vi to Emacs to jEdit. But templates only help prime the process, they do not help maintain the content.

Code and comment completion also helps. Completion is the ability of an environment to use partial input to derive a (more lengthy) full input. An augmented version of completion in Emacs has been experimented with, for example.

Likewise, documentation lint checkers, particularly those embedded in development environments and documentation generators are also useful. Source text highlighting, as in font-lock mode in Emacs, is viewed as an extremely weak form of lint-checking. The error reports issued by Javadoc and its siblings are a stronger form of lint-checking and are quite useful for documentation coverage analysis, especially when a part of the regular build process. Finally, scripts integrated into a revision control system provide a quality firewall to a source code repository in much the same fashion.

More can and should be done. The approach taken here is to build and use what are called Knowledgeable Development Environments (KDEs), already discussed in Section 6.5. These development environments use knowledge representation and formal reasoning behind the scenes to help the user work smarter and not harder.

Work has begun on such environment. By extending powerful Emacs modes and tools that are part of a base development environment (e.g., XEmacs coupled with the object-oriented browser, hyperbole, JDE, semantic, and speedbar) with a kind system, it is hoped that the bar on development environments can be raised.

Current and future work on KDEs is discussed in Chapter 8.

7.1.11 Embedding Semantic Properties

When a semantic property is bound to a particular instance, for example, an `author` tag is used in some Java source code, what does this formally mean beyond questions of structural conformance? How do these semantic properties help guide the development process and exercise the system during testing? How do new tools take advantage of these properties?

First, the semantic properties must be embedded into the language in use. Second, domain-specific semantics using kind interpretations must be defined. Lastly, kind theory's belief truth structures are used to guide program development.

Syntactic embedding for two programming languages and one specification language will first be discussed. In the latter parts of this next section the other two points will be addressed.

7.1.11.1 Programming Languages

Semantic extensions have been used in two programming languages: Java and Eiffel.

Java. Semantic properties are embedded in Java code using Javadoc-style comments. This makes for an uncomplicated, parseable syntax, and the kind composition of semantic properties to constructs is realized by textual concatenation. The code in Section 6.3.2 shows their use.

Existing tools already use these properties for translating specifications, primarily in the form of contracts, into run-time test code. Reto Kramer's iContract [221], the University of Oldenburg's Semantic Group's Jass tool [198], Findler and Felliason's contract soundness checking tool [119], and Kiniry and Cheong's JPP [211] are several examples. Other tools use these properties for statically analysis; e.g., Leino's ESCJava [329], Jacob's LOOP [195], and the EBON tool.

Eiffel. In Eiffel, as mentioned earlier, indexing clauses are used as well as regularly structured comments to denote semantic properties. Using structured comments is necessary because the syntax of Eiffel dictates that indexing clauses only appear once, at the top of a source file.

The syntax of comments that use semantic properties is identical to indexing clauses. Accordingly, the same parser code can be used in both instances. The use of semantically meaningful comments is not new in Eiffel—feature blocks are often annotated with classifications via short comments.

An example of such use is as follows, directly from one of the Eiffel-based projects that uses semantic properties [208].

```
indexing
   description: "The Extended BON scanner."
   project:     "The Extended BON Tool Suite"
   author:      "Joseph R. Kiniry <kiniry@acm.org>"
   copyright:   "Copyright (C) 2001 Joseph R. Kiniry"
   version:     "$Revision: 1.26 $"
   license:     "Eiffel Forum Freeware License v1"
```

7.1.11.2 Specification Languages

Semantic properties have also been used to extend the BON specification language.

BON. BON stands for the Business Object Notation. BON is described in whole in Walden and Nerson's *Seamless Object-Oriented Software Architecture* [364], extended from an earlier paper by Nerson [290].

Primary Aspects. BON is an unusual specification language in that it is seamless, reversible, and focuses on contracting. BON also has both a textual and a graphical form.

BON is seamless because it is designed to be used during all phases of program development. Multiple refinement levels (high-level design with charts, detailed design with types, and dynamism with scenarios and events), coupled with explicit refinement relationships between those levels, means that BON can be used all the way from domain analysis to unit and system testing and code maintenance.

Reversibility summarizes the weak but invertible nature of BON's semantics. By virtue of its design, every construct described in BON is fully realizable in program code. One can specify system structure, types, contracts, events, scenarios, and more. Each of these constructs can not only be interpreted into program code, but program code can be interpreted into BON. This makes BON unique insofar as, with proper tool support, a system specification need not become out-of-date if it is written in BON.

Finally, BON focuses on software contracts as a primary means of expressing system semantics. These contracts have exactly the same semantics as discussed earlier with regards to object-oriented models, because BON is an object-oriented specification language.

Limitations and Opportunities. Because BON is fully reversible, BON is a (logically and functionally) weak specification language. BON needs to be extended in a well-understood, structured fashion in such a way that it is more expressive, but still retains its original attractive properties. An answer to this need is Extended BON.

Extended BON. Extended BON (EBON) is a syntactic and semantic extension of BON that incorporates additional formal specification, in the form of semantic properties, into the BON language. Since Extended BON is a proper superset of BON, the suite of tools that work with EBON continue to work with BON specifications.

An Extended BON scanner, parser, documentation generator, and design model checker are currently being developed. This set of tools is meant to help software developers keep their documentation, design, and implementation in sync as well as help find and correct errors as early as possible in the process of creating software artifacts.

This work is only recently underway, so only the current directions will be outlined and the full EBON grammar in provided Appendix C.

Details on Use. Translations from BON to a source language and viceversa are to be represented by kind theory interpretations. This means that changes to either side of the translation can not only be translated, but can be checked for validity according to its (dual) model. This specification-code conformance (validity) checking is what is called design model checking, and it is discussed in more detail below. This terminology is used because the specification is the theory and the program code is the model, when viewed from the logical perspective.

A change in the source code that is part of an EBON interpretation image will automatically trigger a corresponding change in the EBON specification. Likewise, any change in the EBON specification will automatically trigger a corresponding change in the source code.

Some of these translations entail more than a transfer of information from a specification to a comment in the source code. For example, an `invariant` semantic tag is interpreted not only as documentation, but also as run-time test code. This interpretation is not detailed here, as it is outside the scope of this discussion. It follows the same lines as related tools that support contract-based assertions in Java and Eiffel mentioned elsewhere in this chapter.

Belief-Driven Development. Some BON extensions are nonreversible because they represent system aspects that are either very difficult or impossible to derive. For example, the `time-complexity` semantic property specifies the computational complexity of a feature. It is (rarely) possible to extract such information from an algorithm with automated tools. But the fact is, the algorithm author often knows her algorithm's complexity. Thus, stating the

complexity as part of the algorithm specification with a semantic property is easy and straightforward task.

Now the question arises: How do are such specifications that are not automatically checkable known to remain valid? This is where the earlier-mentioned belief truth structures of kind theory come into play.

When the programmer writes the original `time-complexity` semantic property for a feature, she is stating a belief about that feature. Beliefs in kind theory are autoepistemic (the representation of the programmer is part of the logical sentence encoding the belief), have an associated strength or surety metric, and include a set of evidence supporting the belief.

A number of techniques are used to ensure that old or out-of-date beliefs are rechecked. With regards to this example, a continued validity condition is defined as part of the evidence, which is machine checkable. Currently, if the program code or documentation to which the complexity metric belief is bound radically changes in size, or if the feature has a change in type, author, or other complexity-impacting specification (e.g., concurrency, space-complexity, etc.), then the validity condition is tripped and the developer is challenged to re-check and validate the belief, restarting the process.

Design Model Checking. Given a reversible specification language, comparing a specification to its implementation (and vice-versa) is what is called design model checking. As briefly mentioned previously, such specifications are used for more than design; they are used in requirements analysis, they drive unit testing, and they help during refactoring. This work is recently underway, so it is outlined here to give a flavor of the research.

Because a BON specification B is reversible, each substructure $S \subset_p B$ is reversible as well. A substructure B is bound to a realization R by specifying that $R binds-to B$. The operator *binds-to* is a realization of a refinement ($<_p$) operation, thus is accompanied by a partial interpretation from R to B, and a full interpretation from B to R. These interpretations are purely syntactic and structural.

Certain bindings are completely implicit by virtue of code standard rules, primarily naming conventions (e.g. a BON feature called f in a class c automatically binds to a Java method called f in a Java class called c). All nonreversible semantic properties are bound using truth structures, as already discussed above.

As either the specification or the implementation changes, those changes, realized by interpretations between kind realizations, are automatically interpreted into their adjoint domains (code to specification and viceversa). Therefore, any modification to a specification is automatically reflected in its realization(s), and viceversa.

Examples of Use. BON and EBON have been used to specify four software systems. These systems include a component architecture called the connector framework (discussed briefly in Section 8.2.4), a multiresolution mesh library, a monitoring system called IDebug that has been mentioned several times, and the architecture of the primary product a company (DALi, Inc.). Students have also used BON extensively in the Distributed Computation Laboratory (CS141[5]) course that taught in 2002. Lastly, a full EBON specification of the Jiki is also being written which will be released as part of its new version.

There are several reasons for building this large library of specifications from so many different domains and authors. First, the more input files that are available, the better the EBON tool suite can be tested, particularly the lexers and parsers. Second, testing out the language in so many different domains exercises its informal completeness, where it might be lacking, what constructs are missing, etc. Getting so many different people to learn the language and apply it to different systems determine which constructs and concepts are most and least useful, which ones are the most complicated, etc.

Finally, the intent is that these specifications are used to drive both design model checking, as discussed in the last section, as well as to generate semantic component interfaces, as discussed in Section 7.2. The Jiki is the core testbed for such activities, given its multi-interface component communication modalities.

7.1.12 Experiences

Semantic properties have been used within the Compositional Computing research group, in the classroom, and in two corporate settings.

The Compositional Computing Group at Caltech has used semantic properties in its distributed and concurrent architectures, written in Java and Eiffel, over the past five years. Their utility has been witnessed by first-hand experience. The introduction of these technologies to new students and collaborators is particularly facilitated by semantic properties.

Students grumble at first when they are told that their comments now have a precise syntax and a semantics. The students initially think of this as being just more work on their part—yet another reason to hand in a late assignment. But, as the term goes by, the students incorporate the precise documentation with semantic properties and related tools into their development process. After a few weeks of indoctrination, they not only stop complaining, but start praising the process and tools. Higher quality systems are seen, and the students report spending less time on their homework than when they started the course.

These languages, process, and tools were also used in a corporate setting to develop a

[5] http://www.cs.caltech.edu/ cs141/

enormously complex, distributed, concurrent architecture. When showing the system to potential funders and collaborators, being able to present the system architecture and code with this level of specification and documentation invariably increased the value proposition. Uniformly, investors were not only shocked that a startup would design their system, but to think that lightweight formal methods were used to design, build, test, and document the system was absolutely unheard of.

Feedback from these three domains have been incorporated into the work. The set of semantic properties is still evolving, albeit at a rapidly decreasing rate. The tools see refinement for incorporation into new development processes, better error handling, and more complete and correct functionality. This user feedback is essential to understanding how these technologies and theory can be exposed in academic and industrial settings.

Now that semantic properties and their basic use have been outlined, their more advanced purpose can be reviewed—specifying kinds for semantic components and semantic composition.

7.2 Semantic Composition

Building complex software systems necessitates the use of component-based architectures. In theory, of the set of components needed for a design, only some small portion of them are custom; the rest are reused or refactored existing pieces of software. Unfortunately, this is an idealized situation. Just because two components should work together does not mean that they will work together.

The glue that holds components together is not solely technology. The contracts that bind systems together implicitly define more than their explicit type. These conceptual contracts describe essential aspects of extra-system semantics: e.g., object models, type systems, data representation, interface action semantics, legal and contractual obligations, and more.

Designers and developers spend inordinate amounts of time technologically duct-taping systems to fulfill these conceptual contracts because system-wide semantics have not been rigorously characterized or codified [109]. This section describes a formal characterization of the problem and discusses an initial implementation of the resulting theoretical system.

7.2.1 Background

Modern software systems are increasingly complicated. Combine new technologies, complex architectures, and incredible amounts of legacy code, and you have systems that no one can even partially comprehend. As complexity rises, code quality drops and system reliability suffers.

Building reliable complex systems necessitates the use of advanced software engineering practices and tools. But even with this bag of tricks, the modern software engineer is often overwhelmed with the technological problems inherent in building such systems. Most projects have to support multiple programming languages and models, deal with several different architectures (both hardware and software), have distributed and concurrent subsystems, and must be faster, cheaper, and better than their competitors.

What is worse, while these technological challenges are daunting, it is the nontechnological issues that are often insurmountable. The widespread lack of system, design, and component documentation, ignorance about testing and quality assurance, and lack of communication/knowledge across teams/companies, make for a very difficult working environment. The situation is further compounded by the social problems rampant in information technology: the Not-Invented-Here syndrome; issues of trust among companies, teams, and developers; developer education and motivation; and managerial and business pressures.

7.2.2 Semantic Components with Conceptual Contracts

Because these conceptual contracts cannot currently be specified, developers of reusable constructs (like chunks of code or paragraphs of documentation) can not say what they mean. There is a meaning-mismatch between what they want to build and what they do build. This information-impedance muddles communication between collaborators as well as complicates composition between components.

A new semantic theory, specifically designed to address this and related problems, can help solve this meaning-mismatch. If such a product were widely available and integrated into all aspects of system development, then systems would be easier to build, better documented, and more robust. Put more plainly, systems would be more correct.

The aims are to (1) provide a means by which components can be semantically (conceptually, formally) described, (2) present an algorithm which automatically determines when components can inter-operate regardless of syntax or component model, and (3) describe the means by which the adapters necessary to compose such components can be automatically generated.

A component that is specified, manipulated, and reasoned about using this (or functionally similar) theory, tools, and techniques is what is called a semantic component.

7.2.3 Semantic Compatibility

The proposal is that, instead of having only a syntactic interface to a component, a higher-level semantic specification is provided. Additionally, providing this semantic specification does not entail the need for a developer to learn any new heavyweight specification language, theory, or tools.

The key to this new solution is the notion of semantic compatibility [210]. Components are described explicitly and implicitly with lightweight, inline, domain-specific documentation extensions called semantic properties. These properties have a formal semantics that are specified in kind theory and are realized in specific programming languages and systems.

When used in tandem with a computational realization of kind theory (a kind system), these semantic components are composed automatically through the generation of glue code, and such compositions are formally validated.

7.2.4 Kinding with Semantic Properties

Semantic properties are used in kinding an instance (e.g., a method, a class, a type) in the following fashion. Before going into full detail with respect to the the related theory and the kinding algorithm, a detailed example will be reviewed to get the idea across. Its focus will be exclusively on kinding the Java method from Section 7.1.11.1.

The kind of this method contains much more information than its type. First, the kind contains everything discussed with respect to the method's signature. Additionally, a semantic interpretation of all of the documentation attached to the method is included. And, as a final element, a more complete representation of its type is also incorporated.

In more detail:

1. All the information that is inherent in the explicit specification of the method's signature and type are first composed.

   This particular instance is called Debug.isOff. With respect to classification, this construct is a Java method. This is encoded as

   Debug.isOff : JAVAMETHOD

The method's signature is interpreted as follows:

$$\text{Debug.isOff.ParameterSet} \subset_p \text{Debug.isOff}$$

$$\text{Debug.isOff.ReturnType} \subset_p \text{Debug.isOff}$$

where

$$\text{Debug.isOff.ParameterSet} : \textsc{JavaParameterSet}$$

$$\text{Debug.isOff.Parameter0} \subset_p \text{Debug.isOff.ParameterSet}$$

$$\text{Debug.isOff.Parameter0} : \textsc{JavaParameter}$$

$$\text{Debug.isOff.Parameter0Type} \subset_p \text{Debug.isOff.Parameter0}$$

$$\text{Debug.isOff.Parameter0Name} \subset_p \text{Debug.isOff.Parameter0}$$

$$\text{Debug.isOff.Parameter0Type} : \textsc{JavaType}$$

$$\text{Debug.isOff.Parameter0Type} \equiv \text{java.lang.Thread}$$

$$\text{Debug.isOff.Parameter0Name} : \textsc{JavaIdentifier}$$

$$\text{Debug.isOff.Parameter0Name} \equiv \text{thread}$$

$$\ldots \text{etc} \ldots$$

All of the type and signature information for the method is reflectively encoded in a kind theoretic context.

The structure of the related kinds like JavaParameter encode the necessary structure that must be provided by the interpretation, a type and a name in this case. Thus, this part of the process is no different than the parsing and typing that goes along with compiling the method.

2. All the supplementary information embedded via the use of the semantic properties and extra-type information is also interpreted into a kind theoretic context.

   For example, the fact that the method is declared as having `GUARDED` concurrency semantics is encoded with the concurrent kind:

$$\text{ConcurrencySemantics0} \subset_p \text{Debug.isOff}$$

$$\text{ConcurrencySemantics0} : \textsc{JavaConcurrencySemantics}$$

$$\text{ConcurrencySemantics0} \equiv \text{GuardedSemantics}$$

Likewise, the method's visibility, signature concurrency (use of the `synchronized` keyword), side effects (*modifies*), precondition, parameter and return value documentation, and meta-information (*review*) are also interpreted.

3. Finally, an interpretation to a more complete representation of the method's type is constructed by taking advantage of the domain semantics of refinement. In this example, stronger refinement semantics are attached via the use of the specified, explicit, classical contract.

   The existence of the precondition means that, after interpreting such, it is possible to: (a) Check that overridden methods properly weaken the precondition for behavioral subsumption. (b) Interpret such specification into run-time test code—here that entails a literal insertion of the code snippet into the proper place(s) in a rewritten version of the method. (c) Use this behavioral specification as a guard in semantic composition (see below).

The kind of this method is a formal best-effort at the specification of its semantics. Now, the rules of kind and instance composition come into play with respect to defining semantic compatibility.

7.2.5 Component Kind

Two instances (e.g., objects) are compatible if they can inter-operate in some fashion correctly and in a sound manner. For example, the two objects can perform their intended roles in an interactive manner and the composition of the two objects is as correct as the two objects when analyzed individually.

An object O is a realization of a semantic component, that is, O : SEMANTICCOMPONENT, if it provides sufficient information via semantic properties that a kind system can interpret its structure into a predefined kind. This parsing step is an interpretation to some K which is a semantic component.

A SEMANTICCOMPONENT is a kind that contains both a PROVIDES and a REQUIRES kind. Each of these enclosed kinds specifies a set of kind which the component exposes, or on which it depends, respectively. This two substructures are a generalization of the common *exports* and *imports* clauses of architecture description and component specification languages.

To determine the semantic compatibility of two instances, the following formal definition is used.

Definition 75 (Semantic Compatibility) *Let I and J be two semantic components.*

$$\Gamma, I : \textsc{SemanticComponent}, J : \textsc{SemanticComponent} \vdash \diamond$$

Furthermore, I is part of the provisions of an enclosing semantic component C_P, and J is part of the requirements of an enclosing semantic component C_R,

$$\Gamma \vdash, I \subset_p P, P : \textsc{Provides}, P \subset_p C_P,$$
$$J \subset_p R, R : \textsc{Requires}, R \subset_p C_R \vdash \diamond$$

*It is said that P and R are **semantically compatible** if an interpretation exists that will convert R into the ontology of P; if there is a $R \rightsquigarrow P$ in, or derivable from, the context Γ.*

*I and J are **semantically equivalent** if $I = J$.*

The existence of such an interpretation can be tested by checking to see if the canonical forms of P and R structurally contain each other.

Theorem 24 (Semantic Compatibility Check)

$$\Gamma, [R] \subset_p [P] \vdash R \rightsquigarrow P$$

The proof of this theorem is straightforward. If $[R] \subset_p [P]$ then $[P] \supset [R]$ (by definition of these operators). Ergo, by the rule (Partial Equiv*), $P \lessdot R$. By the definition of partial equivalence, when $P \lessdot R$, a full interpretation exists that takes P to R; that is, $P \rightsquigarrow R$. This full interpretation is exactly the conversion operator that is needed to guarantee semantic compatibility.

Finally, it is said that I and J can be semantically composed if: (a) they are semantically compatible, and (b) their composition is realizable within their instance domain.

This realization within their instance domain is the glue code on which composition depends. This construct is called a semantic bridge—a chain of equivalences between two instances that ensures their contextual base equivalency.

7.2.6 Examples

All the examples below are defined independently of source object language and ignore the subtle problems of class and type versioning that are solved in the full system and are not described here. These are only illustrative, not prescriptive, examples.

Finally, the tight coupling demonstrated below is theoretically equivalent to the more dynamic coupling found in loosely typed systems. The same rules and implications hold in such an architecture.

Some of the following examples will use the following type:

```
Type DateType
    method setDate(day: Integer;
                   month: Integer;
                   year: Integer);
    method getDate();
EndType
```

Before examining these examples, take note that if two classes are class or type compatible, they are clearly semantically compatible.

7.2.6.1 Standard Object Semantic Compatibility

```
Class Date
    method setDate(day: Integer;
                   month: Integer;
                   year: Integer);
    method getDate();
end;

Class SetDate
    callmethod writeDate(day: Integer;
                         month: Integer;
                         year: Integer);
    callmethod readDate();
end;
```

The methods tagged with the `callmethod` keyword are part of the REQUIRES of the class; those tagged with the `method` keyword are part of the PROVIDES part.

These classes are not type compatible since their outbound and inbound interfaces are of two different types (`DateType` and some new type; lets call it `AnotherDateType`).

If the only difference between the methods `setDate` and `writeDate` is exactly their syntax, then these classes are semantically compatible.

This mapping exists because both of these classes realize the same DATE kind. This fact is known because at some point a developer defined this ontology by virtue of a claim or a belief on the classes. Such a truth structure could have been defined explicitly through the specification of a semantic property (*realizes*) on the classes, via manipulation in the development environment, or by direct input with the kind system.

Now, because Date : DATE, then a mapping exists from each of its parts (e.g., the `setDate` method) to each of the parts of the kind (the canonical SETDATE kind feature).

These maps, when used in composition, define a renaming (akin to an alpha-renaming for instances) for the classes. Such a renaming is realized either by directly manipulating the source

text (if this is permissible in the current context), or by generating a wrapper class automatically. Thus, an adapter which maps calls from `writeDate` to `setDate` and from `readDate` to `getDate` will allow the composition of these two classes to perform correctly.

7.2.6.2 Extended Object Semantic Compatibility

The above example is based on a syntactic difference between two classes. Here is a more complicated example.

Consider the following two classes.

```
Class ISODate
    -- requires: year > 1970
    method setDate(year: Integer;
                   month: Integer;
                   day: Integer);
    method getDate(): ISODate;
EndClass

Class SetDate
    -- requires: year > 0
   callmethod setDate(day: Integer;
                      month: Integer;
                      year: Integer);
EndClass
```

To compose an instance of `SetDate` with an instance of `ISODate`: (a) the negotiation of the reordering of the parameters of the `setDate` method is necessary, and (b) the behavioral conformance of the composition must be checked.

This reordering is another form of the above alpha-renaming because the structural operators in kind theory ($\subset_p$ and $\supset$) are order-independent. The behavioral conformance is verified because $year > 1970 \Rightarrow year > 0$.

Thus, again the appropriate code can be automatically generated to guarantee semantic compositionality.

7.2.6.3 Ontological Semantic Compatibility

The final example is an example of a solution that would rely more strongly upon ontology-based semantic information encoded in kind theory.

Consider the following classes.

```
Class ISODate
    method setDate(year: Integer;
                   month: Integer;
```

```
                    day: Integer);
    method getDate(): ISODate;
EndClass

Class OffsetDate
    method setDate(days_since_jan1_1970:
                    Integer);
    method getDate(): OffsetDate;
EndClass
```

Assume that the parameter days_since_jan1_1970 was annotated with a reference to a kind that described what days_since_jan1_1970 meant. The context of such a description would necessarily have to have a ground—a common, base understanding that is universal.

In this case, the ground element is the notion of a day. The relationship between the parameter days_since_jan1_1970 and the day ground element need be established.

This relationship might be constructed any of a number of correct, equivalent manners. For example, a direct interpretation from the triple year/month/day to days_since_jan1_1970 would suffice.

Or perhaps a more complicated, multistage interpretation would be all that is available. For example, the composition of interpretations from year to month, then month to day, then day to days_since_jan1_1970, would provide enough information for the generation of a semantic bridge.

This composition of interpretations is automatically discovered and verified using the semantic compatibility theorem via the rewriting logic-based component search algorithm of kind theory.

7.2.7 Implementations

The earliest implementation of this research was done by a student (Roman Ginis) working with the author in 1998. The Boyer-Moore theorem prover (Nqthm) was used to hand-specify the above examples (and more) and prove semantic compositionality. Glue code was also written by hand to see what steps were necessary, in typical structured languages, for realizing the resulting interpretations.

The theoretical infrastructure has now been completed to completely describe the semantic components and reason about their composition. A realization of the theory in a kind system has been implemented in SRI's Maude logical framework [73]. Currently, the realization of renaming, reordering, and simple interpretations is direct: it is snippets of Java program code that are explicit realizations of the corresponding interpretation.

The next step is to lift these examples into a general context, automatically generating code in a variety of contexts, rather than using pre-written, parameterized glue code.

To this end, the Jiki [207] is being used as a testbed of semantic compositionality. Its components are distributed JavaBeans, and they interact via a number of technologies including local and remote method calls, message passing via HTTP and other protocols, and a tuple-based coordination mechanism based upon Jini's JavaSpaces.

These components have already been specified with both the Extended BON specification language as well as the semantic properties in their program code.

Thus, the next step is to use this application with an existing specification as a testing ground for automatic generation of glue code for a variety of interface modalities. The goal is for semantic components to be used as a kind of metacomponent architecture where glue code that utilizes a large variety of equivalent communication substrates is automatically generated both at compile and run-time.

Abstracting that component-based architecture, particularly with a view toward component quality-of-service, is already finished. The result of which is what is called the Connector Architecture, inspired by Allen and Garlan's similar work [10], and is discussed in Chapter 8.

Chapter 8

Conclusion

This is a large vision. Global, open, collaborative knowledge reuse is many years off, even for specific problem domains like software engineering with reusable components. This thesis outlined what is likely a ten-to-fifteen year research plan and, while this represents only the first few years of the work, it provides a reasonable basis on which to build. The next steps along this path in several directions have already been taken and many other research directions have been uncovered as a result.

8.1 Functions and Judgments for Knowledge Manipulation

Six primary aspects of knowledge and information are recognized by the knowledge engineering community. They are creation, publication, discovery, comprehension, composition, and collaboration. Because kind theory is a theory about software as knowledge, its coverage of these aspects should be evaluated.

8.1.1 For Creation

The categorization and organization of existing knowledge via any regular structure assists knowledge creation. Existing techniques, particularly static ontologies and facets, are clearly less flexible than the mutability of kind contexts with equivalencies, interpretations, and truth structures (claims and beliefs).

New information is based upon existing knowledge except in the most extraordinary of circumstances. Thus, the ability to search over existing knowledge is of key importance in the creation of new knowledge. The search capabilities enabled by the algebraic model of kind theory, particularly when brought to use via the Jiki, will provide a reasonable mechanism for knowledge search.

Creation seems to be primarily an iterative process. It is a process of fits and starts, search-

ing out a space of possibilities, refining an idea or solution until the creator is satisfied with the result. This iterative process can be realized theoretically by interpretation kind (refinement over time), and practically by the model of UNIVERSAL that includes notions of versioning. Likewise, the built-in capability for informal to formal semantics refinement via interpretations helps support the natural iterative process of knowledge.

With creation comes a notion of ownership as well. A creator often has a need or desire to associate himself with the assets that he has originated. Whether it is a matter of intellectual property, source attribution for trust graphs, or pride, the theoretical support via agents and truth structures, and the practical use of an `author` attribute within the realization of UNIVERSAL, cover the common needs of ownership.

Finally, as witness by the number of academic publications that are not single authored, knowledge creation is often a collaborative effort. The collaborative aspects of kind theory: explicit agents and contexts, semantics resolution via truth structures and interpretation, etc., show promise in fulfilling some of the requirements in collaborative software engineering.

8.1.2 For Publication

Publication of knowledge is assisted by several features of the theory and realization.

First, existing data can be characterized in enormous volume by integrated interpretation tools. This realization of kinds and instances supports the direct reference of existing data via the `URI` part of UNIVERSAL. Additionally, the necessary context of this knowledge capture (e.g., which version of a compiler a piece of source code works with) is completely characterizable by the explicit theoretical context associated with each instance.

Each new asset, whether written by hand or automatically interpreted, has full traceability of critical publication factor like authorship, revisions, copyright, etc. Traceability is enabled by properties and their semantics when bound to assets via the part-of function.

For example, associated with each asset within the Jiki are an *author* and a *permissions* semantic properties. The semantics of these properties are such that, if an asset has a read-only realization of the permissions kind, only the owner can manipulate the asset. This semantics is specified in kind theory within the architecture and is realized in the Java code. This is an example of binding a realization that is programmatic, rather than formal, to a kind.

Since this Web architecture also supports branching versioning through the use of CVS [338], refinement and history traces of an asset that are not necessarily linear are supported. The formal semantics of this design (i.e., refine the current loose, informal specification) are not yet specified. Additionally, the Jiki can automatically notify interested parties as changes are made in assets that have NOTIFICATION realizations. This functionality is reported to be quite

helpful for the publication problem in the open, dynamic environment that is the Internet.

Finally, publication is accomplished with the push of a button. For example, protocols like WebDAV [373] support nearly the same notions of ownership, revision, etc. that are used in the Jiki. Thus, a WebDAV-enabled Jiki, constructed via a kind composition of a WebDAV component suite [368] and the current Jiki, is another example of another compositional system constructed using kind theory.

8.1.3 For Discovery

Knowledge discovery is significantly centered on the problem of search.

Software repositories described in the literature (e.g., [41, 144, 172, 190, 242, 253, 278, 297, 323, 342, 384]) nearly uniformly support only three kinds of search mechanisms: string matching in comments, facets or other fixed classification schemas, and signature matching.

Each of these features can be handled using kind theory in the same manner as in [150], as order sorted algebras are a model for kind theory.

Other variants, like community-designed ontology searches [36], signature matching under permutation ([64, 259]), with labels, (as seen in OCaml's labeled mode [239] and Objective-C [227, 308]), etc. are straightforward as well through the use of equivalence and interpretation operators.

The possibilities for search with stronger partial specification and agent-enabled discovery seem to have promise. The root of these ideas comes from the various forms of behavioral specification [23, 301] and is similar to some recent unpublished work [337].

8.1.3.1 Partial Specification Matching

Assume that software stored as realizations in a kind system include specifications using Design by Contract or a similar discipline. A user, by providing a partial specification of a component that he is interested in, can perform a search using partial formal specification matching in addition to all the standard search methods mentioned above. Because specification refinement is behavioral refinement, searching using is-a as the query judgment will result in all assets that fulfill the partial specification.

For example, consider an Eiffel feature with the signature

```
Method m = Numeric foo(String bar)
Precondition p = bar /= Void and then bar.length > 0
p is-part-of m
```

In Eiffel, inheritance is realized by a PARENT specialization kind function because subtyping in Eiffel is behavioral conformance. Thus kind theory's notion of inheritance integrates

Eiffel's covariant type system and its in-language specification refinement mechanism based upon Hoare logic.

If any two assets in Eiffel are related by inheritance, e.g., classes via the `inherit` keyword or features via covariant specialization, then a partial interpretation exists between children and their parents. This interpretation widens types, strengthens preconditions, and weakens postconditions.

Thus, if the user searches with a partial method specification of

```
Method : m' = Integer methodName(ColoredString fieldname)
Precondition p' = fieldname /= Void
p' is-part-of m'
```

then the instance `m` is included in the result set because it conforms to this partial specification via the inheritance full interpretation because `Integer` interprets to `Numeric`, `ColoredString` interprets to `String`, the naming rewrites are trivial, and the precondition is strengthened.

8.1.3.2 Agent-Enabled Discovery

Agent-enabled discovery is often designed as discovery by-example. Research on software agents for such purposes is prolific [319]. This technology is even seeing use in research digital libraries, like that of the ACM, and commercial ventures like Amazon's "suggestions" functionality.

Kind theory and systems realize this type of pushed knowledge discovery by virtue of (a) its loose semantics for judgments in the theory, and (b) the rewriting logic semantics of the realization.

Judgments for discovery, primarily search, do not have any characterization at the kind level beyond correctness. Thus, a reasonable realization of a search operation has unbounded computation. This leads to the realization of such computation in rewriting logic using Maude. Because theories specified in rewriting logic need not be finite, unbounded search procedures can be initiated on behalf of an agent, as discussed in Section 6.4.5.

8.1.4 For Comprehension

Comprehending a new piece of knowledge seems to be the most difficult aspect of the five facets for the human being. This fact is exemplified by the standard software engineering mantra of "learning a new class takes one day of work." Several aspects of kind theory and its realization seem to assist in the comprehension of new knowledge, both by human and computational agents.

First, the context of each and every construct is explicit. To understand a piece of information, what it depends upon, how it was constructed, what it relates to are all important. All of that information is explicitly captured in kind theory contexts.

Likewise, comprehension is assisted by the traceability of assets in this realization. Each step that was taken by the creating or annotating agents is captured and can be searched, reviewed, etc.

Another powerful mechanism for comprehension is "learning by example". If a user wants to learn about a new asset, one way to present that information is to show analogues that are from domains the user claims to understand. A real-world example of this method are books with titles like "Java for C++ Programmers". Teaching new topics by example is a frequently used methodology with complex topics like foreign languages and programming languages. Such an information conveyance is a natural fit for kind theory's focus on realizations via interpretations of kind.

Recalling the discussion of refinement in Chapter 5, this framework naturally reflects the natural progression of learning, that of moving from low to high complexity as one learns more about a given topic. Refinements are subkinding relations that realize movement between precision and expression levels. For example, moving from a natural language to a pseudocode to a mathematical specification is such a refinement process. A user can explore these refinements to learn about a given asset, complementing their refined knowledge about the construct.

Finally, the community-oriented aspects of the theory and realization can assist in knowledge comprehension. The communal experiences of the classroom, messageboard, or chatroom are all conducive to learning new material, especially that which is very new and unfamiliar. The kind theory realization, especially the Jiki Web architecture, are a natural fit for such comprehension mechanisms.

8.1.5 For Composition

It is hypothesized that the set of compositions and interpretations relevant to the "typical set" of knowledge operations in a specific domain is relatively small. Thus, it is not unlikely that this set is characterized early in the lifecycle of a domain by its initial ontology builders. This means that most agents doing knowledge manipulation and creation will reuse existing compositions and interpretations the majority of the time. Relatively little work will go on at this level in any given domain since the bulk of operations are characterizing and interpreting new and existing assets.

Modifying an existing asset is accomplished through the creation of a new asset and the

introduction of an interpretation that translates the original asset to the new one. Often this interpretation is captured implicitly because most changes are superficial. An example of such superficial change are the differential changelogs of revision control tools like RCS and CVS.

There is a significant amount of controversy about the notion of composition [54, 57, 58, 60, 61, 118]. Some computer scientists even believe that composition is at the core of the key problems for the discipline. Regardless of the veracity of these positions, composition is defined in a relatively small number of ways.

For example, in a natural language, perhaps English, composition operators are primarily prepositions and conjuctions; the first for semantics and the second for syntax. The total number of compositional words, even in a language as large as English, is only in the dozens.

Likewise, composition in programming languages and systems construction comes in a relatively small number of forms. Several of these forms are characterized throughout in this chapter. The resulting set is surprisingly small, on the order of a dozen or so kind.

8.1.6 For Collaboration

The final aspect of knowledge theory realized in kind theory is collaboration. This work adopts and integrates the best of what is witnessed in various online and virtual communities [38, 99, 225], either in the theory or the realization.

Kind theory supports collaboration in several ways: the explicit notion of agent, the first-class and reflective nature of all constructs, and the ability to appeal to external formalisms and systems.

The belief and feedback constructs permit individual and group feedback cycles which contribute to collaboration [52, 55, 142]. Trust graphs, constructed via truth values about agents, focus and organize dynamic and emergent collaborations effectively [49, 316].

Additionally, traceability and refinement work toward a more living Web of information. Such is the attempt to model a knowledge dataset that is more like a scientific, community, or legal record than a database of facts.

Finally, the realization, with its Web-centric nature, support for versioning, ownership, history tracing, and concurrent access, make for a highly collaborative system.

8.2 Software

8.2.1 Semantic Properties

The next steps using semantic properties are on two fronts. First, embedding semantic properties in the Java-centric specification language JML looks like an interesting direction. Second,

developing new tools and technologies to realize knowledgeable development environments that use kind theory as a formal foundation is a major goal.

8.2.1.1 JML

JML is the Java Modeling Language [233]. JML is a Java- and model-centric language in the same tradition as Larch and VDM. JML is used to specify detailed semantic aspects of Java code and some tool support exists for type-checking and translating these specifications into documentation and run-time test code [40, 135, 312]. The formal semantics of JML have been partially specified via a logic as part of the LOOP project [195].

Extending JML with semantic properties would follow the same course that has been used for BON. Because semantic properties have already been integrated with Java, and given the existing tool support for JML, realizing interdomain interpretations that preserve a vast amount of information about JML-specified Java systems using kind theory should be relatively straightforward.

8.2.1.2 Social Implications

Knowledgeable development environments will have social implications for software development.

First, this challenging, interactive style imposed by knowledgeable development environments is not typical—introducing some kind of formal methods "paper clip" needs to be avoided. Thus, the environment needs to tune itself to the interactive style and development process of the user. Theoretically representing such styles in kind theory so that tuning is part of the logical context looks to be an interesting challenge.

Second, it has been noted that most developers are very uncomfortable starting from scratch, especially with regards to system documentation and informal and formal specifications. If some existing documentation or specification exists, developers are much more likely to continue in that trend because they feel that they are contributing rather than creating.

Because the EBON tool suite will automatically generate a base specification from program code, and because the specification-code validity conformance is automatically maintained, a primer as well as a positive feedback cycle now exists for lightweight specification with semantic properties. Only time and experience will tell whether this is a sufficient fire to light the correct software fuse.

8.2.2 Knowledgeable Environments

As mentioned previously, the work on KDEs continues. Augmenting the Emacs-based development environment in two ways is a first step.

First, integrating an interactive front-end like Emacs with the kind system realized in Maude. The availability of Emacs-centric tools for proof systems like Proof General[1] make this a relatively straightforward exercise. The most time-consuming aspect is writing interpretation engines that translate annotated source code to and from a kind representation format. Several such tools are being prototyped now [208, 211]. The second step is the integration of the KDE with the Jiki-based reusable knowledge repository, as already discussed. Finally, two new features that are planned for implementation within the KDE are *documentation inheritance* and *perspective*.

Eiffel development environments contain tools that provide what are called the *flat*, *short*, and *contract* views of a class. Flat forms show the flattened version of a class—all inherited features are flattened into a single viewpoint. The short form eliminates the implementation of all methods so that the developer can focus on a class's interface. The contract form is like the short form except the contracts of the class (feature preconditions and postcondition, and class invariants) are shown. These forms can be composed into *flat short* or *flat contract* forms which have the same meaning as in Eiffel tools.

Knowledgeable documentation inheritance is an extended version of such views. Rather than manually program the semantics of the "flattening" operation, the formal specification in kind theory automatically interprets the appropriate instances into a new form for rendering within the knowledgeable development environment. And because such interpretations are often fully reversible, the flattened forms can be edited and the changes will properly "percolate" to their original source locations.

Perspectives enable the user to specify which role(s) she is in while interacting with the kind-enabled system. Since kind theory is autoepistemic, the specification of a role (represented by an *agent* within the theory) permits automatic "filtering" of information according to, for example, visibility rules as discussed in Section 7.1.6. This user-centric filtering of information, much like narrowing modes within Emacs, helps the user focus on the problem at hand, ignoring all information that she either is not interested in, concerned with, or should not see.

These are only a few of the ideas for how to expose the user-centric aspects of kind theory via development environments, incorporating the use of semantic properties throughout.

[1] http://zermelo.dcs.ed.ac.uk/~proofgen/

8.2.3 Architecture

As mentioned previously, a basic component-based Wiki architecture has already been constructed. It is used as a foundation for an Open Source realization of a Web-enabled kind system. It has been used by nearly a dozen people and organizations as the basis for specialized Java-based Wikis in research, academia, and industry. Many of these projects have submitted their source code changes and integrating them into the next generation of the architecture is of high priority. The resulting architecture will support authentication (agent identification), several database back-ends (efficient kind manipulation), and full XML support in data representation (thus capturing the syntax of $A_{\mathcal{T}}$ in XML-based metadata). But several more pieces of functionality are necessary to make this architecture a very compelling kind system.

First, an integrated Web-based user interface needs to be developed so that the each facet of knowledge is well-represented. The interface sketches from Section 6.4.4 give some idea of the directions for this work. Integrating the recent work in algebraic semiotics [155, 152, 153, 254, 160, 163, 291] so that functional, reasonable, aesthetically pleasing interfaces can automatically be generated via interpretations from data to visual domains is also of interest.

Second, as mentioned earlier, augmenting the Jiki to provide a WebDAV [373] interface is important. This would permit the reading and writing of kinds and instances in an integrated fashion into the dozens of commercial and research tools that support this protocol [368].

Third, the architecture itself should be reflectively represented within the repository. This will represent the development process of the architecture as a knowledge domain and provide a workbench for experimentation in integrated knowledge and software. Coupled with WebDAV support, the reflective approach will duplicate and extend many of the successes of similar work [52] but with a formal foundation. A paper is being prepared that describes this system and the implications to maintenance, evolvability, and design [204].

Such a representation lends itself to another line of research that has already been started. By representing components as instances of kinds, an automated form of *semantic component composition* is enabled, as discussed in Section 7.2. Using semantic components, communication is completely abstracted from the communication substrate (e.g., method calls, messages, etc.). During program construction (static composition) and instantiation (dynamic composition) the kind system chooses specific communication mechanisms and automatically generates glue code that fulfills the communication requirements of the components based upon their specifications.

8.2.4 Specification of Components

Work has already begun in this domain on two fronts. First, as already discussed, a specification language has been developed as part of the EBON work mentioned earlier [208]. This language, derived from a semantically annotated code standard [209], captures many of the component-based system aspects necessary for such static and dynamic compositions.

There are several opportunities for improvements to the EBON language. For example, no provision is made for exceptional circumstances in BON (e.g., exceptions, signals, etc.). These and related weaknesses will be revealed only through additional use in a variety of domains. Such information will drive further refinements of the language.

Additionally, a component architecture has been designed based upon these ideas, initially targeting the Eiffel and Java languages. This work, called the *Connector Framework*, focused on connectors as first-class entities, inheriting from earlier work from Garlan and Shaw [10, 138, 331].

The specification language covers quality-of-service and temporal domains. The former supports system-wide performance and throughput guarantees [248] and provides the foundation for a more flexible software architecture. The latter provides a foundation for concurrent system analysis via specifications as seen in ESCJava [329], and run-time testing as used in JML [233], Jass [198], and iContract [221].

8.2.5 Reusable Software

A parsing system for an extension of the Java language has already been constructed. It is called JPP [211]. Now that the theory is complete, and after a basic kind system is implemented, finishing this work so that fully annotated Java source code can be parsed into a kind instance representation is a next step.

A benchmark for such a system is the full representation, preferably exposed via the Jiki, of the full Java class libraries. This system would permit the documentation and evolution of a collection of several thousand classes spread across dozens, if not hundreds, of packages. It would help the developer community provide more detailed, higher quality feedback to library (and other reusable code) authors.

A natural subproject for this work is a community annotation of the full core Java class library so that each Java class has a proper, full formal specification. Such a goal is a dream of many a frustrated Java developer, as witnessed by the bug reports on the JDC[2] Web site.

Given that the core of the EBON tool suite has already been implemented, another natural step along this path is the full representation of the core reuse libraries from some of

[2] http://developer.java.sun.com/

the more complete and correct programming languages including Eiffel [267], OCaml [239], Smalltalk [169]. By focusing on both non-mainstream languages concerned with software quality and populist languages like Java, it is hoped that research and industrial communities can be convinced of the benefits of using kind.

8.2.6 A Programming Language and an IDE

The final step in the construction of a kind system is the development of a new kind programming language. It is envisioned that such a language will be a purely modal declarative language, a new idea in itself. Such a language coupled with a complete set of reusable constructs would mean that the lines drawn between designers, programmers, architects, and testers would be erased.

This language would be the core of a kind-based integrated development environment that completely supported the integrated, immersive paradigm of kind. An implementation of such is under consideration in languages and environments that have strong tool support, particularly the languages mentioned above and the popular development environment Emacs.

Early integration with the current prototype kind system would be accomplished through the use of the new component-oriented design of Maude 2.0. Eventually it will be the case that the work transitions completely out of Maude and into a custom-built kind theory engine that realizes exactly the logic. This transition will simplify the system and make it more efficient in its computation and reasoning capability.

8.2.7 Integrating Knowledge from Non-Software Domains

Each problem domain needs an importing tool which converts information into instances in kind theory. Currently, such tools are hand-built, but integrating this type of functionality into existing tools that manipulate reusable constructs is straightforward. Examples include extending a compiler to archive program constructs during compilation, adding kind functionality to a Photoshop plugin to enable the user to store images and related data as kind instances as they work, or writing kind functions for Emacs to store realizations as an author writes.

Unsurprisingly, the vision is that, in the future, a kind theory interpreter sits within the system as a component, utilized by all tools that manipulate data artifacts. For example, as a programmer rewrites a Java method, it looks like he is changing characters, moving code blocks, implementing an optimized version of the same algorithm. But, in truth, behind the scenes he is manipulating instances of kind. Re-implementing an algorithm in a more efficient manner means that the same algorithm is now realized twice, each with an associated complexity specifications that denote the optimization when interpreted in context.

These ideas are by no means completely new. The Intentional Programming community has built technologies to view all structure as ASTs [335, 336]. Likewise, the automatic and assisted theorem proving communities have similar goals. Packages like X-Symbol and the Proof General provide interface technology to a variety of formal back-ends much in the same manner as is proposed for kind theory. Finally, the challenges inherent in the integration of so many semantic domains and the computational issues in extremely large data sizes as seen in the budding TeraGrid project are realized [37]. It is hoped that interaction with researchers on this or similar projects will help determine how kind theory can assist in making rigorous their knowledge engineering problems.

8.3 Theoretical Issues

No completeness or procedural soundness proof exists for *KT*. Both would have utility, but only after it becomes clear what completeness means in this context.

8.3.1 Alternative Models

Kind theory's loose semantics, especially for judgments, mean that several logics are likely models. The three that are most attractive for future investigation are fuzzy set theory [358], modal logic [180], and autoepistemic logic [279]. Working through the full details of each of these domains is necessary to understand: (a) if in fact these logics are legal models of the theory, (b) if the choice impacts the interface and usability of the kind system, and (c) if there are kind theoretical interpretations between the logics.

8.3.2 Type Theory

The work with type systems has prompted several research avenues. As mentioned earlier, fully understanding the connection between the notion of kind and that which is used in type theory is important for broad acceptance of the theory.

The close connection between type theory and algebras, both practically as seen in Section 4.6, and theoretically via a topoi bridge has been witnessed. Additionally, anecdotal evidence reveals that many researchers who construct and reason about type systems do not use any computational tool support to check and validate their work. A reasonable course of work then would be to provide a basic transformational theory and tool support for type theory practitioners and related mathematicians so that they can easily and with surety validate their work in a non-threatening (i.e., not a theorem prover) system.

8.3.3 Model Theory

Institutions are related to kind in several ways: a logic-independent model theory is some kind of structural dual to this work. Thus, understanding institutions and the role that they can play in this work is a highly important next theoretical step.

8.3.4 Categorical Work

There are numerous possibilities for formalizing kind theory using a categorical mathematics. Basic category theory can be used to characterize the core concepts. Functors can be used to formalize contexts and their operations. Finally, categorical type theory, higher order categorical logic, and the theory of institutions are all reasonable theoretical framework in which to reformulate kind theory in a more precise fashion. Perhaps by using these formalisms, there are further simplifications possible to the theory or model. It is also likely that categorical results will imply new kind theory results in a similar fashion.

8.4 Building Bridges

A secondary motivation for this work is the desire to assist in the advancement of knowledge. Fundamentally this means reducing duplicate work and improving communication among researchers by capturing knowledge as it is produced. The first key areas that are amendable to this method in this regard are those constituting the foundations of mathematics and computing. It is many observers' position that too many researchers in these areas become so highly specialized that they cannot communicate with their unknown siblings one research universe over. If only they had a bridge between their worlds, practically, philosophically, ontologically, and mathematically, then perhaps they would start to understand each other.

Appendix A

The Algebra $A_{\mathcal{J}}$ in Full Maude

The specification of $A_{\mathcal{J}}$ comes in three pieces: the prelude, the base, and the rules, all three of which are wrapped up in a top-level. Finally, a set of algebraic unit tests were used to exercise the algebra and check for correctness.

A.1 Top-Level

```
*** Formalization of the Theory of Kind's Type Theory
*** Joseph Kiniry
*** Started January 25, 2000
*** Maude version started April 17, 2000
*** $Id: appendices.tex,v 1.22 2002/06/06 20:16:26 kiniry Exp $

*** ==========================================================================

in prelude
in base
in rules
in tests
```

A.2 Prelude

```
*** Formalization of the Theory of Kind's Type Theory
*** Joseph Kiniry
*** Started January 25, 2000
*** Maude version started April 17, 2000
*** $Id: appendices.tex,v 1.22 2002/06/06 20:16:26 kiniry Exp $

*** Prelude to KTT.
*** Defines of several basic mathematical constructs useful in defining
*** KTT. Included are partial and total ordered sets, basic set theory, lists,
```

```
*** and two kind of tuples (pairs and triples).

*** ==========================================================================

(fth POSET is
  protecting BOOL .
  sort Elt .
  op _<_ : Elt Elt -> Bool .

  vars E1 E2 E3 : Elt .

  eq E1 < E1 = false .
  ceq E1 < E3 = true if E1 < E2 and E2 < E3 .
 endfth)

(fth TOSET is
  including POSET .

  vars X Y : Elt .

  ceq X < Y or Y < X = true if X =/= Y .
 endfth)

(fmod SET[X :: TRIV] is
  protecting MACHINE-INT .

  sorts Set[X] Set?[X] .
  subsorts Elt.X < Set[X] < Set?[X] .

  op err : -> Set?[X] [ctor] .
  op mt : -> Set[X] [ctor] .
  op __ : Set[X] Set[X] -> Set[X] [ctor assoc comm id: mt] .
  op _in_ : Elt.X Set[X] -> Bool .
  op _in_ : Set[X] Set[X] -> Bool .
  op delete : Elt.X Set[X] -> Set[X] .
  op |_| : Set[X] -> MachineInt .
  op _-_ : Set[X] Set[X] -> Set[X] . *** difference
  op _&_ : Set[X] Set[X] -> Set[X] [ctor assoc comm] . *** intersection

  *** exclusive or
  *** op _set-xor_ : Set[X] Set[X] -> Set[X] [prec 35 assoc comm id: mt] .

  vars E E' : Elt.X .   var S S' S'' : Set[X] .

  eq E E = E .
```

```
  eq E in mt = false .
  eq E in (E' S) = (E == E') or (E in S) .
  eq S in S' = ((S S') == S') .

  eq delete(E, mt) = mt .
  eq delete(E, E S) = delete(E, S) .
  ceq delete (E, E' S) = E' delete(E, S) if E =/= E' .

  eq S - mt = S .
  eq S - (E S') = delete(E, S) - S' .

  eq | mt | = 0 .
  eq | E S | = 1 + | delete(E, S) | .

  eq S & S' = ((S S') - (S - S')) - (S' - S) .
endfm)

(view Toset from TRIV to TOSET is
  sort Elt to Elt .
 endv)

(fmod LIST[X :: TRIV] is
  sort List[X] List?[X] .
  subsort Elt.X < List[X] < List?[X] .

  op er   : -> List?[X] [ctor] .
  op nil  : -> List[X] [ctor] .
  op __   : List[X] List[X] -> List[X] [ctor assoc id: nil] .
  op _in_ : Elt.X List[X] -> Bool .

  vars E E' : Elt.X .  var L : List[X] .

  eq E in nil = false .
  eq E in (E' L) = (E == E') or (E in L) .
 endfm)

(fmod PAIR[X :: TRIV, Y :: TRIV] is
  sorts Pair[X, Y] Pair?[X, Y] .
  subsort Pair[X, Y] < Pair?[X, Y] .

  op error : -> Pair?[X, Y] [ctor] .
  op <_;_> : Elt.X Elt.Y -> Pair[X, Y] [ctor] .
  op 1st : Pair[X, Y] -> Elt.X .
  op 2nd : Pair[X, Y] -> Elt.Y .
```

```
  var A : Elt.X .   var B : Elt.Y .

  eq 1st(< A ; B >) = A .
  eq 2nd(< A ; B >) = B .
 endfm)

(fmod TRIPLE[X :: TRIV, Y :: TRIV, Z :: TRIV] is
  sorts Triple[X, Y, Z] Triple?[X, Y, Z] .
  subsort Triple[X, Y, Z] < Triple?[X, Y, Z] .

  op error : -> Triple?[X, Y, Z] [ctor] .
  op <_;_;_> : Elt.X Elt.Y Elt.Z -> Triple[X, Y, Z] [ctor] .
  op 1st : Triple[X, Y, Z] -> Elt.X .
  op 2nd : Triple[X, Y, Z] -> Elt.Y .
  op 3rd : Triple[X, Y, Z] -> Elt.Z .

  var A : Elt.X .   var B : Elt.Y .   var C : Elt.Z .

  eq 1st(< A ; B ; C >) = A .
  eq 2nd(< A ; B ; C >) = B .
  eq 3rd(< A ; B ; C >) = C .
 endfm)
```

A.3 Base

```
*** Formalization of the Theory of Kind's Type Theory
*** Joseph Kiniry
*** Started January 25, 2000
*** Maude version started April 17, 2000
*** $Id: appendices.tex,v 1.22 2002/06/06 20:16:26 kiniry Exp $

*** Basic (core) constructs of KTT.
*** Defines all core constructs of KTT but none of the logic's rules.
*** Included are all grounds, slots, valueslots, types, instances, and
*** all context (type, instance, and environment).

*** ==========================================================================

(fmod GROUND is
  protecting BOOL .
  protecting MACHINE-INT .
  protecting QID .
```

```
  sorts Boolean String Integer Float Ground Ground?
        UniversalType TypeName InstanceName .
  subsorts Boolean String Integer Float < TypeName InstanceName
           < Ground < Ground? < UniversalType .

  *** Add subsort Integer < Float ?

  vars A B : MachineInt .

  *** Constructors for basic types mapped to sorts.
  op #_ : MachineInt -> Integer [ctor] .
  op $_ : Qid -> String [ctor] .
  op !_ : Bool -> Boolean [ctor] .
  op #_._._ : MachineInt MachineInt MachineInt -> Float [ctor] .

  op _<_ : Ground Ground -> Boolean .

  *** The valueless grounds .
  op =0= : -> Ground [ctor] .
  op -0- : -> TypeName [ctor] .
  op ~0~ : -> InstanceName [ctor] .
  *** The error ground.
  op ^G^ : -> Ground? [ctor] .
endfm)

(view VGround from TRIV to GROUND is
  sort Elt to Ground .
 endv)

(fmod GROUND-SET is
  protecting SET[VGround]
  * (sort Set[VGround]      to GroundSet,
     sort Set?[VGround]     to GroundSet?) .

  *** Return only the strings mentioned in a ground set.  Discard all
  *** Integers, Booleans, and Floats.
  op strings-only : GroundSet -> GroundSet .

  var GS : GroundSet .    var G : Ground .
  var I J K : MachineInt .  var Q : Qid .        var B : Bool .

  *** Base cases.
  *** Empty set.
  eq strings-only(mt) = mt .
  *** Singletons of basic types.
```

```
  eq strings-only(# I) = mt .
  eq strings-only($ Q) = $ Q .
  eq strings-only(! B) = mt .
  eq strings-only(# I . J . K) = mt .
  *** Recursive/inductive step.
  ceq strings-only(G GS) = strings-only(G) strings-only(GS)
                           if GS =/= mt .

 endfm)

(fmod GROUND-LIST is
  protecting LIST[VGround]
  * (sort List[VGround] to GroundList,
     sort List?[VGround] to GroundList?,
     op __ to _‘,_) .
 endfm)

(view VSlotType from TRIV to GROUND is
  sort Elt to TypeName .
 endv)

(view VProperty from TRIV to GROUND is
  sort Elt to Ground .
 endv)

(fmod SLOT is
  protecting PAIR[VSlotType, VProperty]
  * (sort Pair[VSlotType, VProperty]  to Slot,
     sort Pair?[VSlotType, VProperty] to Slot?,
     op 1st to type?,
     op 2nd to name?) .

  eq error = < -0- ; =0= > .
endfm)

(view VSlot from TRIV to SLOT is
  sort Elt to Slot .
 endv)

(view VValue from TRIV to GROUND-LIST is
  sort Elt to GroundList .
 endv)

(fmod VALUESLOT is
  protecting PAIR[VSlot, VValue]
```

```
  * (sort Pair[VSlot, VValue]  to ValueSlot,
     sort Pair?[VSlot, VValue] to ValueSlot?,
     **** op <_';_> to <_=_>,
     op 1st to slot?,
     op 2nd to value?) .

  eq error = < error ; =0= > .
endfm)

(view VTypeName from TRIV to GROUND is
  sort Elt to TypeName .
 endv)

(fmod TYPENAME-SET is
  protecting SET[VTypeName]
  * (sort Set[VTypeName]  to TypeNameSet,
     sort Set?[VTypeName] to TypeNameSet?) .
  protecting GROUND-SET .

  subsort TypeNameSet < GroundSet .
  subsort TypeNameSet? < GroundSet? .
 endfm)

(view VValueSlot from TRIV to VALUESLOT is
  sort Elt to ValueSlot .
 endv)

(fmod SLOT-SET is
  protecting SET[VSlot]
  * (sort Set[VSlot]  to SlotSet,
     sort Set?[VSlot] to SlotSet?) .
  protecting TYPENAME-SET .

  subsort Slot? < SlotSet? .

  *** Is a ground a type used in a slotset?
  op _type-in?_ : Ground SlotSet -> Bool .
  *** Is a ground a name used in a slotset?
  op _name-in?_ : Ground SlotSet -> Bool .
  *** Return the set of all types mentioned in a slotset.
  op types-of? : SlotSet -> TypeNameSet .

  var G : Ground .   var S : Slot .   var SS : SlotSet .

  eq G type-in? (mt).SlotSet = false .
```

```
  eq G type-in? (S SS) = (G == type?(S)) or (G type-in? SS) .

  eq G name-in? (mt).SlotSet = false .
  eq G name-in? (S SS) = (G == name?(S)) or (G name-in? SS) .

  eq types-of?(mt) = mt .
  eq types-of?(S) = type?(S) .
  eq types-of?(S SS) = type?(S) types-of?(SS) .
 endfm)

(fmod VALUESLOT-SET is
  protecting SET[VValueSlot]
  * (sort Set[VValueSlot]  to ValueSlotSet,
     sort Set?[VValueSlot] to ValueSlotSet?) .
  protecting SLOT-SET .

  subsort ValueSlot? < ValueSlotSet? .

  *** Is a ground a type used in a valueslotset?
  op _type-in?_ : Ground ValueSlotSet -> Bool .
  *** Is a ground a name used in a valueslotset?
  op _name-in?_ : Ground ValueSlotSet -> Bool .
  *** Return the set of all slots mentioned in a valueslotset.
  op slots-of? : ValueSlotSet -> SlotSet .
  *** Return the set of all values mentioned in a valueslotset.
  op values-of? : ValueSlotSet -> GroundSet .
  *** Converts a list into a set.
  op as-set : GroundList -> GroundSet .

  var S : Slot .        var SS : SlotSet .         var G : Ground .
  var VS : ValueSlot .  var VSS : ValueSlotSet .   var GL : GroundList .

  eq G type-in? (mt).ValueSlotSet = false .
  eq G type-in? (VS VSS) = (G == type?(slot?(VS))) or (G type-in? VSS) .

  eq G name-in? (mt).ValueSlotSet = false .
  eq G name-in? (VS VSS) = (G == name?(slot?(VS))) or (G name-in? VSS) .

  eq slots-of?(mt) = mt .
  eq slots-of?(VS) = slot?(VS) .
  eq slots-of?(VS VSS) = slot?(VS) slots-of?(VSS) .

  eq values-of?(mt) = mt .
  eq values-of?(VS) = as-set(value?(VS)) .
  eq values-of?(VS VSS) = as-set(value?(VS)) values-of?(VSS) .
```

```
  eq as-set(nil) = mt .
  eq as-set(G) = G .
  eq as-set(G ‘, GL) = G as-set(GL) .
 endfm)

(view VSlotSet from TRIV to SLOT-SET is
  sort Elt to SlotSet .
 endv)

(view VValueSlotSet from TRIV to VALUESLOT-SET is
  sort Elt to ValueSlotSet .
 endv)

(view VTypeNameSet from TRIV to TYPENAME-SET is
  sort Elt to TypeNameSet .
 endv)

(fmod TYPE is
  protecting TRIPLE[VTypeName, VTypeNameSet, VSlotSet]
  * (sort Triple[VTypeName, VTypeNameSet, VSlotSet]  to Type,
     sort Triple?[VTypeName, VTypeNameSet, VSlotSet] to Type?,
     op 1st to name?,
     op 2nd to type?,
     op 3rd to slots?) .

  ops Bool Int Float String Universal : -> Type .

  eq Bool      = < $ 'Bool      ; $ 'Universal ; mt > .
  eq Int       = < $ 'Int       ; $ 'Universal ; mt > .
  eq Float     = < $ 'Float     ; $ 'Universal ; mt > .
  eq String    = < $ 'String    ; $ 'Universal ; mt > .
  eq Universal = < $ 'Universal ; $ 'Universal ; mt > .
  eq error     = < -0-          ; mt           ; mt > .
 endfm)

(view VInstanceName from TRIV to GROUND is
  sort Elt to InstanceName .
 endv)

(fmod INSTANCENAME-SET is
  protecting SET[VInstanceName]
  * (sort Set[VInstanceName]  to InstanceNameSet,
     sort Set?[VInstanceName] to InstanceNameSet?) .
  protecting GROUND-SET .
```

```
  subsort InstanceNameSet < GroundSet .
  subsort InstanceNameSet? < GroundSet? .
 endfm)

(fmod INSTANCE is
  protecting TRIPLE[VInstanceName, VTypeName, VValueSlotSet]
  * (sort Triple[VInstanceName, VTypeName, VValueSlotSet]  to Instance,
     sort Triple?[VInstanceName, VTypeName, VValueSlotSet] to Instance?,
     op <_;_;_> to `[_;_;_`],
     op 1st to name?,
     op 2nd to type?,
     op 3rd to valueslots?) .

  ops Bool-False Bool-True String-Blank : -> Instance .

  eq Bool-False   = [ ! false  ; $ 'Bool   ; mt ] .
  eq Bool-True    = [ ! true   ; $ 'Bool   ; mt ] .
  eq String-Blank = [ $ 'blank ; $ 'String ; mt ] .
  eq error        = [ ~0~      ; -0-       ; mt ] .
 endfm)

(view VType from TRIV to TYPE is
  sort Elt to Type .
 endv)

(view VInstance from TRIV to INSTANCE is
  sort Elt to Instance .
 endv)

(fmod TYPECONTEXT is
  protecting SET[VType]
  * (sort Set[VType]  to TypeContext,
     sort Set?[VType] to TypeContext?) .
  protecting VALUESLOT-SET .

  subsort Type? < TypeContext? .

  op GammaT0 : -> TypeContext [ctor] .
  op Basic : -> TypeContext [ctor] .
  op _in_ : GroundSet TypeContext -> Bool .
  op get : TypeContext Ground -> Type .
  op types-in? : TypeContext -> TypeNameSet .

  var T T' : Type .   var TS : TypeNameSet .   var N : TypeName .
```

```
  var SS : SlotSet .  var Gt : TypeContext .   var G : Ground .
  var S : Slot .      var VSS : ValueSlotSet . var GS : GroundSet .

  eq Basic = Universal Bool Int Float String .
  eq GammaT0 = Basic .

  eq (mt).GroundSet in Gt = true .
  eq G in Gt = get(Gt, G) =/= err .
  eq G GS in Gt = (G in Gt) and (GS in Gt) .

  eq get(mt, G) = err .
  eq get(T Gt, G) = if (name?(T) == G) then T else get(Gt, G) fi .

  eq types-in?(mt) = mt .
  eq types-in?(T) = name?(T) .
  eq types-in?(T Gt) = name?(T) types-in?(Gt) .
 endfm)

(fmod INSTANCECONTEXT is
  protecting SET[VInstance]
  * (sort Set[VInstance]  to InstanceContext,
     sort Set?[VInstance] to InstanceContext?) .

  subsort Instance? < InstanceContext? .

  op GammaI0 : -> InstanceContext [ctor] .
  op _in_ : Ground InstanceContext -> Bool .
  op get : InstanceContext Ground -> Instance .
  op instances-in? : InstanceContext -> GroundSet .

  var G : Ground .         var I : Instance .     var T : TypeName .
  var VSS : ValueSlotSet . var N : InstanceName . var Gi : InstanceContext .

  eq GammaI0 = Bool-False Bool-True String-Blank .

  eq G in Gi = get(Gi, G) =/= err .

  eq get(mt, G) = err .
  eq get(I Gi, G) = if (name?(I) == G) then I else get(Gi, G) fi .

  eq instances-in?(mt) = mt .
  eq instances-in?(I) = name?(I) .
  eq instances-in?(I Gi) = name?(I) instances-in?(Gi) .
 endfm)
```

```
(fmod ENVIRONMENT is
  protecting TYPECONTEXT .
  protecting INSTANCECONTEXT .

  sorts Environment Environment? .
  subsorts Environment < Environment? .

  op err : -> Environment? [ctor] .
  op _#_ : TypeContext InstanceContext -> Environment [ctor] .
  op Gamma : -> Environment [ctor] .

  eq err = error # error .
  eq Gamma = GammaT0 # GammaI0 .
 endfm)
```

A.4 Rules

```
*** Formalization of the Theory of Kind's Type Theory
*** Joseph Kiniry
*** Started January 25, 2000
*** Maude version started April 17, 2000
*** $Id: appendices.tex,v 1.22 2002/06/06 20:16:26 kiniry Exp $

*** The rules of KTT.
*** All of the judgments of KTT are defined in this module.  Type rules
*** mentioned in the thesis are loosely cross-referenced here in comments.

*** ==========================================================================

(fmod KTT is
  protecting ENVIRONMENT .

  *** Could move these into module TYPECONTEXT.
  op  is-type? : TypeContext Ground -> Bool .
  ops is-subtype-of? is? is-part-of? : TypeContext Ground Ground -> Bool .
  op  type-of? : TypeContext GroundSet -> TypeNameSet .
  op  slots-of? : TypeContext TypeNameSet -> SlotSet .

  *** Likewise, could move these into module INSTANCECONTEXT.
  op  is-instance? : Environment Ground -> Bool .
  ops is-a-type-of? is? is-part-of? : Environment Ground Ground -> Bool .
  op  type-of? : Environment Ground -> TypeNameSet .

  *** Validity checking functions.
```

```
op  valid? : TypeContext TypeContext -> Bool .
op  valid? : Environment InstanceContext -> Bool .

*** Validity helper functions.
op validparents? : TypeContext Ground GroundSet -> Bool .
op validslots? : TypeContext Type -> Bool .
op validvalueslots? : TypeContext Ground ValueSlotSet -> Bool .

op add_to_ : TypeContext Environment -> Environment [ctor] .
op add_to_ : InstanceContext Environment -> Environment [ctor] .
op add_to_ : Environment Environment -> Environment [ctor] .

var Gt Gt' : TypeContext . var Gi Gi' : InstanceContext .
var E : Environment .      var G, G0, G1 : Ground .
var GS : GroundSet .       var S : Slot .
var I : Instance .         var SS : SlotSet .
var VSS : ValueSlotSet .   var N N' : TypeName .
var TS : TypeNameSet .     var T : Type .
var VS : ValueSlot .       var IN : InstanceName .

*** Done.
eq validparents?(Gt, G, GS) = not(G in GS) and (GS in Gt) and (Gt =/= mt) .

*** Gt must be a valid TypeContext.
eq validslots?(Gt, T) = if ((T =/= error) and (Gt =/= mt)) then
                          (slots?(T) == mt) or
                            (((slots?(T) & slots-of?(Gt, type?(T))) == mt) and
                             (types-of?(slots?(T)) in (types-in?(Gt) name?(T))))
                        else false fi .

*** valid? presumes that all parameter TypeContext's are valid.
eq valid?(Gt, T) = not(name?(T) in Gt) and
                   validparents?(Gt, name?(T), type?(T)) and
                   validslots?(Gt, T) .
eq valid?(Gt, Gt' T) = not(name?(T) in Gt Gt') and
                       validparents?(Gt Gt', name?(T), type?(T)) and
                       validslots?(Gt Gt', T) .

*** Done.
eq add Gt' to Gt # Gi = if Gt' == (mt).TypeContext then
                           Gt # Gi
                        else if valid?(Gt, Gt') then
                                (Gt Gt' # Gi)
                             else (err).Environment?
                             fi
```

```
                        fi .

*** Done.
eq is-type?(Gt, G) = G in Gt .
*** Done.
eq is-subtype-of?(Gt, G0, G1) = G1 in type-of?(Gt, G0) .
*** Done.
eq is?(Gt, G0, G1) = is-type?(Gt, G0) and (G0 == G1) .
*** Done.
eq  is-part-of?(Gt, G0, G1) = if is-type?(Gt, G0) and is-type?(Gt, G1) then
                                (G0 type-in? slots-of?(Gt, G1))
                              else false fi .

*** Done.
ceq type-of?(Gt, G) = -0- if not(G in Gt) .
ceq type-of?(Gt, G) = G if G == name?(Universal) .
ceq type-of?(Gt, G) = type?(get(Gt, G)) type-of?(Gt, type?(get(Gt, G)))
                      if (G in Gt) and G =/= name?(Universal) .
eq  type-of?(Gt, N N') = N N' type-of?(Gt, type?(get(Gt, N)))
                         type-of?(Gt, type?(get(Gt, N'))) .
ceq type-of?(Gt, N TS) = N type-of?(Gt, type?(get(Gt, N))) type-of?(Gt, TS)
                         if (TS =/= (mt).TypeNameSet) .

*** Done.
ceq slots-of?(Gt, G) = (err).SlotSet? if not(G in Gt) .
ceq slots-of?(Gt, G) = (mt).SlotSet if G == name?(Universal) .
ceq slots-of?(Gt, G) = slots?(get(Gt, G)) slots-of?(Gt, type-of?(Gt, G))
                       if (G in Gt) and G =/= name?(Universal) .
ceq slots-of?(Gt, N TS) = slots-of?(Gt, N) slots-of?(Gt, TS)
                          if (TS =/= (mt).TypeNameSet) .

*** Done.
eq is-instance?(Gt # Gi, G) = G in Gi .

*** Done.
eq is-a-type-of?(Gt # Gi, G0, G1) = if (is-instance?(Gt # Gi, G0) and
                                        is-type?(Gt, G1)) then
                                      ((type?(get(Gi, G0)) == G1) or
                                       (G1 in type-of?(Gt, type?(get(Gi, G0)))))
                                    else false fi .

*** Done.
eq is?(Gt # Gi, G0, G1) = is-instance?(Gt # Gi, G0) and (G0 == G1) .

*** Done.
```

```
  eq is-part-of?(Gt # Gi, G0, G1) = if is-instance?(Gt # Gi, G0) and
                                      is-instance?(Gt # Gi, G1) then
                                     (G0 in values-of?(valueslots?(get(Gi, G1))))
                                    else false fi .

  *** Done.
  eq type-of?(Gt # Gi, G) = if is-instance?(Gt # Gi, G) then
                              type?(get(Gi, G)) type-of?(Gt, type?(get(Gi, G)))
                            else err fi .

  *** Done.
  eq valid?(Gt # Gi, I) = not(name?(I) in Gi) and
                          is-type?(Gt, type?(I)) and
                          types-of?(slots-of?(valueslots?(I))) in Gt and
                          strings-only(values-of?(valueslots?(I))) in
                            (instances-in?(Gi) name?(I)).

  eq valid?(Gt # Gi, I Gi') = not(name?(I) in Gi) and
                              is-type?(Gt, type?(I)) and
                              types-of?(slots-of?(valueslots?(I))) in Gt and
                              strings-only(values-of?(valueslots?(I))) in
                                (instances-in?(Gi Gi') name?(I)) and
                              valid?(Gt # Gi I, Gi') .

  *** Done.
  eq add Gi' to Gt # Gi = if Gi' == (mt).InstanceContext then
                             Gt # Gi
                          else if valid?(Gt # Gi, Gi') then
                                  (Gt # Gi Gi')
                               else (err).Environment?
                               fi
                          fi .

  *** Done.
  eq add Gt # Gi to Gt' # Gi' = add Gi to (add (Gt - GammaT0) to (Gt' # Gi')) .

endfm)
```

A.5 Unit Tests

```
*** Formalization of the Theory of Kind's Type Theory
*** Joseph Kiniry
*** Started January 25, 2000
*** Maude version started April 17, 2000
```

```
*** $Id: appendices.tex,v 1.22 2002/06/06 20:16:26 kiniry Exp $

*** Tests for KTT.
*** Included is a large, rigorous set of tests for KTT, exercising each
*** operator on basic constructs as well as all judgments in the logic.

*** ==========================================================================

*** Testing GROUND-SET.

(fmod GROUND-SET-TEST is
  protecting GROUND-SET .

  ops i0 i1 i2 s0 s1 s2 b0 b1 f0 f1 f2 : -> Ground .
  ops I S B F : -> GroundSet .

  *** Define some names of legitimate instances of basic types.
  eq i0 = # 0 .
  eq i1 = # 1 .
  eq i2 = # 2 .
  eq s0 = $ 'zero .
  eq s1 = $ 'one .
  eq s2 = $ 'two .
  eq b0 = ! true .
  eq b1 = ! false .
  eq f0 = # 0 . 0 . 0 .
  eq f1 = # -1 . 0 . 5 .
  eq f2 = # 2 . 1 . 5 .

  *** Collect them type-wise into sets.
  eq I = i0 i1 i2 .
  eq S = s0 s1 s2 .
  eq B = b0 b1 .
  eq F = f0 f1 f2 .
endfm)

*** Various tests for strings-only.

*** Empty set.
(red strings-only(mt) == mt .)
*** Singleton sets of instances of names of basic types.
(red strings-only(i0) == mt .)
(red strings-only(s0) == s0 .)
(red strings-only(b0) == mt .)
(red strings-only(f0) == mt .)
```

```
*** Sets of such.
(red strings-only(i0 s0) == s0 .)
(red strings-only(i0 b0) == mt .)
(red strings-only(i0 f0) == mt .)
(red strings-only(s0 b0) == s0 .)
(red strings-only(s0 f0) == s0 .)
(red strings-only(b0 f0) == mt .)

*** Testing SET.

(view VQid from TRIV to QID is
  sort Elt to Qid .
 endv)

(fmod SET-TEST-QID is
  protecting SET[VQid]
  * (sort Set[VQid] to QidSet,
     sort Set?[VQid] to QidSet?) .

  ops a b c d e f : -> Qid .
  eq a = 'a .  eq b = 'b .  eq c = 'c .
  eq d = 'd .  eq e = 'e .  eq f = 'f .

  ops S0 S1 S2 S3 : -> QidSet .

  eq S0 = mt .
  eq S1 = e f S0 .
  eq S2 = b c S1 .
  eq S3 = a b c d e f .
 endfm)

(red S3      ==  a b c d e f .)
(red S3 S3   ==  a a b b c c d d e e f f .)

(red a in S0 == false .)
(red b in S1 == false .)
(red e in S1 == true .)
(red e in S2 == true .)

(red S0 in S0 == true .)
(red S0 in S1 == true .)
(red S0 in S2 == true .)
(red S1 in S2 == true .)
(red S1 in S3 == true .)
(red S2 in S3 == true .)
```

```
(red S3 in S2 == false .)
(red S3 in S0 == false .)

(red S0 - S0 == mt .)
(red S1 - S0 == S1 .)
(red S0 - S2 == mt .)
(red S1 - S2 == mt .)
(red S2 - S1 == b c .)
(red S3 - S0 == S3 .)
(red S2 - S3 == mt .)
(red S3 - S2 == a d .)

(red | S0 | == 0 .)
(red | S1 | == 2 .)
(red | S2 | == 4 .)
(red | S3 | == 6 .)
(red | S1 S0 | == 2 .)
(red | S1 S2 | == 4 .)
(red | S1 S2 S3 | == 6 .)
(red | S3 S3 | == 6 .)

(red S0 & S0 == mt .)
(red S1 & S0 == mt .)
(red S0 & S2 == mt .)
(red S1 & S2 == S1 .)
(red S2 & S1 == S1 .)
(red S3 & S0 == mt .)
(red S2 & S3 == S2 .)

(view VInt from TRIV to MACHINE-INT is
  sort Elt to MachineInt .
 endv)

*** Testing TYPE.

(fmod TYPE-TEST is
  protecting TYPE .

  op N : -> TypeName .
  op TS : -> TypeNameSet .
  op SS : -> SlotSet .

  eq N = $ 'PersonType .
  eq TS = $ 'Universal $ 'NamedThing $ 'PhysicalEntity .
  eq SS = mt .
```

```
  op TE : -> Type .
  eq TE = < N ; TS ; SS > .
 endfm)

(red TE         == < N ; TS ; SS > .)
(red name?(TE)  == $ 'PersonType .)
(red type?(TE)  == $ 'NamedThing $ 'Universal $ 'PhysicalEntity .)
(red slots?(TE) == mt .)

*** Testing INSTANCE.

(fmod INSTANCE-TEST is
  protecting INSTANCE .
  protecting GROUND-SET .

  op I : -> InstanceName .
  op T : -> TypeName .
  op VSS : -> ValueSlotSet .
  op GL : -> GroundList .

  eq GL = $ 'Joe ‘, $ 'Kiniry .
  eq I = $ 'JoeInstance .
  eq T = $ 'PersonType .

  eq VSS = < < $ 'String ; $ 'name >   ; GL        >
           < < $ 'Float  ; $ 'height > ; # 6 . 1 . 5 >
           < < $ 'Float  ; $ 'weight > ; # 89     > .

  op IE : -> Instance .
  eq IE = [ I ; T ; VSS ] .
 endfm)

(red IE              == [ I ; T ; VSS ] .)
(red name?(IE)       ==  $ 'JoeInstance .)
(red type?(IE)       ==  $ 'PersonType .)
(red valueslots?(IE) == < < $ 'Float  ; $ 'height > ; # 6 . 1 . 5 >
                        < < $ 'String ; $ 'name >   ; GL       >
                        < < $ 'Float  ; $ 'weight > ; # 89     > .)

*** Testing TYPECONTEXT.

(fmod TYPECONTEXT-TEST is
  protecting TYPE-TEST .
  protecting TYPECONTEXT .
```

```
  op  Gt : -> TypeContext .
  ops T0 T1 T2 : -> Type .

  eq T0 = < $ 'Type0 ; TS ; SS > .
  eq T1 = < $ 'Type1 ; TS ; SS > .
  eq T2 = < $ 'Type2 ; TS ; SS > .
  eq Gt = TE T0 T1 .

 endfm)

(red Gt                 == T0 T1 TE .)
(red T0 in Gt           == true .)
(red T1 in Gt           == true .)
(red T2 in Gt           == false .)
(red T2 in Gt T2        == true .)
(red name?(T0) in Gt    == true .)
(red name?(T1) in Gt    == true .)
(red name?(T2) in Gt    == false .)
(red name?(T2) in Gt T2 == true .)
(red $ 'Type0 in Gt     == true .)
(red $ 'Type1 in Gt     == true .)
(red $ 'Type2 in Gt     == false .)
(red $ 'Type2 in Gt T2  == true .)

*** Testing rules.

(fmod KTT-TEST is
  protecting TYPE-TEST .
  protecting INSTANCE-TEST .
  protecting KTT .

  ops T0 GT PEG : -> TypeContext .
  op  E0 : -> Environment .
  ops t0 t1 t2 t3 t4 t5 t6 t7 t8 t9 : -> Type .
  ops t10 t11 t12 t13 t14 t15 t16 t17 t18 t19 : -> Type .
  ops t20 t21 t22 t23 t24 t25 t26 t27 t28 t29 : -> Type .
  ops t30 t31 t32 t33 t34 t35 t36 t37 t38 t39 : -> Type .
  ops t40 t41 t42 t43 t44 t45 t46 t47 t48 t49 : -> Type .

  eq T0 = Basic .
  eq E0 = T0 # mt .

  eq t0 = Universal .
  eq t1 = Int .
```

```
eq t2 = < $ 'New ; $ 'Universal ; mt > .
eq t3 = < $ 'Bool ; $ 'Universal ; mt > .
eq t4 = < $ 'New ; $ 'Int ; mt > .
eq t5 = < $ 'Bool ; $ 'Int ; mt > .
eq t6 = < $ 'New ; $ 'DNE ; mt > .
eq t7 = < $ 'Bool ; $ 'DNE ; mt > .

eq t8 = < $ 'TypeT1 ; $ 'Universal ; < $ 'Bool ; $ 'good > > .
eq t9 = < $ 'TypeT1 ; $ 'Universal ; < $ 'DNE ; $ 'bad > > .
eq t10 = < $ 'Bool ; $ 'Universal ; < $ 'Bool ; $ 'good > > .
eq t11 = < $ 'Bool ; $ 'Universal ; < $ 'DNE ; $ 'bad > > .
eq t12 = < $ 'TypeT1 ; $ 'Int ; < $ 'Bool ; $ 'good > > .
eq t13 = < $ 'TypeT1 ; $ 'Int ; < $ 'DNE ; $ 'bad > > .
eq t14 = < $ 'Bool ; $ 'Int ; < $ 'Bool ; $ 'good > > .
eq t15 = < $ 'Bool ; $ 'Int ; < $ 'DNE ; $ 'bad > > .
eq t16 = < $ 'TypeT1 ; $ 'DNE ; < $ 'Bool ; $ 'good > > .
eq t17 = < $ 'TypeT1 ; $ 'DNE ; < $ 'DNE ; $ 'bad > > .
eq t18 = < $ 'Bool ; $ 'DNE ; < $ 'Bool ; $ 'good > > .
eq t19 = < $ 'Bool ; $ 'DNE ; < $ 'DNE ; $ 'bad > > .

eq t20 = < $ 'TypeT1 ; $ 'Universal ; < $ 'Bool ; $ 'good > < $ 'Bool ; $ 'good > > .
eq t21 = < $ 'TypeT1 ; $ 'Universal ; < $ 'Bool ; $ 'good > < $ 'DNE ; $ 'bad > > .
eq t22 = < $ 'TypeT1 ; $ 'Universal ; < $ 'DNE ; $ 'bad > < $ 'Bool ; $ 'good > > .
eq t23 = < $ 'TypTe1 ; $ 'Universal ; < $ 'DNE ; $ 'bad > < $ 'DNE ; $ 'bad > > .

eq t24 = < $ 'Bool ; $ 'Universal ; < $ 'Bool ; $ 'good > < $ 'Bool ; $ 'good > > .
eq t25 = < $ 'Bool ; $ 'Universal ; < $ 'Bool ; $ 'good > < $ 'DNE ; $ 'bad > > .
eq t26 = < $ 'Bool ; $ 'Universal ; < $ 'DNE ; $ 'bad > < $ 'Bool ; $ 'good > > .
eq t27 = < $ 'Bool ; $ 'Universal ; < $ 'DNE ; $ 'bad > < $ 'DNE ; $ 'bad > > .

eq t28 = < $ 'TypeT1 ; $ 'Int ; < $ 'Bool ; $ 'good > < $ 'Bool ; $ 'good > > .
eq t29 = < $ 'TypeT1 ; $ 'Int ; < $ 'Bool ; $ 'good > < $ 'DNE ; $ 'bad > > .
eq t30 = < $ 'TypeT1 ; $ 'Int ; < $ 'DNE ; $ 'bad > < $ 'Bool ; $ 'good > > .
eq t31 = < $ 'TypeT1 ; $ 'Int ; < $ 'DNE ; $ 'bad > < $ 'DNE ; $ 'bad > > .

eq t32 = < $ 'Bool ; $ 'Int ; < $ 'Bool ; $ 'good > < $ 'Bool ; $ 'good > > .
eq t33 = < $ 'Bool ; $ 'Int ; < $ 'Bool ; $ 'good > < $ 'DNE ; $ 'bad > > .
eq t34 = < $ 'Bool ; $ 'Int ; < $ 'DNE ; $ 'bad > < $ 'Bool ; $ 'good > > .
eq t35 = < $ 'Bool ; $ 'Int ; < $ 'DNE ; $ 'bad > < $ 'DNE ; $ 'bad > > .

eq t36 = < $ 'TypeT1 ; $ 'DNE ; < $ 'Bool ; $ 'good > < $ 'Bool ; $ 'good > > .
eq t37 = < $ 'TypeT1 ; $ 'DNE ; < $ 'Bool ; $ 'good > < $ 'DNE ; $ 'bad > > .
eq t38 = < $ 'TypeT1 ; $ 'DNE ; < $ 'DNE ; $ 'bad > < $ 'Bool ; $ 'good > > .
eq t39 = < $ 'TypeT1 ; $ 'DNE ; < $ 'DNE ; $ 'bad > < $ 'DNE ; $ 'bad > > .
```

```
  eq t40 = < $ 'Bool ; $ 'DNE ; < $ 'Bool ; $ 'good > < $ 'Bool ; $ 'good > > .
  eq t41 = < $ 'Bool ; $ 'DNE ; < $ 'Bool ; $ 'good > < $ 'DNE ; $ 'bad > > .
  eq t42 = < $ 'Bool ; $ 'DNE ; < $ 'DNE ; $ 'bad > < $ 'Bool ; $ 'good > > .
  eq t43 = < $ 'Bool ; $ 'DNE ; < $ 'DNE ; $ 'bad > < $ 'DNE ; $ 'bad > > .

  *** recursive type
  eq t44 = < $ 'T0 ; $ 'Universal ; < $ 'T0 ; $ 'a > > .
  *** incomplete type
  eq t45 = < $ 'T0 ; $ 'Universal ; < $ 'T1 ; $ 'a > > .
  *** incomplete open typecontext
  eq t46 = < $ 'T0 ; $ 'Universal ; < $ 'T1 ; $ 'a > >
           < $ 'T1 ; $ 'Universal ; < $ 'T2 ; $ 'a > > .
  *** complete closed typecontext
  eq t47 = < $ 'T0 ; $ 'Universal ; < $ 'T1 ; $ 'a > >
           < $ 'T1 ; $ 'Universal ; < $ 'T0 ; $ 'a > > .

  eq GT =
          < $ 'J ; $ 'I ; mt >
          < $ 'I ; $ 'A ; mt >
          < $ 'H ; $ 'F $ 'B $ 'C ; mt >
          < $ 'G ; $ 'A $ 'E ; mt >
          < $ 'F ; $ 'D $ 'E ; mt >
          < $ 'E ; $ 'B $ 'C ; mt >
          < $ 'D ; $ 'A $ 'B ; mt >
          < $ 'C ; $ 'Universal ; mt >
          < $ 'B ; $ 'Universal ; mt >
          < $ 'A ; $ 'Universal ; mt >
          < $ 'Bool      ; $ 'Universal ; mt >
          < $ 'Int       ; $ 'Universal ; mt >
          < $ 'Float     ; $ 'Universal ; mt >
          < $ 'String    ; $ 'Universal ; mt >
          < $ 'Universal ; $ 'Universal ; mt > .

   eq PEG = TE < $ 'NamedThing ; $ 'Universal ;
                 < $ 'String ; $ 'name > >
               < $ 'PhysicalEntity ; $ 'Universal ;
                 < $ 'Float  ; $ 'height >
                 < $ 'Float  ; $ 'weight > >
               Basic .
 endfm)

(red E0 == T0 # mt .)

*** validparents?
```

```
(red validparents?(mt, -0-, mt)                                  == false .)
(red validparents?(mt, $ 'Bool, mt)                              == false .)
(red validparents?(mt, -0-, $ 'Bool)                             == false .)
(red validparents?(mt, $ 'Bool, $ 'Universal)                    == false .)
(red validparents?(Basic, $ 'A, $ 'Universal)                    == true .)
(red validparents?(Basic, $ 'A, $ 'Bool $ 'Universal)            == true .)
(red validparents?(Basic, $ 'A, $ 'Bool $ 'Int $ 'Universal) == true .)
(red validparents?(Basic, $ 'A, $ 'Foobar)                       == false .)
(red validparents?(Basic, $ 'A, $ 'Int $ 'Foobar)                == false .)

*** validslots?

(red validslots?(mt, error) == false .)
(red validslots?(mt, < $ 'new ; $ 'Universal ; mt >)    == false .)
(red validslots?(Basic, < $ 'new ; $ 'Universal ; mt >) == true .)
(red validslots?(Basic, < $ 'new ; $ 'Universal ; < $ 'Float ; $ 'foobar > >) == true .)
(red validslots?(Basic, < $ 'new ; $ 'Universal ; < $ 'Foobar ; $ 'foobar > >) == false .)
(red validslots?(Basic, < $ 'new ; $ 'Universal ;
                        < $ 'Foobar ; $ 'foobar > < $ 'Float ; $ 'foobar > >) == false .)
(red validslots?(Basic, < $ 'new ; $ 'Universal ;
                        < $ 'Int ; $ 'foobar > < $ 'Float ; $ 'foobar > >) == true .)
(red validslots?(Basic, < $ 'new ; $ 'Universal ;
                        < $ 'Int ; $ 'foobar > < $ 'Foobar ; $ 'foobar > >) == false .)

(red validslots?(< $ 'TestType ; $ 'Universal ; mt > Basic,
                 < $ 'new ; $ 'Universal ; < $ 'TestType ; $ 'foobar > >) == true .)
(red validslots?(< $ 'TestType ; $ 'Universal ; mt > Basic,
                 < $ 'new ; $ 'Universal ;
                   < $ 'TestType ; $ 'tt >
                   < $ 'Bool     ; $ 'b >
                   < $ 'Int      ; $ 'i > >) == true .)
(red validslots?(< $ 'TestType ; $ 'Universal ; mt > Basic,
                 < $ 'new ; $ 'Universal ;
                   < $ 'Float     ; $ 'f >
                   < $ 'Universal ; $ 'u >
                   < $ 'String    ; $ 's > >) == true .)
(red validslots?(< $ 'TestType ; $ 'Universal ; mt > Basic,
                 < $ 'new ; $ 'Universal ;
                   < $ 'Float       ; $ 'f >
                   < $ 'Universal   ; $ 'u >
                   < $ 'UnknownType ; $ 'u >
                   < $ 'String      ; $ 's > >) == false .)

*** valid? (for types) depends upon _in_, validparents?, and validslots?
```

```
(red valid?(T0, t0) == false .)
(red valid?(T0, t1) == false .)
(red valid?(T0, t2) == true .)
(red valid?(T0, t3) == false .)
(red valid?(T0, t4) == true .)
(red valid?(T0, t5) == false .)
(red valid?(T0, t6) == false .)
(red valid?(T0, t7) == false .)

(red valid?(T0, t8) == true .)
(red valid?(T0, t9) == false .)

(red valid?(T0, t10) == false .)
(red valid?(T0, t11) == false .)
(red valid?(T0, t12) == true .)
(red valid?(T0, t13) == false .)
(red valid?(T0, t14) == false .)
(red valid?(T0, t15) == false .)
(red valid?(T0, t16) == false .)
(red valid?(T0, t17) == false .)
(red valid?(T0, t18) == false .)
(red valid?(T0, t19) == false .)

(red valid?(T0, t20) == true .)
(red valid?(T0, t21) == false .)
(red valid?(T0, t22) == false .)
(red valid?(T0, t23) == false .)
(red valid?(T0, t24) == false .)
(red valid?(T0, t25) == false .)
(red valid?(T0, t26) == false .)
(red valid?(T0, t27) == false .)
(red valid?(T0, t28) == true .)
(red valid?(T0, t29) == false .)

(red valid?(T0, t30) == false .)
(red valid?(T0, t31) == false .)
(red valid?(T0, t32) == false .)
(red valid?(T0, t33) == false .)
(red valid?(T0, t34) == false .)
(red valid?(T0, t35) == false .)
(red valid?(T0, t36) == false .)
(red valid?(T0, t37) == false .)
(red valid?(T0, t38) == false .)
(red valid?(T0, t39) == false .)
```

```
(red valid?(T0, t40) == false .)
(red valid?(T0, t41) == false .)
(red valid?(T0, t42) == false .)
(red valid?(T0, t43) == false .)

(red valid?(T0, t44) == true .)
(red valid?(T0, t45) == false .)
(red valid?(T0, t46) == false .)
(red valid?(T0, t47) == true .)

*** add_to_ for types

(red add (mt).TypeContext to (Basic # mt) .)
(red add < $ 'TestType ; $ 'Universal ; (mt).SlotSet > to (Basic # mt) ==
  < $ 'TestType ; $ 'Universal ; mt > Basic # mt .)
(red add t44 to Basic # mt == t44 Basic # mt .)
(red add t45 to Basic # mt == (err).Environment? .)
(red add t46 to Basic # mt == (err).Environment? .)
(red add t47 to Basic # mt == t47 Basic # mt .)

*** type-of?

(red type-of?(Basic, name?(Universal)) == name?(Universal) .)
(red type-of?(Basic, name?(Bool)) == name?(Universal) .)
(red type-of?(PEG, $ 'NamedThing) == name?(Universal) .)
(red type-of?(PEG, $ 'PhysicalEntity) == name?(Universal) .)
(red type-of?(PEG, $ 'PersonType) == $ 'Universal $ 'NamedThing $ 'PhysicalEntity .)

(red type-of?(GT, $ 'Universal) == name?(Universal) .)
(red type-of?(GT, $ 'A) == name?(Universal) .)
(red type-of?(GT, $ 'B) == name?(Universal) .)
(red type-of?(GT, $ 'C) == name?(Universal) .)
(red type-of?(GT, $ 'D) == $ 'Universal $ 'A $ 'B .)
(red type-of?(GT, $ 'E) == $ 'Universal $ 'B $ 'C .)
(red type-of?(GT, $ 'F) == $ 'Universal $ 'A $ 'B $ 'C $ 'D $ 'E .)
(red type-of?(GT, $ 'G) == $ 'Universal $ 'A $ 'B $ 'C $ 'E .)
(red type-of?(GT, $ 'H) == $ 'Universal $ 'A $ 'B $ 'C $ 'D $ 'E $ 'F .)

*** slots-of?

(red slots-of?(Basic, name?(Universal)) == mt .)
(red slots-of?(Basic, name?(Bool)) == mt .)
(red slots-of?(PEG, $ 'NamedThing) == < $ 'String ; $ 'name > .)
(red slots-of?(PEG, $ 'PhysicalEntity) ==
```

```
   < $ 'Float ; $ 'height > < $ 'Float ; $ 'weight > .)
(red slots-of?(PEG, $ 'PersonType) ==
   < $ 'String ; $ 'name >
   < $ 'Float ; $ 'height >
   < $ 'Float ; $ 'weight > .)

(red slots-of?(GT, $ 'Universal) == mt .)
(red slots-of?(GT, $ 'A) == mt .)
(red slots-of?(GT, $ 'B) == mt .)
(red slots-of?(GT, $ 'C) == mt .)
(red slots-of?(GT, $ 'D) == mt .)
(red slots-of?(GT, $ 'E) == mt .)
(red slots-of?(GT, $ 'F) == mt .)
(red slots-of?(GT, $ 'G) == mt .)
(red slots-of?(GT, $ 'H) == mt .)

*** is-type?

(red is-type?(Basic, $ 'Foo)          == false .)
(red is-type?(Basic, $ 'Universal)    == true .)
(red is-type?(Basic, $ 'Bool)         == true .)
(red is-type?(Basic, $ 'bool)         == false .)
(red is-type?(PEG, $ 'PhysicalEntity) == true .)
(red is-type?(PEG, $ 'Int)            == true .)
(red is-type?(PEG, $ 'PersonType)     == true .)

*** is-subtype-of?

(red is-subtype-of?(Basic, $ 'Universal, $ 'Universal) == true .)
(red is-subtype-of?(Basic, $ 'Universal, $ 'Bool) == false .)
(red is-subtype-of?(Basic, $ 'Bool, $ 'Universal) == true .)
(red is-subtype-of?(Basic, $ 'Float, $ 'Int) == false .)
(red is-subtype-of?(Basic, $ 'Int, $ 'Float) == false .)
(red is-subtype-of?(PEG, $ 'PhysicalEntity, $ 'Universal) == true .)
(red is-subtype-of?(PEG, $ 'Universal, $ 'PhysicalEntity) == false .)
(red is-subtype-of?(PEG, $ 'PersonType, $ 'PhysicalEntity) == true .)
(red is-subtype-of?(PEG, $ 'PersonType, $ 'Universal) == true .)
(red is-subtype-of?(GT, $ 'A, $ 'Universal) == true .)
(red is-subtype-of?(GT, $ 'D, $ 'Universal) == true .)
(red is-subtype-of?(GT, $ 'H, $ 'Universal) == true .)
(red is-subtype-of?(GT, $ 'F, $ 'B) == true .)
(red is-subtype-of?(GT, $ 'F, $ 'I) == false .)
(red is-subtype-of?(GT, $ 'J, $ 'A) == true .)
(red is-subtype-of?(GT, $ 'H, $ 'E) == true .)
(red is-subtype-of?(GT, $ 'E, $ 'H) == false .)
```

```
*** is?

(red is?(Basic, $ 'Universal, $ 'Universal) == true .)
(red is?(Basic, $ 'Bool, $ 'Universal) == false .)
(red is?(Basic, $ 'bool, $ 'bool) == false .)
(red is?(Basic, $ 'Int, $ 'Int) == true .)
(red is?(PEG, $ 'PersonType, $ 'PersonType) == true .)
(red is?(PEG, $ 'PysicalEntity, $ 'PersonType) == false .)
(red is?(GT, $ 'D, $ 'D) == true .)
(red is?(GT, $ 'F, $ 'D) == false .)
(red is?(GT, $ 'K, $ 'K) == false .)

*** is-part-of?

(red is-part-of?(Basic, $ 'Universal, $ 'Universal) == false .)
(red is-part-of?(Basic, $ 'bool, $ 'bool) == false .)
(red is-part-of?(Basic, $ 'Bool, $ 'Universal) == false .)
(red is-part-of?(PEG, $ 'PersonType, $ 'PersonType) == false .)
(red is-part-of?(PEG, $ 'Float, $ 'PersonType) == true .)
(red is-part-of?(PEG, $ 'Int, $ 'PersonType) == false .)
(red is-part-of?(GT, $ 'D, $ 'D) == false .)
(red is-part-of?(GT, $ 'F, $ 'D) == false .)
(red is-part-of?(GT, $ 'K, $ 'K) == false .)

*** is-instance?
(red is-instance?(Basic # mt, -0-) == false .)
(red is-instance?(Basic # mt, $ 'Universal) == false .)
(red is-instance?(Basic # [ $ 'new ; $ 'Universal ; mt ], -0-) == false .)
(red is-instance?(Basic # [ $ 'new ; $ 'Universal ; mt ], $ 'Bool) == false .)
(red is-instance?(Basic # [ $ 'new ; $ 'Universal ; mt ], $ 'new) == true .)
(red is-instance?(Basic # String-Blank [ $ 'new ; $ 'Universal ; mt ], $ 'new) == true .)
(red is-instance?(Basic # String-Blank Bool-False Bool-True, $ 'new) == false .)

*** is-a-type-of?
(red is-a-type-of?(Basic # mt, -0-, -0-) == false .)
(red is-a-type-of?(Basic # mt, -0-, $ 'Universal) == false .)
(red is-a-type-of?(Basic # [ $ 'new ; $ 'Universal ; mt ], -0-, -0-)
  == false .)
(red is-a-type-of?(Basic # [ $ 'new ; $ 'Universal ; mt ], -0-, $ 'new)
  == false .)
(red is-a-type-of?(Basic # [ $ 'new ; $ 'Universal ; mt ], $ 'new, $ 'Bool)
  == false .)
(red is-a-type-of?(Basic # [ $ 'new ; $ 'Universal ; mt ], $ 'new, $ 'Universal)
  == true .)
```

```
(red is-a-type-of?(Basic # String-Blank [ $ 'new ; $ 'Bool ; mt ], $ 'new, $ 'Bool)
  == true .)
(red is-a-type-of?(Basic # String-Blank [ $ 'new ; $ 'Bool ; mt ], $ 'new, $ 'Universal)
  == true .)
(red is-a-type-of?(Basic # String-Blank Bool-False Bool-True, $ 'new, $ 'Universal)
  == false .)

*** is?
(red is?(Basic # mt, -0-, -0-) == false .)
(red is?(Basic # mt, -0-, $ 'Universal) == false .)
(red is?(Basic # [ $ 'new ; $ 'Universal ; mt ], -0-, -0-) == false .)
(red is?(Basic # [ $ 'new ; $ 'Universal ; mt ], -0-, $ 'new) == false .)
(red is?(Basic # [ $ 'new ; $ 'Universal ; mt ], $ 'new, $ 'Bool) == false .)
(red is?(Basic # [ $ 'new ; $ 'Universal ; mt ], $ 'new, $ 'new) == true .)
(red is?(Basic # String-Blank [ $ 'new ; $ 'Bool ; mt ], $ 'new, $ 'Bool)
  == false .)
(red is?(Basic # String-Blank [ $ 'new ; $ 'Bool ; mt ], $ 'new, $ 'blank)
  == false .)
(red is?(Basic # String-Blank Bool-False Bool-True, $ 'new, $ 'new) == false .)

*** is-part-of?
(red is-part-of?(Basic # mt, -0-, -0-) == false .)
(red is-part-of?(Basic # mt, -0-, $ 'Universal) == false .)
(red is-part-of?(Basic # [ $ 'new ; $ 'Universal ; mt ], -0-, -0-) == false .)
(red is-part-of?(Basic # [ $ 'new ; $ 'Universal ; mt ], -0-, $ 'new)
  == false .)
(red is-part-of?(Basic # [ $ 'new ; $ 'Universal ; mt ], $ 'new, $ 'Bool)
  == false .)
(red is-part-of?(PEG TE # IE [ # 89 ; $ 'Int ; mt ]
                             [ # 6 . 1 . 5 ; $ 'Float ; mt ]
                             [ $ 'Joe ; $ 'String ; mt ]
                             [ $ 'Kiniry ; $ 'String ; mt ], # 89, $ 'JoeInstance)
  == true .)
(red is-part-of?(PEG TE # IE [ # 89 ; $ 'Int ; mt ]
                             [ # 6 . 1 . 5 ; $ 'Float ; mt ]
                             [ $ 'Joe ; $ 'String ; mt ]
                             [ $ 'Kiniry ; $ 'String ; mt ], $ 'Joe, $ 'JoeInstance)
  == true .)
(red is-part-of?(PEG TE # IE [ $ 0 ; $ 'Int ; mt ]
                             [ # 89 ; $ 'Int ; mt ]
                             [ # 6 . 1 . 5 ; $ 'Float ; mt ]
                             [ $ 'Joe ; $ 'String ; mt ]
                             [ $ 'Kiniry ; $ 'String ; mt ], $ 0, $ 'JoeInstance)
  == false .)
```

```
*** type-of?
(red type-of?(Basic # mt, -0-) == err .)
(red type-of?(Basic # mt, $ 'Universal) == err .)
(red type-of?(Basic # [ $ 'new ; $ 'Universal ; mt ], -0-) == err .)
(red type-of?(Basic # [ $ 'new ; $ 'Universal ; mt ], $ 'new)
  == $ 'Universal .)
(red type-of?(Basic # [ $ 'new ; $ 'Universal ; mt ], $ 'Bool) == err .)
(red type-of?(Basic # [ $ 'new ; $ 'Universal ; mt ], $ 'New) == err .)
(red type-of?(Basic # String-Blank [ $ 'new ; $ 'Bool ; mt ], $ 'new)
  == $ 'Bool $ 'Universal .)
(red type-of?(Basic # String-Blank [ $ 'new ; $ 'Bool ; mt ], $ 'Bool)
  == err .)
(red type-of?(Basic # String-Blank [ $ 'new ; $ 'Bool ; mt ], $ 'blank)
  == $ 'String $ 'Universal .)
(red type-of?(Basic # String-Blank Bool-False Bool-True, $ 'new) == err .)

*** valid? (for instances) depends upon many of the above functions.

(red valid?(Basic # mt, error) == false .)
(red valid?(Basic # mt, Bool-False) == true .)
(red valid?(Basic # mt, String-Blank) == true .)
(red valid?(Gamma, error) == false .)
(red valid?(Gamma, Bool-False) == false .)
(red valid?(Gamma, String-Blank) == false .)
(red valid?(Gamma, [ $ 'new ; $ 'Universal ; mt ]) == true .)
(red valid?(PEG TE # [ # 89 ; $ 'Int ; mt ]
                     [ # 6 . 1 . 5 ; $ 'Float ; mt ]
                     [ $ 'Joe ; $ 'String ; mt ]
                     [ $ 'Kiniry ; $ 'String ; mt ], IE) == true .)

*** add_to_ for instances.

(red add (mt).InstanceContext to Basic # mt .)
(red add [ # 89 ; $ 'Int ; mt ] to (PEG TE # mt) == PEG TE # [ # 89 ; $ 'Int ; mt ] .)
(red add [ # 89 ; $ 'Foo ; mt ] to (PEG TE # mt) == err .)
(red add [ ! false ; $ 'Bool ; mt ] to (PEG TE # GammaIO) == err .)
(red add IE to (PEG TE # mt) == err .)
(red add IE to (PEG TE # [ # 89 ; $ 'Int ; mt ]) == err .)
(red add IE to (PEG TE # [ # 89 ; $ 'Int ; mt ]
                         [ # 6 . 1 . 5 ; $ 'Float ; mt ]
                         [ $ 'Joe ; $ 'String ; mt ]) == err .)
(red add IE to (PEG TE # [ # 89 ; $ 'Int ; mt ]
                         [ # 6 . 1 . 5 ; $ 'Float ; mt ]
                         [ $ 'Joe ; $ 'String ; mt ]
                         [ $ 'Kiniry ; $ 'String ; mt ]) ==
```

```
                        PEG TE # [ # 89 ; $ 'Int ; mt ]
                                 [ # 6 . 1 . 5 ; $ 'Float ; mt ]
                                 [ $ 'Joe ; $ 'String ; mt ]
                                 [ $ 'Kiniry ; $ 'String ; mt ]
                                 IE .)

*** add_to_ for environments + environments.

(red add (GT # mt) to (Basic # mt) == (GT Basic # mt) .)
(red add (< $ 'NamedThing ; $ 'Universal ;
            < $ 'String ; $ 'name > >
          < $ 'PhysicalEntity ; $ 'Universal ;
            < $ 'Float  ; $ 'height >
            < $ 'Float  ; $ 'weight > > # mt) to Gamma ==
  ( < $ 'Bool ; $ 'Universal ; mt > < $ 'Universal ; $ 'Universal ; mt >
    < $ 'String ; $ 'Universal ; mt > < $ 'Float ; $ 'Universal ; mt >
    < $ 'Int ; $ 'Universal ; mt > < $ 'NamedThing ; $ 'Universal ;
    < $ 'String ; $ 'name > > < $ 'PhysicalEntity ; $ 'Universal ;
    < $ 'Float ; $ 'height > < $ 'Float ; $ 'weight > > ) #
  [ ! true ; $ 'Bool ; mt ] [ ! false ; $ 'Bool ; mt ]
  [ $ 'blank ; $ 'String ; mt ] .)
(red add TE to
     (add (< $ 'NamedThing ; $ 'Universal ;
             < $ 'String ; $ 'name > >
           < $ 'PhysicalEntity ; $ 'Universal ;
             < $ 'Float  ; $ 'height >
             < $ 'Float  ; $ 'weight > > # mt) to Gamma) ==
 ( < $ 'Bool ; $ 'Universal ; mt > < $ 'Universal ; $ 'Universal ; mt >
   < $ 'String ; $ 'Universal ; mt > < $ 'Float ; $ 'Universal ; mt >
   < $ 'Int ; $ 'Universal ; mt > < $ 'NamedThing ; $ 'Universal ;
   < $ 'String ; $ 'name > > < $ 'PhysicalEntity ; $ 'Universal ;
                              < $ 'Float ; $ 'height >
                              < $ 'Float ; $ 'weight > >
   < $ 'PersonType ; $ 'Universal $ 'NamedThing $ 'PhysicalEntity ; mt > ) #
   [ ! true ; $ 'Bool ; mt ] [ ! false ; $ 'Bool ; mt ]
   [ $ 'blank ; $ 'String ; mt ] .)
(red add (Basic # [ # 89 ; $ 'Int ; mt ]
                  [ # 6 . 1 . 5 ; $ 'Float ; mt ]
                  [ $ 'Joe ; $ 'String ; mt ]
                  [ $ 'Kiniry ; $ 'String ; mt ]) to
         (add TE to
                 (add (< $ 'NamedThing ; $ 'Universal ;
                         < $ 'String ; $ 'name > >
                       < $ 'PhysicalEntity ; $ 'Universal ;
                         < $ 'Float  ; $ 'height >
```

```
                        < $ 'Float  ; $ 'weight > >) to (GammaT0 # mt))) ==
  (Basic < $ 'NamedThing ; $ 'Universal ;
           < $ 'String ; $ 'name > >
         < $ 'PhysicalEntity ; $ 'Universal ;
           < $ 'Float ; $ 'height >
           < $ 'Float ; $ 'weight > >
         < $ 'PersonType ;
           $ 'Universal $ 'NamedThing $ 'PhysicalEntity ; mt > ) #
        [ # 89 ; $ 'Int ; mt ] [ $ 'Joe ; $ 'String ; mt ]
        [ $ 'Kiniry ; $ 'String ; mt ] [ # 6 . 1 . 5 ; $ 'Float ; mt ] .)
(red add IE to (add (Basic # [ # 89 ; $ 'Int ; mt ]
                             [ # 6 . 1 . 5 ; $ 'Float ; mt ]
                             [ $ 'Joe ; $ 'String ; mt ]
                             [ $ 'Kiniry ; $ 'String ; mt ]) to
                    (add TE to
                           (add (< $ 'NamedThing ; $ 'Universal ;
                                  < $ 'String ; $ 'name > >
                                < $ 'PhysicalEntity ; $ 'Universal ;
                                  < $ 'Float  ; $ 'height >
                                  < $ 'Float  ; $ 'weight > > # mt) to Gamma))) ==
  (Basic < $ 'NamedThing ; $ 'Universal ;
           < $ 'String ; $ 'name > >
         < $ 'PhysicalEntity ; $ 'Universal ;
           < $ 'Float ; $ 'height >
           < $ 'Float ; $ 'weight > >
         < $ 'PersonType ; $ 'Universal $ 'NamedThing $ 'PhysicalEntity ; mt > ) #
        [ # 89 ; $ 'Int ; mt ] [ $ 'Joe ; $ 'String ; mt ] GammaI0
        [ $ 'Kiniry ; $ 'String ; mt ] [ # 6 . 1 . 5 ; $ 'Float ; mt ]
        [ $ 'JoeInstance ; $ 'PersonType ;
            < < $ 'String ; $ 'name > ; $ 'Joe , $ 'Kiniry >
            < < $ 'Float ; $ 'height > ; # 6 . 1 . 5 >
            < < $ 'Float ; $ 'weight > ; # 89 > ] .)

*** Recursive types.
(fmod RECURSIVE-TYPE-TEST is
  protecting KTT .

  op List : -> Type .
  eq List = < $ 'List ; $ 'Universal ;
                < $ 'Int ; $ 'value >
                < $ 'List ; $ 'next > > .

  ops A B C : -> Type .
  eq A = < $ 'A ; $ 'Universal ;
           < $ 'B  ; $ 'b > > .
```

```
  eq B = < $ 'B ; $ 'Universal ;
           < $ 'A  ; $ 'a > > .

  eq C = < $ 'C ; $ 'A ;
           < $ 'D  ; $ 'd > > .
 endfm)

*** Adding recursive types to contexts.

(red add List to Gamma == List GammaT0 # GammaI0 .)
(red add A B to Gamma == A B GammaT0 # GammaI0 .)
(red add C to (add A B to Gamma) == err .)

(fmod RECURSIVE-INSTANCE-TEST is
  protecting RECURSIVE-TYPE-TEST .

  ops ListE ABE : -> Environment .

  ops head middle last : -> Instance .

  ops A0 A1 A2 A3 B0 B1 B2 : -> Instance .

  eq ListE = add List to Gamma .
  eq ABE = add A B to Gamma .

  eq head = [ $ 'head ; $ 'List ;
              < < $ 'Int ; $ 'value > ; # 5 >
              < < $ 'List ; $ 'next > ; $ 'middle > ] .
  eq middle = [ $ 'middle ; $ 'List ;
              < < $ 'Int ; $ 'value > ; # 3 >
              < < $ 'List ; $ 'next > ; $ 'last > ] .
  eq last = [ $ 'last ; $ 'List ;
              < < $ 'Int ; $ 'value > ; # 0 > ] .

  eq A0 = [ $ 'A0 ; $ 'A ;
              < < $ 'B ; $ 'b > ; $ 'B0 > ] .
  eq B0 = [ $ 'B0 ; $ 'B ; mt ] .

  eq A1 = [ $ 'A1 ; $ 'A ;
              < < $ 'B ; $ 'b > ; $ 'B1 > ] .
  eq B1 = [ $ 'B1 ; $ 'B ;
              < < $ 'A ; $ 'a > ; $ 'A1 > ] .

  eq A2 = [ $ 'A2 ; $ 'A ;
              < < $ 'B ; $ 'b > ; $ 'B2 > ] .
```

```
  eq B2 = [ $ 'B2 ; $ 'B ;
              < < $ 'A ; $ 'a > ; $ 'A3 > ] .
  eq A3 = [ $ 'A3 ; $ 'A ;
              < < $ 'B ; $ 'b > ; $ 'B1 > ] .
 endfm)

*** Need to deal with basic types!
*** Correct because implicit nil for next and #0 is an instance of Int.
(red add last to ListE == List GammaT0 # GammaI0 last .)
*** Fails because head depends upon middle.
(red add head last to ListE == err .)
*** Correct because closed set.
(red add head middle last to ListE
  == List GammaT0 # GammaI0 head middle last .)

*** Error because B0 undefined.
(red add A0 to ABE == err .)
*** Correct because B0 depends on only type B.
(red add B0 to ABE == GammaT0 A B # GammaI0 B0 .)
*** Correct because B0 depends on only type B and A0 depends on B0.
(red add A0 B0 to ABE == GammaT0 A B # GammaI0 A0 B0 .)
*** Correct because closed cycle.
(red add A1 B1 to ABE == GammaT0 A B # GammaI0 A1 B1 .)
*** Fails because A3 depends upon B1.
(red add A2 B2 A3 to ABE == err .)
```

Appendix B

Semantic Properties

Table B.1: Meta-Information Properties

Property	Context	Purpose/Description
`author` Fullname [email]	Modules	Lists an author of the module. The text has no special internal structure. A doc comment may contain multiple author properties. A standard author property includes only the full name of the author and should not include other information (like an email address, web page, etc.) unless necessary.
`bon` Text	Modules, Features	Provides a one-sentence summary of the construct. This sentence is meant to correlate with the BON specification of the construct.
`bug` Text	Modules, Features	Describes a known bug in the module or feature. One property should be used for each bug. If a construct has a known bug, it should be described. Omission of a bug description is considered as bad an offense as the existence of the bug in the first place.
`copyright` Text	Files	A copyright notice.
`description` Text	Files	A brief description of a file.
`history` Text	All	Describes how a feature has been modified over time. For each major change, the description should include: who made it, when it was made (date and version), what was done, why it was done, and a reference to the change request and/or user requirement that resulted in the change being made.
`license` Text	Files	A license notice.
`title` Text	Files	Documents the title of the project with which the file is associated.

Table B.2: Pending Work Properties

Property	Context	Purpose/Description
`idea` Author [Classifier] - Text	All	Describes an idea for something from the named user. The optional *Classifier* argument is for ad hoc categorization of ideas. Typical examples include optimizing algorithms, proving correctness of new code, renaming variables or features to conform to standards, cleaning up code, etc.
`review` Author - Text	All	Suggests/Reminds a reader that a particular block of code still needs to be reviewed (by the named user) for some reason. Typical reasons include the "something's not right here" syndrome, boundary condition checks, not understanding an algorithm, etc.
`todo` Author - Text	All	Describes some work that still needs to be accomplished by the named user. Typical examples include optimizing algorithms, proving correctness of new code, renaming variables or features to conform to standards, cleaning up code, etc.

Table B.3: Formal Specification Properties

Property	Context	Purpose/Description
[label:] `ensure` (Predicate) [Exception -] Description	Features	Describes a postcondition that is true after a feature has completed successfully. If a postcondition is violated, then *Exception* is raised. One property is used for each postcondition. Note that no requirement is made on the visibility of postconditions. They are often used as the private, development specifications of a feature. The default visibility of a postcondition is the visibility of the feature to which it is bound.
`generate` [Description]	Modules, Features	Describes new, possibly unreferenced, entities (for example, new threads of control) constructed as a result of the execution of a feature or the instantiation of a module.
`invariant` (Predicate) [Exception] Description	Modules, Features	Specifies a module or feature invariant. An invariant is an expression that must evaluate to true whenever the entity being described is in a stable state. If an invariant is violated when the entity is in a stable state, the *Exception* condition is instantiated. The notions of exceptional conditions and stable states are domain-specific.
`modify` (SINGLE-ASSIGNMENT | QUERY | Expression) Description	All	Indicates the semantics of a construct as follows: `SINGLE-ASSIGNMENT` indicates that the construct will be set or operated upon exactly once. Once the variable is set or the feature is called, it will never be set or called again. `QUERY` indicates that the construct is read-only and has no side-effects. `Expression` indicates that a feature modifies an object and describes how it does so.
[label:] `require` (Predicate) [Exception -] [Description]	Features	Describes a precondition that must be true before the feature can be safely invoked. One property is used for each precondition. If a precondition is violated then the *Exception* is the result. The visibility of preconditions follows the same rules as the visibility of postconditions.

Table B.4: Concurrency Properties

Property		Context	Purpose/Description
`concurrency`	(SEQUENTIAL | GUARDED | CONCURRENT | TIMEOUT ⟨value⟩ Exception | FAILURE Exception | SPECIAL) [Description]	Modules, Features	Describes the concurrency strategy and/or approach taken by (necessary for) the module or feature. The execution context for a feature should be described at this point. The meanings of the possible parameters are as follows: SEQUENTIAL means that callers must coordinate so that only one call/access to the object in question may be outstanding at once. If simultaneous calls occur, the the semantics and integrity of the system are not guaranteed. GUARDED means that multiple calls/accesses from concurrent threads may occur simultaneously to one object in question, but only one is allowed to commence; the others are blocked until the performance of the first operation is complete. This is the behavior of synchronized instance methods in Java. CONCURRENT indicates that multiple calls/accesses from concurrent threads may occur simultaneously to the object in question, and that all will proceed concurrently with correct semantics. This is the default behavior of Java methods and fields and Eiffel features. TIMEOUT *value* indicates that if a call to this feature is blocked for a time period greater than or equal to *value*, the exception *Exception* will be raised. The value is specified with units. FAILURE means that if a call to the feature is currently underway, all additional calls with fail and the exception *Exception* will be raised. SPECIAL indicates that the feature has concurrency semantics that are not covered by the preceding cases. Make sure to explain the particulars of the feature's semantics in sufficient detail that the reader will be quite clear on your feature's unusual semantics. A feature lacking a concurrency property is considered CONCURRENT. The semantics description is optional on features that are labeled as SEQUENTIAL or GUARDED. In general, all features should have concurrency properties.

Table B.5: Usage Information Properties

Property	Context	Purpose/Description
`param` parameter-name [WHERE (Expression)] Description	Features	Describes a feature parameter. The description may be continued on the following line(s). The expression indicates restrictions on argument values. Restrictions that result in exceptions, for example IllegalArgumentException in Java, should be indicated as such. In particular, it is important to indicate whether reference arguments are allowed to be null. There must be one param property for each and every parameter to the feature.
`return` Description	Features	Describes a feature's return value. The description may be continued on the following line(s). If the feature being documented is a getter feature (or similar), a return property is still necessary; however, in such a case, the actual feature description may be omitted since the return property completely describes the feature.
`exception` ExceptionName [IF (Expression)] Description	Features	Describes an exception that can be raised by the feature, and the circumstances under which it is raised. The guard indicates restrictions on argument values. In particular, it is important to indicate whether reference arguments are allowed to be null. There must be one exception property for each and every exception declared to be raised by the feature. In Java, for instance, a RuntimeException or IllegalArgumentException declared in the feature signature must be documented with this tag. It is a good idea to declare in the feature signature any runtime exceptions that are raised as a result of conditions controllable at compile-time (such as the feature being called in the wrong system state, or a lack of sufficient range checking on passed argument values).

Table B.6: Version Information Properties

Property	Context	Purpose/Description
`version` version-string	Files, Modules	Denotes the version of the class. The text has no special internal structure. Any construct may contain at most one version property. The version property refers to the version of the software project that contains this class, rather than the version of this particular class.
`deprecated` Reference to replacement API.	All	Indicates that this API element should no longer be used (even though it may continue to work). By convention, the text describes the API (or APIs) that replace the deprecated API. For example: `deprecated Replaced by setBounds(int, int, int, int).` If the API is obsolete and there is no replacement, the argument to deprecated should be "No replacement".
`since` version-string	All	Indicates the (release) version number in which this construct first appeared. This version number, like that for the version property, is a project-wide version number rather than a version number for the particular class in which the feature appears. No since property is required for features that have existed since the first version of the project. Often the cvs tag for a version is used here for the version-string, but this is not a requirement. Note that current versions of cvs do not permit tags with periods (e.g. "0.1.0" is illegal).

Table B.7: Inheritance Properties

Property	Context	Purpose/Description
`hides` FeatureName [Description]	Variables	Indicates that a particular variable hides a variable in a enclosing context (e.g. a parent class).
`overrides` FeatureName [Description]	Features	Indicates that a particular feature overrides a feature in an enclosing context (e.g. a parent class). If the overriding feature does not either call or implement a superset of the semantics of the feature it overrides, its effects must be documented (using precondition, postcondition, and invariant properties and the feature description, as appropriate).

Table B.8: Documentation Properties

Property	Context	Purpose/Description
`design` Description	All	Provides information about design decisions you have made and/or useful things to know about parts of your code.
`equivalent` (Expression | Code Reference)	Modules, Features	Documents convenience or specialized features that can be defined in terms of a few operations using other features.
`example` Description	All	Provides one or more examples of how to use a construct.
`see` APIName Label	All	References the reader to other related material. Some source processors use this information to generate hyperlinked documentation. APIName can be the name of a Language API or an HTML anchor. This property is often used to reference an external document that describes pertinent business rules or information relevant to the source code being documented.

Table B.9: Dependency Properties

Property	Context	Purpose/Description
`references` (Expression) [Description]	Modules, Features, Variables	Indicates that the construct references other constructs like objects, instances, files, etc. This is used to indicate subtle interdependencies between classes, instances, etc.
`use` (Expression) [Description]	All	Indicates exactly those elements that are utilized by this element. This property is used to indicate hidden dependencies.

Table B.10: Miscellaneous Properties

Property	Context	Purpose/Description
`guard` (Expression) [Description]	Features	Indicates that actions use guarded waits (e.g. in `java.lang.Object.wait()`) until the condition specified in Expression holds.
`values` (Expression) [Description]	Variables	Describes the possible values of a variable, including ranges and/or distinct values.
`(time|space)-complexity` (Expression) [Description]	Features	Documents the time or space complexity of a feature. Expression should be in big-O notation. E.g. `O(n*n)` and `O(n^2)` are equivalent, but the second is the more normal way to express a square term. The free terms of the expression should be documented in the description; usually, they are related directly to instance variables of the surrounding or related class.

Appendix C

The Extended BON Grammar

This grammar specification is a corrected and augmented form of that which is available in Walden and Nerson's text [364]. Therefore the lexical notions of identifiers, free operators, comment, strings, etc. are not covered in detail.

A few small corrections were made to the original BON grammar when errors were discovered during the implementation of a BON scanner and parser.

C.1 Grammar Corrections

List Concatenators. The grammar is unclear as to the usage of list concatenators (e.g. ";" in rules like Enumeration_list). This is due to the lack of clarity of the discussion of the ellipses meta-construct ("...") in the grammar specification. This grammar was written such that such concatenators are only used between two terms and not at the end of a term.

Final List Terminators. Likewise, the grammar is unclear with respect to the use of final list terminators for constructs like Index_clause. These final list terminators should be optional. Most of the text examples from the text follow this model.

The problem is that it is unclear what the intention of such terminators is; given the Eiffel influence on BON it would be surprising if any terminators were necessary at all.

Ellipses. While ellipses are used throughout the examples, no keyword or operator for such is specified in the grammar. The token ELLIPSES_TOKEN was added to deal with this situation.

In the original grammar this token is used only in the Indirection_list. It was necessary to add it to the Static_component rule because it is used in this manner in the example specifications. The new rule is

```
Static_component: Cluster_rule
        | Class_rule
```

```
                | Static_relation
                | ELLIPSES_TOKEN ;
```

C.2 Extensions

Not all semantic properties defined in Chapter 7 are appropriate for BON. For example, BON already contains constructs for denoting contracts.

For each property that does add new value to a BON, the decision must be made as to whether the construction should be embedded via a indexing clause, a structured comment, or via grammar augmentation. Given that no tools exist that process the BON textual grammar, the latter option is chosen when at all possible.

Structured comments in EBON are specified with the "properties" keyword which complements the existing "indexing" keyword. This explicit keyword is used to simplify parsing.

C.3 EBON Grammar

C.3.1 Specification Top

```
Bon_specification = { Specification_element ...} +

Specification_element = Informal_chart | Class_dictionary |
                        Static_diagram | Dynamic_diagram |
                        Notational_tuning | SP_file_clause
```

C.3.2 Informal Charts

```
Informal_chart = System_chart | Cluster_chart | Class_chart |
                 Event_chart | Scenario_chart | Creation_chart

Class_dictionary = dictionary System_name
                   { Dictionary_entry ...} +
                   end

Dictionary_entry = class Class_name cluster Cluster_name
                   description Manifest_textblock

System_chart =    system_chart System_name
```

```
                  [ indexing Index_list ]
                  [ properties { SP_system_clause ...} + ]
                  [ explanation Manifest_string ]
                  [ part Manifest_string ]
                  [ Cluster_entries ]
                  end
Cluster_entries = { Cluster_entry ...} +
Cluster_entry =   cluster Cluster_name description Manifest_textblock
System_name =     Identifier

Index_list =      { Index_clause ";" ...} +
Index_clause =    Identifier ":" Index_term_list |
                  SP_module_clause
Index_term_list = { Index_string "," ...} +
Index_string =    Manifest_string

Cluster_chart = cluster_chart Cluster_name
                [ indexing Index_list ]
                [ properties { SP_cluster_clause ...} + ]
                [ explanation Manifest_string ]
                [ part Manifest_string ]
                [ Class_entries ]
                [ Cluster_entries ]
                end
Class_entries = { Class_entry ...} +
Class_entry =   class Class_name description Manifest_textblock
Cluster_name =  Identifier

Class_chart =     class_chart Class_name
                  [ indexing Index_list ]
                  [ properties { SP_module_clause ...} + ]
                  [ explanation Manifest_string ]
                  [ part Manifest_string ]
                  [ inherit Class_name_list ]
                  [ query Query_list ]
                  [ command Command_list ]
```

```
                [ constraint Constraint_list ]
                end
Query_list =    { Manifest_string "," ...} +
Command_list =  { Manifest_string "," ...} +
Constraint_list = { Manifest_string "," ...} +
Class_name_list = { Class_name "," ...} +
Class_name =    Identifier

Event_chart =   event_chart System_name
                [ incoming | outgoing ]
                [ indexing Index_list ]
                [ properties { SP_module_clause ...} + ]
                [ explanation Manifest_string ]
                [ part Manifest_string ]
                [ Event_entries ]
                end
Event_entries = { Event_entry ...} +
Event_entry =   event Manifest_string involves Class_name_list

Scenario_chart =   scenario_chart System_name
                   [ indexing Index_list ]
                   [ properties { SP_module_clause ...} + ]
                   [ explanation Manifest_string ]
                   [ part Manifest_string ]
                   [ Scenario_entries ]
                   end
Scenario_entries = { Scenario_entry ...} +
Scenario_entry =   scenario Manifest_string description Manifest_textblock

Creation_chart =   creation_chart System_name
                   [ indexing Index_list ]
                   [ properties { SP_module_clause ...} + ]
                   [ explanation Manifest_string ]
                   [ part Manifest_string ]
                   [ Creation_entries ]
                   end
```

```
Creation_entries = { Creation_entry ...} +
Creation_entry =   creator Class_name creates Class_name_list
```

C.3.3 Static Diagrams

```
Static_diagram =     static_diagram [ Extended_id ] [ Comment ]
                     component Static_block end
Extended_id =        Identifier | Integer
Comment =            { [ Line_comment | Ebon_comment ] New_line ...} +
Line_comment =       "--" Simple_string
Ebon_comment =       "--*" Semantic_properties "*--"
Static_block =       { Static_component ...}
Static_component =   Cluster | Class | Static_relation | "..."

Cluster =            cluster Cluster_name
                     [ reused ] [ Comment ]
                     [ Cluster_components ]
Cluster_components = component Static_block end
Class =              [ root | deferred | effective ]
                     class Class_name [ Formal_generics ]
                     [ reused ][ persistent ][ interfaced ] [ Comment ]
                     [ Class_interface ]
Static_relation =    Inheritance_relation | Client_relation

Inheritance_relation =     Child inherit [ "{" Multiplicity "}" ]
                           Parent [ Semantic_label ]
Client_relation =          Client client [ Client_entities ] [ Type_mark ]
                           Supplier [ Semantic_label ]
Client_entities =          "{" Client_entity_expression "}"
Client_entity_expression = Client_entity_list | Multiplicity
Client_entity_list =       { Client_entity "," ...} +
Client_entity =            Feature_name | Supplier_indirection |
                           Parent_indirection
Supplier_indirection =     [ Indirection_feature_part ":" ]
                           Generic_indirection
Indirection_feature_part = Feature_name | Indirection_feature_list
```

```
Indirection_feature_list = "(" Feature_name_list ")"
Parent_indirection =       "->" Generic_indirection

Generic_indirection = Formal_generic_name | Named_indirection
Named_indirection =   Class_name "[" Indirection_list "]"
Indirection_list =    { Indirection_element "," ...} +
Indirection_element = "..." | Named_indirection
Type_mark =           ":" | ":{" | Shared_mark
Shared_mark =         ":" "(" Multiplicity ")"

Child =                 Static_ref
Parent =                Static_ref
Client =                Static_ref
Supplier =              Static_ref
Static_ref =            { Cluster_prefix ...} Static_component_name
Cluster_prefix =        Cluster_name "."
Static_component_name = Class_name | Cluster_name
Multiplicity =          Integer
Semantic_label =        Manifest_string
```

C.3.4 Class Interface Description

```
Class_interface =   [ indexing Index_list ]
                    [ properties { SP_module_clause ...} + ]
                    [ inherit Parent_class_list ]
                    Features
                    [ invariant Class_invariant ]
                    end
Class_invariant =   Assertion
Parent_class_list = { Class_type ";" ...} +
Features =          { Feature_clause ...} +

Feature_clause =         feature [ Selective_export ]
                         [ Comment ]
                         [ SP_feature_clauses ]
                         Feature_specifications
```

```
Feature_specifications = { Feature_specification ...} +
Feature_specification =  [ deferred | effective | redefined ]
                         Feature_name_list [ Type_mark Type ]
                         [ Rename_clause ]
                         [ Comment ]
                         [ Param_comments ]
                         [ Feature_arguments ]
                         [ Contract_clause ]

Contract_clause =        Contracting_conditions end
Contracting_conditions = Precondition | Postcondition | Pre_and_post |
                         Generate_clause | Modify_clause | Concurrency_clause
Precondition =           require Assertion
Postcondition =          ensure Assertion
Pre_and_post =           Precondition Postcondition

Selective_export =  "{" Class_name_list "}"
Feature_name_list = { Feature_name "," ...} +
Feature_name =      Identifier | Prefix | Infix
Rename_clause =     "{" Renaming "}"
Renaming =          "^" Class_name "." Feature_name
Feature_arguments = { Feature_argument ...} +
Feature_argument =  "->" [ Identifier_list ":" ] Type
Identifier_list =   { Identifier "," ...} +
Prefix =            prefix '"'  Prefix_operator '"'
Infix =             infix '"' Infix_operator '"'
Prefix_operator =   Unary | Free_operator
Infix_operator =    Binary | Free_operator

Param_comments =    { Param_clause New_line ...} +

Formal_generics =     "[" Formal_generic_list "]"
Formal_generic_list = { Formal_generic "," ...} +
Formal_generic =      Formal_generic_name [ "->" Class_type ]
Formal_generic_name = Identifier
Class_type =          Class_name [ Actual_generics ]
```

```
Actual_generics =     "[" Type_list "]"
Type_list =           { Type "," ...} +
Type =                Class_type | Formal_generic_name

Unary =  delta | old | not | "+" | "-"
Binary = "+" | "-" | "*" | "/" |
         "<" | ">" | "<=" | ">=" |
         "=" | "/=" | "//" | "\\" | "^" |
         or | xor | and | "->" | "<->" | member_of | ":"
```

C.3.5 Formal Assertions

```
Assertion =          { Assertion_clause ";" ...} +
Assertion_clause =   Boolean_expression | Comment
Boolean_expression = Expression
Expression =         Quantification | Call | Operator_expression | Constant
Quantification =     Quantifier Range_expression [ Restriction ] Proposition
Quantifier =         for_all | exists
Range_expression =   { Variable_range ";" ...} +
Restriction =        such_that Boolean_expression
Proposition =        it_holds Boolean_expression
Variable_range =     Member_range | Type_range
Member_range =       Identifier_list member_of Set_expression
Type_range =         Identifier_list ":" Type

Call =                    [ Parenthesized_qualifier ] Call_chain
Parenthesized_qualifier = Parenthesized "."
Call_chain =              { Unqualified_call "." ...} +
Unqualified_call =        Identifier [ Actual_arguments ]
Actual_arguments =        "(" Expression_list ")"
Expression_list =         { Expression "," ...} +
Operator_expression =     Parenthesized | Unary_expression |
                          Binary_expression
Parenthesized =           "(" Expression ")"

Unary_expression =    Prefix_operator Expression
```

```
Binary_expression =   Expression Infix_operator Expression
Set_expression =      Enumerated_set | Call | Operator_expression
Enumerated_set =      "{" Enumeration_list "}"
Enumeration_list =    { Enumeration_element "," ...} +
Enumeration_element = Expression | Interval
Interval =            Integer_interval | Character_interval
Integer_interval =    Integer_constant ".." Integer_constant
Character_interval =  Character_constant ".." Character_constant

Constant =           Manifest_constant | Current | Void
Manifest_constant =  Boolean_constant | Character_constant |
                     Integer_constant | Real_constant |
                     Manifest_string
Sign =               "+" | "-"
Boolean_constant =   true | false
Character_constant = "'" Character "'"
Integer_constant =   [ Sign ] Integer
Real_constant =      [ Sign ] Real
Manifest_textblock = String_begin String String_end
String =             { Simple_string New_line ...} +
Manifest_string =    String_begin Simple_string String_end
```

C.3.6 Dynamic Diagrams

```
Dynamic_diagram =   dynamic_diagram [ Extended_id ] [ Comment ]
                    component Dynamic_block end
Dynamic_block =     { Dynamic_component ...}
Dynamic_component = Scenario_description |
                    Object_group |
                    Object_stack |
                    Object |
                    Message_relation

Scenario_description = scenario Scenario_name [ Comment ]
                       action Labeled_actions end
Labeled_actions =      { Labeled_action ...} +
```

```
Labeled_action =       Action_label Action_description
Action_label =         Manifest_string
Action_description =   Manifest_textblock
Scenario_name =        Manifest_string

Object_group =     [ nameless ] object_group Group_name [ Comment ]
                   [ Group_components ]
Group_components = component Dynamic_block end
Object_stack =     object_stack Object_name [ Comment ]
Object =           object Object_name [ Comment ]

Message_relation =       Caller calls Receiver [ Message_label ]
Caller =                 Dynamic_ref
Receiver =               Dynamic_ref
Dynamic_ref =            { Group_prefix ...} Dynamic_component_name
Group_prefix =           Group_name "."
Dynamic_component_name = Object_name | Group_name
Object_name =            Class_name [ "." Extended_id ]
Group_name =             Extended_id
Message_label =          Manifest_string
```

C.3.7 Notational Tuning

```
Notational_tuning =   Change_string_marks |
                      Change_concatenator |
                      Change_prefix
Change_string_marks = string_marks Manifest_string Manifest_string
Change_concatenator = concatenator Manifest_string
Change_prefix =       keyword_prefix Manifest_string
```

C.3.8 Semantic Properties

```
Semantic_properties = { Semantic_property New_line ...} +
Semantic_property =   SP_all_clause

Author_clause = author ":" Author_fullname [ Email_address ]
Author_fullname = Manifest_string
```

```
Email_address = Manifest_string
Bug_clause = bug ":" Manifest_string
Copyright_clause = copyright ":" Manifest_string
Description_clause = description ":" Manifest_string
History_clause = history ":" Manifest_string
License_clause = license ":" Manifest_string
Title_clause = title ":" Manifest_string

Idea_clause = idea ":" Author_clause [ Classifier_clause ] - Manifest_string
Author_clause = Manifest_string
Classifier_clause = Manifest_string
Review_clause = review ":" Author_clause - Manifest_string
Todo_clause = todo ":" Author_clause - Manifest_string

Generate_clause = generate EBON_Expression [ Manifest_string ]
Modify_clause = modify Modification_type Manifest_string
Modification_type = SINGLE_ASSIGNMENT | QUERY | EBON_Expression

Concurrency_clause = concurrency Concurrency_type [ Manifest_string ]
Concurrency_type = SEQUENTIAL | GUARDED | CONCURRENT |
                   TIMEOUT Real_constant EBON_Exception |
                   FAILURE EBON_Exception |
                   SPECIAL
EBON_Exception = Call

Param_clause = param Feature_name [ WHERE EBON_Expression ] Manifest_string
Return_clause = return Manifest_string
Exception_clause = Exception_name [ IF EBON_Expression ] Manifest_string
Exception_name = Identifier

Version_clause = version Version_string
Version_string = Manifest_string
Deprecated_clause = deprecated Manifest_string
Since_clause = since Version_string

Hides_clause = hides Feature_name [ Manifest_string ]
```

```
Overrides_clause = overrides Feature_name [ Manifest_string ]

Design_clause = design Manifest_string
Equivalent_clause = equivalent EBON_Expression
Example_clause = example Manifest_string
See_clause = see Class_name "." Feature_name

References_clause = references EBON_Expression [ Manifest_string ]
Use_clause = use EBON_Expression [ Manifest_string ]

Guard_clause = guard EBON_Expression [ Manifest_string ]
Values_clause = values EBON_Expression [ Manifest_string ]
Complexity_clause = Complexity_type EBON_Expression [ Manifest_string ]
Complexity_type = time-complexity | space-complexity

SP_system_clause = [ SP_file_clause | SP_module_clause ]
SP_cluster_clause = SP_system_clause
SP_feature_clauses = { SP_feature_clause ... } +

SP_file_clause = Copyright_clause | Description_clause | License_clause |
                 Title_clause | Idea_clause | Review_clause | Todo_clause

SP_module_clause = Author_clause | Bon_clause | Bug_clause | History_clause |
                   Idea_clause | Review_clause | Todo_clause

SP_feature_clause = Bug_clause | History_clause | Idea_clause |
                    Review_clause | Todo_clause

SP_all_clause = SP_file_clause | SP_module_clause | SP_feature_clause
```

C.4 Reserved Words

The original BON reserved words are as follows.

The new reserved words that have been added to BON are as follows.

Table C.1: BON Reserved Words

action	creator	false	not	reused
and	Current	feature	object	root
calls	deferred	for_all	object_group	scenario
class	delta	incoming	object_stack	scenario_chart
class_chart	description	indexing	old	static_diagram
client	dictionary	infix	or	string_marks
cluster	dynamic_diagram	inherit	outgoing	such_that
cluster_chart	effective	interfaced	part	system_chart
command	end	invariant	persistent	true
component	ensure	involves	prefix	Void
concatenator	event	it_holds	query	xor
constraint	event_chart	keyword_prefix	redefined	
creates	exists	member_of	require	
creation_chart	explanation	nameless	Result	

Table C.2: New EBON Reserved Words

author	example	overrides	title
bon	exception	param	todo
bug	generate	references	use
concurrency	guard	require	values
copyright	hides	return	version
deprecated	history	review	properties
description	idea	see	
design	invariant	since	
ensure	license	space-complexity	
equivalent	modifies	time-complexity	

Bibliography

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer–Verlag, 1996.

[2] ACM SIGPLAN: Programming Languages. *The First Conference on Object Oriented Programming, Systems, Languages, and Applications*. ACM Press and Addison-Wesley Publishing Company, 1986.

[3] ACM SIGPLAN: Programming Languages. *The Second Conference on Object Oriented Programming, Systems, Languages, and Applications*. ACM Press and Addison-Wesley Publishing Company, 1987.

[4] ACM SIGPLAN: Programming Languages. *The Eighth Conference on Object Oriented Programming, Systems, Languages, and Applications*. ACM Press and Addison-Wesley Publishing Company, 1993.

[5] ACM SIGPLAN: Programming Languages. *The Tenth Conference on Object Oriented Programming, Systems, Languages, and Applications*. ACM Press and Addison-Wesley Publishing Company, 1995.

[6] Helmut Adametz. Semantics of the type view and the type configuration of the π language. Technical report, Fraunhofer Institute of Software and System Technology, Berlin, Germany, 1992.

[7] S. Agerholm and M. Gordon. Experiments with ZF set theory in HOL and isabelle. In E. T. Schubert, P. J. Windley, and J. Alves-Foss, editors, *Proceedings of the Eighth International Workshop on Higher Order Logic Theorem Proving And Its Applications*, pages 32–45. Springer–Verlag, 1995.

[8] S. Aguzzoli, A. Ciabattoni, and A. Di Nola. Sequent calculi for finite-valued Lukasiewicz logics via boolean decompositions. *Journal of Logic and Computation*, 10(2):213–222, April 2000.

[9] W. Aitken, B. Dickens, P. Kwiatkowski, O. De Moor , D. Richter, and C. Simonyi. Transformation in intentional programming. In *Proceedings of the Fifth International Conference*

on Software Reuse, pages 114–123. IEEE Computer Society Technical Committee on Software Reuse and the ACM, 1998.

[10] R. Allen and D. Garlan. Beyond definition use—architectural interconnection. *ACM SIGPLAN Notices*, 29(8):35–44, 1994.

[11] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.

[12] J. Alves-Foss, editor. *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*. Springer–Verlag, 1999.

[13] S. Amoroso. A mathematical method for system conceptualization. In *IEEE Workshop on Languages for Automation: Cognitive Aspects In Information Processing*, pages 25–34. IEEE Computer Society, 1985.

[14] A. Anderson, N. Belnap, and M. Dunn. *Entailment. The Logic of Relevant and Necessity*, volume 2. Princeton University Press, 1992.

[15] W. F. Appelbe and A. P. Ravn. Encapsulation constructs in systems programming languages. *ACM Transactions on Programming Languages and Systems*, 6(2):129–158, April 1984.

[16] Guillermo Arango. *Software Reusability*, chapter Domain Analysis Methods, pages 17–49. Ellis Horwood Workshop Series. Ellis Horwood, 1994.

[17] M. Arbib and E. Manes. *Arrows, Structures, and Functors: The Categorical Imperative*. Academic Press, 1975.

[18] Aristotle. *The Organon*. Harvard University Press, 1938.

[19] Aristotle. *Aristotle's Prior and Posterior Analytics*. Garland Publishers, 1980. Edited by W. D. Ross.

[20] D. M. Armstrong. *Belief Truth and Knowledge*. Cambridge University Press, 1973.

[21] M. Arrais and J. L. Fiadeiro. *Unifying Theories in Different Institutions*, volume 1130 of *Lecture Notes in Computer Science*, pages 81–101. Springer–Verlag, 1996.

[22] H. Astudillo. Reorganizing split objects. *ACM SIGPLAN Notices*, 31(10):138–149, 1996.

[23] S. Atkinson. Modelling formal integrated component retrieval. In *Proceedings of the Fifth International Conference on Software Reuse*. IEEE Computer Society Technical Committee on Software Reuse and the ACM, 1998.

[24] Robert Axelrod. *The Complexity of Cooperation: Agent-Based Models of Competition and Collaboration*. Princeton University Press, 1997.

[25] D. Bardou and C. Dony. Split objects: A disciplined use of delegation within objects. *ACM SIGPLAN Notices*, 31(10):122–137, 1996.

[26] H. P. Barendregt. *Handbook of Logic in Computer Science*, chapter Lambda Calculus with Types.

[27] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Company, 1985.

[28] H. P. Barendregt. *The Lambda Calculus*. Number 103 in Studies in Logic and the Foundations of Mathematics. North-Holland Publishing Company, revised edition, 1991.

[29] D. R. Barstow. An experiment in knowledge-based automatic programming. *Artificial Intelligence*, 12(2):73–119, 1979.

[30] D. R. Barstow. Domain-specific automatic programming. *IEEE Transactions on Software Engineering*, 11(11):1321–1336, November 1985.

[31] D. Batens. *Paraconsistent Logic: Essays on the Inconsistent*, chapter Dynamic Dialectical Logics, pages 187–217. Philosophia, 1989.

[32] D. Batens. *Logic at Work: Essays dedicated to the memory of Helena Rasiowa*, chapter Inconsistency-Adaptive Logics, pages 445–472. Physica Verlag (Springer), 1998.

[33] Diderik Batens. *Frontiers of Paraconsistent Logic*. Taylor and Francis, Inc., 2000.

[34] Don Batory. Compositional validation and subjectivity and GenVoca generators. In *Proceedings of the Fourth International Conference on Software Reuse*, pages 166–175, 1996. Is sometimes titled "Subjectivity and Software System Generators".

[35] Don Batory and Bart J. Geraci. Validating component compositions in software system generators. In *Proceedings of the Fourth International Conference on Software Reuse*, pages 72–81, 1996.

[36] T. Bench-Capon and G. Malcolm. *Formalising Ontologies and Their Relations*, volume 1677 of *Lecture Notes in Computer Science*, pages 250–259. Springer–Verlag, 1999.

[37] Fran Berman. From TeraGrid to knowledge grid. *Communications of the ACM*, 44(11):27–28, November 2001.

[38] Mishkin Berteig. OOMind: Open education community. http://www.oomind.com/, April 2001.

[39] P. Besnard and A. Hunter. *Handbook of Defeasible Reasoning and Uncertainty Management Systems*, volume 2, chapter Reasoning with Actual and Potential Contradictions. Kluwer Academic Publishing, 1998.

[40] Abhay Bhorkar. A run-time assertion checker for Java using JML. Technical Report 00-08, Department of Computer Science, Iowa State University, May 2000.

[41] T. J. Biggerstaff. The library scaling problem and the limits of concrete component reuse. In *Proceedings of the Third International Conference on Software Reuse*, pages 102–109, 1994.

[42] Ted J. Biggerstaff and Alan J. Perlis, editors. *Software Reusability*, volume I: Concepts and Models of *Frontier Series*. ACM Press, 1989.

[43] Ted J. Biggerstaff and Alan J. Perlis, editors. *Software Reusability*, volume II: Applications and Experience of *Frontier Series*. ACM Press, 1989.

[44] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley Publishing Company, 1994.

[45] A. H. Borning. Classes versus prototypes in object-oriented languages. In *Proceedings of the 1986 Fall Joint Computer Conference*, pages 36–40. ACM Press, 1986.

[46] Rod Burstall and J. Goguen. The semantics of Clear, a specification language. In D. Bjorner, editor, *Proceedings of the 1979 Copenhagen Winter School on Abstract Software Specification*, volume 86 of *Lecture Notes in Computer Science*, pages 292–332. Springer–Verlag, 1980.

[47] Luca Cardelli. *Semantics of Data Types*, chapter A Semantics of Multiple Inheritance, pages 51–67. Springer–Verlag, 1984.

[48] Luca Cardelli. Typeful programming. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, IFIP State of the Art Reports Series. Springer–Verlag, February 1989.

[49] G. Caronni. Walking the web of trust. In *Proceedings IEEE Ninth International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2000)*, pages 153–158. IEEE Computer Society, 2000.

[50] Guiseppe Castagna. Covariance and contravariance: Conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, May 1995.

[51] S. Castano and V. De Antonellis. A constructive approach to reuse of conceptual components. In *Proceedings of the Second International Workshop on Software Reusability*, pages 19–28, 1993.

[52] T. Catlin, P. Bush, and N. Yankelovich. Internote: Extending a hypermedia framework to support annotative collaboration. *SIGCHI Bulletin*, pages 365–378, 1989.

[53] Maura Cerioli and José Meseguer. May I Borrow Your Logic? (transporting logical structures along maps). *Theoretical Computer Science*, 173(2):311–347, 1997.

[54] K. Mani Chandy and Michel Charpentier. An experiment in program composition and proof. *Formal Methods in System Design*, April 1999. Accepted for publication.

[55] K. Mani Chandy, Joseph R. Kiniry, Adam Rifkin, and Daniel M. Zimmerman. Webs of archived distributed computations for asynchronous collaboration. *Journal of Supercomputing*, 11, 1997.

[56] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley Publishing Company, 1988.

[57] K. Mani Chandy and Adam Rifkin. Systematic composition of objects in distributed internet applications: Processes and sessions. In *Hawaii International Conference on System Sciences*, pages 63–75. IEEE Computer Society, January 1997.

[58] K. Mani Chandy and Beverly Sanders. Reasoning about program composition. Technical Report TR96-035, University of Florida, November 1996.

[59] C. C. Chang and H. J. Keisler. *Model Theory*, volume 73 of *Studies in Logic and the Foundations of Math*. North-Holland Publishing Company, third edition, 1990.

[60] Michel Charpentier and K. Mani Chandy. Examples of program composition illustrating the use of universal properties. In J. Rolim, editor, *International Workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA'99)*, volume 1586 of *Lecture Notes in Computer Science*, pages 1215–1227. Springer–Verlag, April 1999.

[61] Michel Charpentier and K. Mani Chandy. Towards a compositional approach to the design and verification of distributed systems, September 1999.

[62] L. Chaudron and N. Maille. First order logic formal concept analysis: From logic programming to theory, 1998.

[63] Jianhua Chen. The generalized logic of only knowing (GOL) that covers the notion of epistemic specifications. *Journal of Logic and Computation*, 7(2):159–170, April 1997.

[64] P. Chen, R. Hennicker, and M. Jarke. On the retrieval of reusable software components. In *Proceedings of the Second International Workshop on Software Reusability*, pages 99–108, 1993.

[65] Bart Childs and Johannes Sametinger. Literate programming and documentation reuse. In *Proceedings of the Fourth International Conference on Software Reuse*, pages 205–214, 1996.

[66] Roderick M. Chisholm. *Theory of Knowledge*. Prentice-Hall, Inc., second edition, 1977.

[67] Roderick M. Chisholm. *The Foundations of Knowing*. University of Minnesota Press, 1982.

[68] Franco Civello. Roles for composite objects in object-oriented analysis and design. In OOPSLA-93 [4], pages 376–393.

[69] Conceptual knowledge markup language (CKML) DTD. Available via http://www.ontologos.org/.

[70] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José Quesada. Maude as a metalanguage. In *Second International Workshop on Rewriting Logic and its Applications*, Electronic Notes in Theoretical Computer Science. Elsevier Science, Inc., 1998.

[71] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José Quesada. A maude tutorial. Available at the Maude web site http://maude.csl.sri.com/, March 2000.

[72] Manuel Clavel, Francisco Durán, Steven Eker, José Meseguer, and Mark-Oliver Stehr. Maude as a formal meta-tool. In *Proceedings of the World Congress on Formal Methods in the Development of Computing Systems*, 1999.

[73] Manuel Clavel, Steven Eker, Patrick Lincoln, and José Meseguer. Principles of maude. In *In Proceedings, First International Workshop on Rewriting Logic and its Applications*, Electronic Notes in Theoretical Computer Science. Elsevier Science, Inc., 1996.

[74] J. C. Cleaveland. Building application generators. *IEEE Software*, 5(4):25–33, July 1988.

[75] Peter Coad and Edward Yourdon. *OOA: Object-Oriented Analysis, Second Edition*. Prentice-Hall, Inc., 1990.

[76] Peter Coad and Edward Yourdon. *OOD: Object-Oriented Design*. Prentice-Hall, Inc., 1990.

[77] Paul J. Cohen. A minimal model for set theory. *Bulletin of the American Mathematical Society*, 69:537–540, 1963.

[78] S. Cohen and L. M. Northrop. Object-oriented technology and domain analysis. In *Proceedings of the Fifth International Conference on Software Reuse*. IEEE Computer Society Technical Committee on Software Reuse and the ACM, 1998.

[79] Derek Coleman et al. *Object-Oriented Development: The Fusion Method*. Prentice-Hall, Inc., 1993.

[80] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Computer Science Department, Cornell University, 1995.

[81] W. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. In *Proceedings of the Fourth Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 433–443. ACM SIGPLAN: Programming Languages, ACM Press and Addison-Wesley Publishing Company, 1989.

[82] D. E. Cooke. Issues in CASE technology transfer. In *Fourth International Workshop on Computer-Aided Software Engineering*, pages 78–9, 1990.

[83] D. E. Cooke. Towards a formalism to produce a programmer assistant CASE tool. *IEEE Transactions on Knowledge and Data Engineering*, 2(3):320–326, September 1990.

[84] D. E. Cooke. An issue of the next generation of problem-solving environments. *Journal of Systems Integration*, (1):39–52, February 1992.

[85] D. E. Cooke. Issues surrounding specification languages for software automation. In *Proceedings, Fifth International Workshop on Computer-Aided Software Engineering*, pages 120–123, 1992.

[86] D. E. Cooke, R. Duran, A. Gates, and V. Kreinovich. Bag languages, concurrency, horn logic programs, and linear logic. In *SEKE '94. The Sixth International Conference on Software Engineering and Knowledge Engineering*, pages 289–297. Knowledge Syst. Inst., 1994.

[87] D. E. Cooke and A. Gates. On the development of a method to synthesize programs from requirements specifications. *International Journal of Software Engineering and Knowledge Engineering*, 1(1):21–38, March 1991.

[88] D. E. Cooke, V. Kreinovich, and S. A. Starks. ALPS: A logic for program synthesis (motivated by fuzzy logic). In *1998 IEEE International Conference on Fuzzy Systems Proceedings*, pages 779–784, 1998.

[89] Microsoft Corp. The .NET common language runtime, 2001. See website at http://msdn.microsoft.com/net/.

[90] Rational Software Corporation et al. *UML Notation Guide, version 1.1*. The UML 1.1 Consortium, September 1997.

[91] Rational Software Corporation et al. *UML Semantics, version 1.1*. The UML 1.1 Consortium, September 1997.

[92] Rational Software Corporation et al. UML summary, version 1.1. Technical report, The UML 1.1 Consortium, September 1997.

[93] J. F. Costa, A. Sernadas, and C. Sernadas. Object inheritance beyond subtyping. *Acta Informatica*, 31(1):5–26, 1994.

[94] J. Cramer, W. Fey, M. Goedicke, and M. Grosse-Rhode. Towards a formally based component description language — a foundation for reuse. *Lecture Notes in Computer Science*, 494:358–378, 1991.

[95] Karl Crary, Robert Harper, and Sidd Puri. What is a recursive module? *ACM SIGPLAN Notices*, 34(5):60–63, May 1999.

[96] N. C. A. da Costa. *Sistemas Formais Inconsistentes*. PhD thesis, Universidade Federal do Parana, 1963.

[97] C. Damasio and L. Pereira. *Handbook of Defeasible Reasoning and Uncertainty Management Systems*, chapter A Survey of Paraconsistent Semantics for Logic Programs, pages 241–320. Kluwer Academic Publishing, 1998.

[98] P. Devanbu, R. J. Brachman, P. G. Sefridge, and B. W. Ballard. LaSSIE: A knowledge-based software information system. *Communications of the ACM*, 34(5):34–49, 1991.

[99] Dist-Obj FAQ and mailing list archives. Available via http://www.distributedcoalition.org/. A community of researchers discussing distributed object computing issues of the 21st century.

[100] The DMOZ open directory project. http://dmoz.org/.

[101] Christophe Dony, Jacques Malenfant, and Pierre Cointe. Prototype-based languages: From a new taxonomy to constructive proposals and their validation. In *Proceedings of the Seventh Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 201–217. ACM SIGPLAN: Programming Languages, ACM Press and Addison-Wesley Publishing Company, 1992.

[102] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner. *The Coq Proof Assistant User's Guide*. INRIA, Rocquencourt, France, rapport techniques 154 edition, 1993.

[103] Desmond D'Souza and Alan Wills. *Objects, Components, and Frameworks with UML: the Catalysis Approach*. Addison-Wesley Publishing Company, 1998.

[104] Francisco Durán and José Meseguer. The maude specification of full maude. Technical report, SRI International Computer Science Laboratory, May 1999.

[105] Liesbeth Dusink and Jan van Katwijk. Reuse dimensions. In *Proceedings of SSR '95*, 1995.

[106] Steve Easterbrook, John Callahan, and Virginie Wiels. V & V through inconsistency tracking and analysis. In *Proceedings of the Ninth International Workshop on Software Specification and Design*, 1998.

[107] Umberto Eco. *A Theory of Semiotics*. Indiana University Press, 1982.

[108] G. Wagner (editor). Handling inconsistency in knowledge systems. *Journal of Applied Non-Classical Logics*, 7(1–2), 1997.

[109] Stephen Edwards. Common interface models for components are necessary to support composability. In WISR4 [381].

[110] Peter W. Eklund, Gerard Ellis, and Graham Mann, editors. *Conceptual Structures: Knowledge Representation As Interlingua: Fourth International Conference On Conceptual Structures, ICCS'96, Sydney, Australia, August 19-22, 1996, Proceedings*, volume 1115 of *Lecture Notes in Computer Science*. Springer–Verlag, August 1996.

[111] Carl Ellison and Bruce Schneier. Ten risks of PKI: What you're not being told about public key infrastructure. http://www.counterpane.com/pki-risks-ft.txt.

[112] Interactive Software Engineering. EiffelStudio: A guided tour, 2001. Details available via http://www.eiffel.com/doc/online/eiffel50/intro/studio/.

[113] David Epstein, John Motil, and Joseph R. Kiniry. JJ, can a beginners language include design by contract? *Dr. Dobbs Journal*, April 2000.

[114] ESSLLI-02. *Foundations for the Semantic Web*, August 2002.

[115] R. Fagin, J. Y. Halpern, and M. Y. Vardi. A model-theoretic analysis of knowledge. *Journal of the Association of Computing Machinery*, 38(2):382–428, April 1991.

[116] A. Farquhar, R. Fikes, W. Pratt, and J. Rice. Collaborative ontology construction for information integration, 1995.

[117] S. Ferré and O. Ridoux. A logical generalization of formal concept analysis, 2000.

[118] J. L. Fiadeiro and T. Maibaum. A mathematical toolbox for the software architect. In *Proceedings of the Eighth International Workshop on Software Specification and Design (IWSSD8 – '96)*, 1996.

[119] R. Findler and M. Felleisen. Contract soundness for object-oriented languages. In *Proceedings of Sixteenth International Conference Object-Oriented Programming, Systems, Languages, and Applications*, 2001.

[120] Tim Finin et al. DRAFT specification of the KQML agent-communication language. Technical report, The DARPA Knowledge Sharing Initiative External Interfaces Working Group, June 1993.

[121] Donald G. Firesmith, Brian Henderson-Sellers, and Ian Graham. *OPEN Modeling Language (OML) Reference Manual*. Cambridge University Press, 1998.

[122] G. Fischer, R. McCall, and A. Morch. JANUS: Integrating hypertext with a knowledge-based design environment. *SIGCHI Bulletin*, pages 105–117, 1989.

[123] Richard Forno and William Feinbloom. A matter of trusting trust: Why current public-key infrastructures are a house of cards. http://www.infowarrior.org/articles/2001-01.html, March 2001.

[124] A. Fuhrmann. *Studies in Logic, Language and Information*, chapter An Essay on Contraction. CSLI, Stanford University, 1997.

[125] A. Fuhrmann. When hyperpropositions meet. *Journal of Philosophical Logic*, 28:1–16, 1999.

[126] Dov M. Gabbay, Ian Hodkinson, and Mark Reynolds. *Temporal Logic: Mathematical Foundations and Computational Aspects*. Oxford University Press, 1994.

[127] Dov M. Gabbay and Philippe Smets, editors. *Handbook of Defeasible Reasoning and Uncertainty Management Systems Volume 1: Quantified Representation of Uncertainty and Imprecision*. Kluwer Academic Publishing, 1998.

[128] Dov M. Gabbay and Philippe Smets, editors. *Handbook of Defeasible Reasoning and Uncertainty Management Systems Volume 2: Reasoning with Actual and Potential Contradictions*. Kluwer Academic Publishing, 1998.

[129] Dov M. Gabbay and Philippe Smets, editors. *Handbook of Defeasible Reasoning and Uncertainty Management Systems Volume 3: Belief Change*. Kluwer Academic Publishing, 1998.

[130] Dov M. Gabbay and Philippe Smets, editors. *Handbook of Defeasible Reasoning and Uncertainty Management Systems Volume 4: Abductive Reasoning and Learning*. Kluwer Academic Publishing, 1998.

[131] Dov M. Gabbay and Philippe Smets, editors. *Handbook of Defeasible Reasoning and Uncertainty Management Systems Volume 5: Algorithms for Uncertainty and Defeasible Reasoning*. Kluwer Academic Publishing, 1998.

[132] P. Gabriel. The object-based specification language π – concepts, syntax, and semantics. *Lecture Notes in Computer Science*, 655:254–270, 1993.

[133] Peter Gabriel. Specification of a computational model for π. Technical report, Fraunhofer Institute of Software and System Technology, Berlin, Germany, 1992.

[134] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, 1995.

[135] Anand Ganapathy. Design and implementation of a JML type checker. Master's thesis, Iowa State University, 1999.

[136] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer–Verlag, 1999.

[137] Peter Gärdenfors. *Conceptual Spaces: The Geometry of Thought*. The MIT Press, 2000.

[138] David Garlan. Higher-order connectors. In *Proceedings of Workshop on Compositional Software Architectures*, January 1998.

[139] David Garlan, R. Allen, and John Ockerbloom. Architectural mismatch, or, why it's hard to build systems out of existing parts. In *International Conference on Software Engineering*. IEEE Computer Society, IEEE Computer Society, May 1995.

[140] R. J. Gautier, H. E. Oliver, M. Ratcliffe, and B. R. Whittle. CDL – a component description language for reuse. *International Journal of Software Engineering and Knowledge Engineering*, 3(4):499–518, December 1993.

[141] D. E. Gay. Interface definition language conversions – recursive types. *ACM SIGPLAN Notices*, 29(8):101–110, 1994.

[142] M. A. Gisi and C. Sacchi. A positive experience with software reuse supported by a software bus framework. In *Proceedings of the Second International Workshop on Software Reusability*, pages 196–203, 1993.

[143] R. Godin, H. Mili, G. W. Mineau, R. Missaoui, A. Arfi, and T. T. Chau. Design of class hierarchies based on concept (Galois) lattices. *Theory And Practice of Object Systems*, 4(2):117–134, 1998.

[144] R. Godin, G. Mineau, R. Missaoui, M. Stgermain, and N. Faraj. Applying concept-formation methods to software reuse. *International Journal of Software Engineering And Knowledge Engineering*, 5(1):119–142, 1995.

[145] R. Godin, R. Missaoui, and H. Alaoui. Incremental concept-formation algorithms based on Galois (concept) lattices. *Computational Intelligence*, 11(2):246–267, 1995.

[146] Michael Goedicke, Wolfgang Ditt, and Herbert Schippers. The π-language reference manual. Technical Report 295/1989, University of Dortmund, Department of Computer Science, 1989.

[147] J. Goguen. Parameterized programming. *IEEE Transactions on Software Engineering*, 10(5):528–543, 1983.

[148] J. Goguen, C. Kirchner, H. Kirchner, A. Megrelis, J. Meseguer, and T. Winkler. An introduction to OBJ3. *Lecture Notes in Computer Science*, 308:258–263, 1988.

[149] J. Goguen and T. Winkler. Introducing OBJ3 . Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, 1988.

[150] Joseph Goguen, Doan Nguyen, José Meseguer, Luqi, Du Zhang, and Valdis Berzins. Software component search. *Journal of Systems Integration*, 6:93–134, 1996.

[151] Joseph A. Goguen. Reusing and interconnecting softare components. *IEEE Computer*, 19(2):16–28, February 1986.

[152] Joseph A. Goguen. An introduction to algebraic semiotics, with applications to user interface design. In Chrystopher Nehaniv, editor, *Computation for Metaphor, Analogy and Agents*, volume 1562 of *Springer Lecture Notes in Artificial Intelligence*, pages 242–291. Springer–Verlag, 1999.

[153] Joseph A. Goguen. Social and semiotic analyses for theorem prover user interface design. *The Computer Journal*, June 1999.

[154] Joseph A. Goguen. Lambda calculus in OBJ3. UCSD CSE 230 course web pages., 2001. Available via http://www-cse.ucsd.edu/users/goguen/courses/230/obj/lambda.obj.html.

[155] Joseph A. Goguen. Towards a design theory for virtual worlds: Algebraic semiotics, with information visualization as a case study. In *Proceedings of the Virtual Worlds and Simulation Conference*, January 2001.

[156] Joseph A. Goguen and Rod Burstall. Introducing institutions. In Edmund Clarke and Dexter Kozen, editors, *Proceedings, Logics of Programming Workshop*, volume 164 of *Lecture Notes in Computer Science*, pages 221–256. Springer–Verlag, 1984.

[157] Joseph A. Goguen and Rod Burstall. Institutions: Abstract model theory for computer science. Technical Report CSLI-85-30, Center for the Study of Language and Information, Stanford University, 1985.

[158] Joseph A. Goguen and Rod Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association of Computing Machinery*, 39(1):95–146, January 1992.

[159] Joseph A. Goguen and Răzvan Diaconescu. An Oxford survey of order sorted algebra. *Mathematical Structures in Computer Science*, 4:363–392, 1994.

[160] Joseph A. Goguen, Kai Lin, Akira Mori, Grigore Roşu, and Akiyoshi Sato. Distributed cooperative formal methods tools. In *Proceedings, Automated Software Engineering*, pages 55–62. IEEE Press, November 1997.

[161] Joseph A. Goguen and Grant Malcolm. *Algebraic Semantics of Imperative Programs*. Foundations of Computing Series. The MIT Press, 1996.

[162] Joseph A. Goguen and José Meseguer. Order-sorted algebra I : Equational deduction for multiple inheritance, overloading, exceptions and partial operations. Technical Report PRG-80, Programming Research Group, Oxford University, December 1989.

[163] Joseph A. Goguen, Akira Mori, and Kai Lin. Algebraic semiotics, proofwebs, and distributed cooperative proving. In *Proceedings of the 1997 Conference on User Interfaces for Theorem Provers*, pages 24–34, September 1997.

[164] Joseph A. Goguen and Adolfo Socorro. Module composition and system design for the object paradigm. Technical Report PRG-117, Programming Research Group, Oxford University, January 1995.

[165] Joseph A. Goguen and Joseph Tardo. An introduction to OBJ: A language for writing and testing software specifications. In Marvin Zelkowitz, editor, *Specification of Reliable Software*, pages 170–189. IEEE Press, 1979. Reprinted in *Software Specification Techniques*, Nehan Gehani and Andrew McGettrick, editors, Addison Wesley, 1985, pages 391–420.

[166] Joseph A. Goguen and Will Tracz. An implementation-oriented semantics for module composition. In Leavens and Sitaraman [234], chapter 11, pages 231–263.

[167] Joseph A. Goguen and David Wolfram. *Object Oriented Databases: Analysis, Design and Construction*, chapter On Types and FOOPS. North-Holland Publishing Company, 1991. Also in Proceedings, IFIP TC2 Conference, Windermere, UK, 2–6 July 1990.

[168] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley Publishing Company, 1983.

[169] Adele Goldberg and K. S. Rubin. *Smalltalk-80: The Language*. Addison-Wesley Publishing Company, revised edition, 1995.

[170] Martin Goldstern and Haim Judah. *The Incompleteness Phenomenon: A New Course in Mathematical Logic*. A. K. Peters, 1995.

[171] H. Gomaa, L. Kerschberg, C. Bosch, V. Sugumaran, and I. Tavakoli. A prototype software engineering environment for domain modeling and reuse. In WISR4 [381].

[172] H. Gomaa, L. Kerschberg, V. Sugumaran, C. Bosch, and I. Tavakoli. A prototype domain modeling environment for reusable software architectures. In *Proceedings of the Third International Conference on Software Reuse*, pages 74–83, 1994.

[173] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley Publishing Company, first edition, August 1996.

[174] A. Phillips Griffiths, editor. *Knowledge and Belief*. Oxford University Press, 1967.

[175] T. Gruber. Towards principles for the design of ontologies used for knowledge sharing. In N. Guarino and R. Poli, editors, *Formal Ontology in Conceptual Analysis and Knowledge Representation*. Kluwer Academic Publishing, 2000. In prepartation.

[176] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing. The MIT Press, 1992.

[177] Carl A. Gunter and John C. Mitchell, editors. *Theoretical Aspects of Object-Oriented Programming*. Foundations of Computing. The MIT Press, 1993.

[178] Susan Haack. *Deviant Logic*. Cambridge University Press, 1974.

[179] Susan Haack. *Philosophy of Logics*. Cambridge University Press, 1978.

[180] David Harel. *Lecture Notes in Computer Science*, volume 68 of *Lecture Notes in Computer Science*, chapter First Order Dynamic Logic. Springer–Verlag, 1979.

[181] W. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). In OOPSLA-93 [4], pages 411–428.

[182] Jonathan D. Hay and Joanne M. Atlee. Composing features and resolving interactions. In *The Eighth SIGSOFT Symposium on Foundations of Software Engineering*, pages 110–119. ACM SIGSOFT, SIGPLAN, 2000.

[183] M. Healy and K. Williamson. Applying category theory to derive engineering software from encoded knowledge. In T. Rus, editor, *Proceedings of the Eighth International Conference on Algebraic Methodology and Software Technology*, volume 1816 of *Lecture Notes in Computer Science*, pages 484–498, 2000.

[184] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. In *European Conference on Object-Oriented Programming/ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 25/10 of *ACM SIGPLAN Notices*, pages 169–180. ACM SIGPLAN: Programming Languages, ACM Press and Addison-Wesley Publishing Company, October 1990.

[185] Scott Henninger. An evolutionary approach to constructing effective software reuse repositories. *IEEE/ACM Transactions on Networking*, 6(2):111–140, 1997.

[186] J. Hickey and A. Nogin. Fast tactic-based theorem proving. In *Proceedings of the Thirteenth International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2000)*, volume 1869 of *Lecture Notes in Computer Science*, pages 252–267. Springer–Verlag, 2000.

[187] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

[188] Douglas R. Hofstadter. *Gödel, Escher, Bach: an Eternal Golden Braid*. Basic Books, 1979.

[189] Gao Hongge, M. M. Kokar, and J. Weyman. An approach to automation of fusion using Specware. In *Proceedings of the Second International Conference on Information Fusion*, volume 1, pages 109–116, 1999.

[190] T. R. Huber. Reducing business and legal risks in software reuse libraries. In *Proceedings of the Third International Conference on Software Reuse*, pages 110–117, 1994.

[191] Walter Hürsch, Karl Lieberherr, and Sougata Mukherjea. Object-oriented schema extension and abstraction. In *Proceedings of the 1993 ACM/SIGAPP Symposium on Applied Computing: States of the Art and Practice*, pages 54–62, 1993.

[192] IBM et al. *Object Constraint Language Specification, version 1.1*. The UML 1.1 Consortium, September 1997.

[193] IEEE P1600.1 – standard upper ontology (SUO) working group. See http://suo.ieee.org/ for more information.

[194] Bart Jacobs. *Categorical Logic and Type Theory*, volume 141 of *Studies in Logic and the Foundations of Mathematics*. Elsevier Science, Inc., 1999.

[195] Bart Jacobs and Erik Poll. A logic for the Java modeling language JML. Technical Report CSI-R0018, Computing Science Institute, University of Nijmegen, November 2000.

[196] Ivar Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley Publishing Company, 1994.

[197] S. Jaśkowski. Propositional calculus for contradictory deductive systems. *Studia Logica*, pages 143–160, 1969.

[198] The Jass homepage, 2001. Available via http://semantik.informatik.uni-oldenburg.de/~jass/.

[199] JavaDoc tool and api, 2002. Available via http://java.sun.com/j2se/javadoc/.

[200] Jean-Marc Jézéquel and Bertrand Meyer. Design by contract: The lessons of Ariande. *IEEE Computer*, pages 129–130, January 1997.

[201] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, Inc., 1986.

[202] Robert E. Kent. A KIF formalization for the IFF category theory ontology. In *Proceedings, Workshop on the IEEE Standard Upper Ontology*, August 2001. Available via http://reliant.teknowledge.com/IJCAI01/.

[203] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. Technical Report SPL97-008 P9710042, XEROX Corporation, February 1997.

[204] Joseph R. Kiniry. A reflective web architecture. In preparation.

[205] Joseph R. Kiniry. *The Infospheres Java Coding Standard*. The Infospheres Group, Department of Computer Science, California Institute of Technology, 1997.

[206] Joseph R. Kiniry. IDebug: An advanced debugging framework for Java. Technical Report CS-TR-98-16, Department of Computer Science, California Institute of Technology, November 1998.

[207] Joseph R. Kiniry. The Jiki: A distributed component-based Java Wiki. Available via http://www.jiki.org/, 1998.

[208] Joseph R. Kiniry. The Extended BON tool suite. http://ebon.sourceforge.net/, 2001.

[209] Joseph R. Kiniry. The KindSoftware coding standard. Technical report, KindSoftware, LLC, 2001. Available via http://www.kindsoftware.com/.

[210] Joseph R. Kiniry. Semantic properties for lightweight specification in knowledgeable development environments. Submitted for publication, 2002.

[211] Joseph R. Kiniry and Elaine Cheong. JPP: A Java pre-processor. Technical Report CS-TR-98-15, Department of Computer Science, California Institute of Technology, November 1998.

[212] Joseph R. Kiniry and Daniel M. Zimmerman. The Infospheres Java coding standard. Technical report, Department of Computer Science, California Institute of Technology, 1997. Available via http://www.infospheres.caltech.edu/.

[213] Stephen Cole Kleene. *Introduction to Metamathematics*. D. Van Nostrand Company, Inc., 1952.

[214] Donald E. Knuth. *Literate Programming*. Number 27 in CSLI Lecture Notes. Center for the Study of Language and Information, 1992.

[215] Donald E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley Publishing Company, third edition, 1997.

[216] Donald E. Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley Publishing Company, third edition, 1997.

[217] Donald E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley Publishing Company, second edition, 1997.

[218] Donald E. Knuth. *Digital Typography*. Cambridge University Press, 1999.

[219] Donald E. Knuth and Silvio Levy. *The CWEB System of Structured Documentation*. Addison-Wesley Publishing Company, third edition, 2001.

[220] B. Konikowska, A. Tarlecki, and A. Blikle. A three-valued logic for software specification and validation. *Lecture Notes in Computer Science*, 328:218–242, 1988. Originally published in Fundamenta Informaticae, vol 14, no. 4, pps. 411–453, 1991.

[221] Reto Kramer. iContract– the Java design by contract tool. In *Proceedings of the Twenty-Fourth Conference on the Technology of Object-Oriented Languages (TOOLS 24)*, volume 26 of *TOOLS Conference Series*. IEEE Computer Society, 1998.

[222] Gerhard Lakemeyer and Bernhard Nebel, editors. *Foundations of Knowledge Representation And Reasoning*, volume 810 of *Lecture Notes in Computer Science*. Springer–Verlag, 1994.

[223] Wilf R. LaLond. Designing families of data types using exemplars. *ACM Transactions on Programming Languages and Systems*, 11(2):212–248, April 1989.

[224] Joachim Lambek and Phil Scott. *Introduction to Higher Order Categorical Logic*. Cambridge University Press, 1986. Cambridge Studies in Advanced Mathematics, Volume 7.

[225] Christoper Landauer and Kirstie L. Bellman, editors. *Virtual Worlds and Simulation Conference*, January 1999.

[226] Kevin Lano, Juan Bicarregui, Tom Maibaum, and Jose Fiadeiro. Composition of reactive system components. In Leavens and Sitaraman [234], chapter 12, pages 267–283.

[227] Don Larkin and Greg Wilson. Object-oriented programming and the Objective-C language. Technical report, NeXT Software, Inc. (now Apple Corp.), 1993.

[228] Ora Lassila. Web metadata: A matter of semantics. *IEEE Internet Computing*, 2(4):30–37, 1998.

[229] Ora Lassila and Ralph Swick. Resource description framework (RDF) model and syntax. Technical report, World Wide Web Consortium, 1998.

[230] L. Latour, T. Wheeler, and B. Frakes. Descriptive and prescriptive aspects of the 3Cs model: Setal working group summary, proceedings of the third annual workshop, methods and tools for reuse. Technical Report 9014, CASE Centre, Syracuse University, Ithaca, NY, June 1990.

[231] Larry Latour and Curtis Meadow. Scaling up the 3Cs model: A schema for extensible generic architectures. In WISR4 [381].

[232] G. T. Leavens. Inheritance of interface specifications (extended abstract). *ACM SIGPLAN Notices*, 29(8):129–138, 1994.

[233] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. *Behavioral Specifications of Business and Systems*, chapter JML: A Notation for Detailed Design, pages 175–188. Kluwer Academic Publishing, 1999.

[234] Gary T. Leavens and Murali Sitaraman, editors. *Foundations of Component-Based Systems*. Cambridge University Press, 2000.

[235] F. Lehmann and R. Wille. A triadic approach to formal concept analysis. *Conceptual Structures: Applications, Implementation And Theory*, 954:32–43, 1995.

[236] Gottfried Wilhelm Leibniz. *The Monadology of Leibniz*. University of Southern California, 1930. Introduction, commentary, and supplementary essays by Herbert Wildon Carr.

[237] Gottfried Wilhelm Leibniz. *General Investigations Concerning the Analysis of Concepts and Truths*. University of Georgia Press, 1968. A translation and an evaluation, by Walter H. O'Briant.

[238] H. Leiss. On type inference for object-oriented programming languages. In *Proceedings of the First Workshop on Computer Science Logic*, pages 151–172. Springer–Verlag, 1988.

[239] X. Leroy. The Objective Caml system (release 2.00). www, August 1998.

[240] Y. Lesprance, H. J. Levesque, L. Fangzhen, D. Marcu, R. Reiter, and R. B. Scherl. Foundations of a logical approach to agent programming. In M. Wooldridge, J. P. Muller, and M. Tambe, editors, *Intelligent Agents II. Agent Theories, Architectures, And Languages. IJCAI'95 Workshop (Atal) Proceedings*, pages 331–346. Springer–Verlag, 1996.

[241] Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In OOPSLA-86 [2], pages 214–223.

[242] C. W. Lillie. Distributed network of reuse libraries offers the best approach to successful software reuse. In *Proceedings of the Third International Conference on Software Reuse*, pages 207–208, 1994.

[243] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, August 1977.

[244] Barbara Liskov. Data abstraction and hierarchy. In OOPSLA-87 [3], pages 17–34.

[245] Barbara Liskov and Jeannette M. Wing. Specifications and their use in defining subtypes. In OOPSLA-93 [4], pages 16–28.

[246] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[247] John Locke. *Essay Concerning Human Understanding*. J. M. Dent & Sons Ltd., 1961. Edited by John W. Yolton.

[248] S. Lorcy, N. Plouzeau, and J.-M. Jézéquel. Reifying quality of service contracts for distributed software. In *Technology of Object-Oriented Languages and Systems*, pages 125–139. IEEE Computer Society, August 1998.

[249] J. Lukasiewicz. *Polish Logic*, chapter Philosophical Remarks on Many-valued Systems of Propositional Logic, pages 40–65. Oxford University Press, 1967.

[250] Markus Lumpe, Franz Achermann, and Oscar Nierstrasz. A formal language for composition. In Leavens and Sitaraman [234], chapter 4, pages 69–90.

[251] Luqi and D. E. Cooke. How to combine nonmonotonic logic and rapid prototyping to help maintain software. *International Journal of Software Engineering and Knowledge Engineering*, 5(1):89–118, March 1995.

[252] Luqi and J. McDowell. Software reuse in specification-based prototyping. In WISR4 [381].

[253] Y. S. Maarek, D. M. Berry, and G. E. Kaiser. An information-retrieval approach for automatically constructing software libraries. *IEEE Transactions on Software Engineering*, 17(8):800–813, 1991.

[254] Grant Malcolm and Joseph A. Goguen. Signs and representations: Semiotics for user interface design. In Ray Paton and Irene Nielson, editors, *Visual Representations and Interpretations*, Springer Workshops in Computing, pages 163–172. Springer–Verlag, 1998. Proceedings of a workshop held in Liverpool.

[255] J. Malenfant. On the semantic diversity of delegation-based programming languages. In OOPSLA-95 [5], pages 215–230.

[256] Ernest G. Manes and Michael A. Arbib. *Algebraic Approaches to Program Semantics*. Texts and Monographs in Computer Science. Springer–Verlag, 1986.

[257] Maria Manzano. *Model Theory*. Oxford University Press, 1999.

[258] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

[259] K. Maruyama and K. Shima. A new class generation mechanism by method integration. In *Proceedings of the Fifth International Conference on Software Reuse*. IEEE Computer Society Technical Committee on Software Reuse and the ACM, 1998.

[260] Berna L. Massingill. Experiments with program parallelization using archetypes and stepwise refinement. In *Proceedings of the Third International Workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA'98/IPPS'98)*, 1998.

[261] Michihiro Matsumoto and Kokichi Futatsigi. Object composition and refinement by using non-observable projection operators. In Kokichi Futatsugi, Joseph A. Goguen, and José Meseguer, editors, *Proceedings of the OBJ/CafeOBJ/Maude Workshop at Formal Methods '99*, pages 133–157. Theta, September 1999.

[262] R. Mayr and T. Nipkow. Higher-order rewrite systems and their confluence. *Theoretical Computer Science*, 192(1):3–29, 1998.

[263] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., second edition, 1988.

[264] Bertrand Meyer. *Eiffel the Language*. Prentice-Hall, Inc., third edition, 1990.

[265] Bertrand Meyer. *Introduction to the Theory of Programming Languages*. Prentice-Hall, Inc., 1991.

[266] Bertrand Meyer. Applying design by contract. *IEEE Computer*, October 1992.

[267] Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, Inc., 1992.

[268] Bertrand Meyer. *Reusable Software: The Base Object-Oriented Component Libraries*. The Object-Oriented Series. Prentice-Hall, Inc., 1994.

[269] Bertrand Meyer. *Eiffel the Language*. Prentice-Hall, Inc., third edition, 2002.

[270] R. Meyer. Relevant arithmetic. *Bulletin of the Section of Logic of the Polish Academy of Sciences*, 5:133–137, 1976.

[271] R. Meyer and C. Mortensen. Inconsistent models for relevant arithmetics. *Journal of Symbolic Logic*, 49:917–929, 1984.

[272] H. Mili, E. Ah-Ki, R. Godin, and H. Mcheick. Another nail to the coffin of faceted controlled-vocabulary component classification and retrieval. In *Software Engineering Notes*, pages 89–98. ACM Press, May 1997.

[273] H. Mili, F. Mili, and A. Mili. Reusing software – issues and research directions. *IEEE Transactions on Software Engineering*, 21(6):528–562, 1995.

[274] Yoo Min-Jung, J. Briot, and J. Ferber. Using components for modeling intelligent and collaborative mobile agents. In *Proceedings Seventh IEEE International Workshop on Enabling Technologies: Infrastucture for Collaborative Enterprises (WET ICE '98)*, pages 276–281. IEEE Computer Society, IEEE Computer Society, 1998.

[275] Jack Minker, editor. *Logic-Based Artificial Intelligence*, volume 597 of *The Kluwer International Series in Engineering and Computer Science*. Kluwer Academic Publishing, 2000.

[276] J. M. Molina-Bravo and E. Pimentel. Composing programs in a rewriting logic for declarative programming. Technical Report LO/0203006v1, Dpto. Lenguajes y Ciencias de la Computación, University of Málaga, March 2002.

[277] David E. Monarchi and Gretchen I. Puhr. A research topology for object-oriented analysis and design. *Communications of the ACM*, 35(9):35–47, September 1992.

[278] J. W. Moore. Debate on software reuse libraries. In *Proceedings of the Third International Conference on Software Reuse*, pages 203–204, 1994.

[279] R. C. Moore. *Formal Theories of the Common Sense World*, chapter A Formal Theory of Knowledge and Action. Ablex Publishing Company, 1984.

[280] J.-M. Morel and J. Faget. The REBOOT environment (software reuse). In *Proceedings of the Second International Workshop on Software Reusability*, pages 80–88, 1993.

[281] M. Moriconi and X. Qian. Correctness and composition of software architectures. In *ACM SIGSOFT: Software Engineering*, pages 164–174. Association of Computing Machinery, December 1994.

[282] Peter Mosses. Unified algebras and institutions. In *Proceedings, Fourth Annual Conference on Logic in Computer Science*, pages 304–312. IEEE Press, 1989.

[283] Patrick A. Muckelbauer. *Structural Subtyping in a Distributed Object System*. PhD thesis, Purdue University, 1996.

[284] Philippe Mulet, Jacques Malenfant, and Pierre Cointe. Towards a methodology for explicit composition of metaobjects. In OOPSLA-95 [5], pages 316–330.

[285] Ketan Mulmuley. *Full Abstraction and Semantic Equivalence*. The MIT Press, 1987. 1986 ACM Doctoral Dissertation Award Winner.

[286] Milton K. Munitz, editor. *Logic and Ontology*. New York University Press, 1973. Contributions to a seminar on ontology held under the auspices of the New York University Institute of Philosophy for the year 1970-1971.

[287] Kumiyo Nakakoji. Software reuse in integrated, domain-oriented knowledge-based design environments. In WISR4 [381].

[288] M. Navarro, F. Orejas, and A. Sanchez. On the correctness of modular systems. *Theoretical Computer Science*, 140(1):139–177, 1995.

[289] J. Neighbors. The Draco approach to constructing software from reusable modules. *IEEE Transactions on Software Engineering*, 10(5):564–573, 1984.

[290] Jean-Marc Nerson. Applying object-oriented analysis and design. *Communications of the ACM*, 35(9):63–74, September 1992.

[291] Moritz Neumüller. Applying computer semiotics to hypertext theory and the world wide web. In *Proceedings of the Sixth International Workshop on Open Hypermedia Systems and the Second International Workshop on Structural Computing*, volume 1903 of *Lecture Notes in Computer Science*, pages 57–65. Springer–Verlag, 2000.

[292] C. Neuss and R. E. Kent. Conceptual analysis of resource meta-information. *Computer-Networks And ISDN Systems*, 27(6):973–984, 1995.

[293] O. Nierstrasz and T. D. Meijler. Requirements for a composition language. In *Object-Based Models and Languages for Concurrent Systems. ECOOP'94 Workshop on Models and Languages for Coordination of Parallelism and Distribution*, pages 147–161. Springer–Verlag, 1995.

[294] O. Nierstrasz and T. D. Meijler. Research directions in software composition. *ACM Computing Surveys*, 27(2):262–267, June 1995.

[295] Oscar Nierstrasz, S. Gibbs, and Dennis Tsichritzis. Component-oriented software development. *Communications of the ACM*, 35(9):160–165, September 1992.

[296] Oscar Nierstrasz and Dennis Tsichritzis, editors. *Object-Oriented Software Composition*. Prentice-Hall, Inc., 1995.

[297] J. Q. Ning, K. Miriyala, and W. Kozaczynski. An architecture-driven, business-specific, and component-based approach to software engineering. In *Proceedings of the Third International Conference on Software Reuse*, pages 84–93, 1994.

[298] E. Ohlebusch. Church-rosser theorems for abstract reduction modulo an equivalence relation. *Rewriting Techniques and Applications*, 1379:17–31, 1998.

[299] J. D. Olson and R. E. Kent. Conceptual knowledge markup language, an XML application. Unpublished presentation, given at the XML Developers Day, August 21, 1997, Montreal Canada, August 1997.

[300] Ontology markup language (OML) DTD. Available via http://www.ontologos.org/.

[301] F. Orejas, M. Navarro, and A. Sanchez. Implementation and behavioral equivalence — a survey. *Lecture Notes in Computer Science*, 655:93–125, 1993.

[302] H. Ossher, M. Kaplan, W. Harrison, A. Katz, and V. Kruskal. Subject-oriented composition rules. In OOPSLA-95 [5], pages 235–250.

[303] H. Partsch and R. Steinbruggen. Program transformation systems. *ACM Computing Surveys*, 15(3):199–236, September 1983.

[304] Sukesh Patel, William Chu, Brian Sayrs, and Steve Sherman. A top-down software reuse support environment. In WISR4 [381].

[305] Charles Sanders Peirce. *Peirce on Signs: Writings on Semiotic by Charles Sanders Peirce*. University of North Carolina Press, 1984. Edited by James Hoopes.

[306] Dewayne E. Perry. Software evolution and 'light' semantics. In *Proceedings of ICSE '99*, 1999.

[307] Frank Pfenning. Logical frameworks. In *Handbook of Automated Reasoning*, pages 1063–1147. The MIT Press, 2001.

[308] L. J. Pinson and R. S. Wiener. *Objective-C: Object-Oriented Programming Techniques*. Addison-Wesley Publishing Company, 1991.

[309] Plato. *The Republic, with an English translation by Paul Shorey*. Harvard University Press, 1963. In two volumes. Discussions on classification found in Book V.

[310] B. Poizat. *A Course in Model Theory: An Introduction to Contemporary Mathematical Logic*. Springer–Verlag, 2000.

[311] G. Priest, R. Routley, and J. Norman, editors. *Paraconsistent Logic: Essays on Inconsistent*. Philosophia, 1989.

[312] Arun D. Raghavan. Design of a JML documentation generator. Technical Report 00-12, Department of Computer Science, Iowa State University, March 2000.

[313] Ravi Ramamoorthi, Adam Rifkin, Boris Dimitrov, and K. Mani Chandy. A general resource reservation framework for scientific computing. In *Proceedings of the First International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE) Conference*, December 1997.

[314] C. V. Ramamoorthy and D. E. Cooke. The correspondence between methods of artificial intelligence and the production and maintenance of evolutionary software. In *Third International Conference on Tools for Artificial Intelligence TAI '91*, pages 114–18, 1991.

[315] C. V. Ramamoorthy, D. E. Cooke, and C. Baral. Maintaining the truth of specifications in evolutionary software. *International Journal on Artificial Intelligence Tools (Architectures, Languages, Algorithms)*, 2(1):15–31, March 1993.

[316] P. V. Rangan. An axiomatic basis of trust in distributed systems. In *Proceedings of the 1988 IEEE Symposium on Security And Privacy*, pages 204–211. IEEE Computer Society, 1988.

[317] M. B. Ratcliffe and R. J. Gautier. System development through the reuse of existing components. *Software Engineering Journal*, 6(6):406–412, 1991.

[318] Nicholas Rescher and Alasdair Urquhart. *Temporal Logic [by] Nicholas Rescher and Alasdair Urquhart*, volume 3 of *Library of Exact Philosophy*. Springer–Verlag, 1971.

[319] B. J. Rhodes and P. Maes. Just-in-time information retrieval agents. *IBM Systems Journal*, 39(3–4):685–704, 2000.

[320] C. Rich and R. C. Waters. Automatic programming: Myths and prospects. *IEEE Computer*, 21(8):40–51, August 1988.

[321] Charles Rich and Richard C. Walters. *Software Reusability*, chapter Formalizing Reusable Software Components in the Programmer's Apprentice. Volume II: Applications and Experience of Biggerstaff and Perlis [43], 1989.

[322] Grigore Roşu. Abstract semantics for module composition. Technical Report CS2000-653, University of California, San Diego, May 2000.

[323] R. Ruggia and A. P. Ambrosio. A toolkit for reuse in conceptual modelling. *Advanced Information Systems Engineering*, 1250:173–186, 1997.

[324] Jim Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, Bill Lorensen, and William Lorenson. *Object-Oriented Modeling and Design*. Prentice-Hall, Inc., 1991.

[325] Jim Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley Publishing Company, 1998.

[326] H. Saiedian. An evaluation of extended entity-relationship model. *Information And Software Technology*, 39(7):449–462, 1997.

[327] Giovanni Sambin and Jan Smith, editors. *Twenty-Five Years of Constructive Type Theory*. Oxford University Press, 1998.

[328] David Sandberg. An alternative to subclassing. In OOPSLA-86 [2], pages 424–428.

[329] Silvija Seres, K. Rustan M. Leino, and James B. Saxe. ESC/Java quick reference. Technical Report Technical Note 2000-004, Compaq SRC, October 2000.

[330] N. Shankar, S. Owre, and J. M. Rushby. A tutorial on specification and verification using PVS. Technical report, Computer Science Laboratory, SRI International, March 1993.

[331] Mary Shaw. Procedure calls are the assembly language of system interconnection: Connectors deserve first-class status. In *Proceedings of the Workshop on Studies of Software Design*, May 1993.

[332] Y. Shinkawa and M. J. Matsumoto. Knowledge-based software composition using rough set theory. *IEICE Transactions on Information and Systems*, E83-D(4):691–700, April 2000.

[333] C. Silverstein, S. Brin, and R. Motwani. Beyond market baskets: Generalizing association rules to dependence rules. *Data Mining & Knowledge Discovery*, 2(1):39–68, 1998.

[334] C. Silverstein, S. Brin, R. Motwani, and J. Ullman. Scalable techniques for mining causal structures. *Data Mining & Knowledge Discovery*, 4(2–3):163–192, 2000.

[335] Charles Simonyi. The death of computer languages: The birth of intentional programming. Technical Report MSR-TR-95-52, Microsoft Research, September 1995.

[336] Charles Simonyi. Intentional programming – innovation in the legacy age. Presented at IFIP WG 2.1 meeting, June 1996.

[337] M. Simos and J. Anthony. Weaving the model web: A multi-modeling approach to concepts and features in domain engineering. In *Proceedings of the Fifth International Conference on Software Reuse*, 1998.

[338] M. D. Simpson. CVS version control and branch management: What to do when version control gets complicated. *Dr. Dobbs Journal*, 25(10):108–114, October 2000.

[339] G. Sindre, R. Conradi, and E. Karlsson. The REBOOT approach to software reuse. *Journal of System Software*, 30(2):201, 1995.

[340] Slashcode/Slashdot's Karma system. http://slashcode.com/.

[341] Monique Snoeck and Guido Dedene. Existence dependency: The key to semantic integrity between structural and behavioral aspects of object types. *IEEE Transactions on Software Engineering*, 24(4), April 1998.

[342] J. Solderisch. Beyond a software library: Building an organon. In *Proceedings of the Third International Conference on Software Reuse*, pages 205–206, 1994.

[343] James Solderitsch and John Thalhamer. Asset library open architecture framework — sharing reusable assets. In WISR4 [381].

[344] L. S. Sorumgard, G. Sindre, and F. Stokke. Experiences from application of a faceted classification scheme. In *Proceedings of the Second International Workshop on Software Reusability*, pages 116–124, 1993.

[345] Y. V. Srinivas and R. Jüllig. Specware: Formal support for composing software. *Lecture Notes in Computer Science*, 947:399–422, 1993.

[346] Guy Steele. *Common Lisp: The Language*. Digital Press, second edition, 1990.

[347] Mark-Oliver Stehr and José Meseguer. Pure type systems in rewriting logic. In *Proceedings of LFM'99: Workshop on Logical Frameworks and Meta-languages*, September 1999.

[348] Lynn Andrea Stein. Delegation is inheritance. In OOPSLA-87 [3], pages 138–146.

[349] Thomas Streicher. *Semantics of Type Theory: Correctness, Completeness, and Independence Results*. Birkhäuser, 1991.

[350] Standard upper merged ontology, 2002. Available via http://ontology.teknowledge.com/.

[351] SUO OFF foundation ontology, 2002. Available via http://suo.ieee.org/IFF/.

[352] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Publishing Company, 1997.

[353] PLATINUM technology, Veronica Bowman, Edward Swanstrom, et al. *Paradigm Plus: Methods Manual Addendum (OOCL)*. PLATINUM technology, 3.52 edition, 1997.

[354] P. Thistlewaite, M. McRobbie, and R. Meyer. *Research Notes in Theoretical Computer Science*, chapter Automated Theorem-Proving in Non-Classical Logics. Pitman, 1988.

[355] W. Tracz. *Formal Specification of Parameterized Programs in LILEANNA*. PhD thesis, Stanford University, 1992.

[356] Will Tracz. Implementation working group summary. In J. Baldo, editor, *Reuse in Practice Workshop*, page 107, July 1989.

[357] Will Tracz. LILEANNA: A parameterized programming language. In *Proceedings of the Second International Workshop on Software Reusability*, pages 66–78, 1993.

[358] Raymond Turner. *Logics for Artificial Intelligence*. Ellis Horwood Series in Artificial Intelligence. Ellis Horwood Limited, 1984.

[359] D. Ungar and R. B. Smith. Self: The power of simplicity. *Lisp and Symbolic Computation*, 4(3):187–205, 1991.

[360] Unknown, editor. *Model Theory and Applications*, volume 195 of *American Mathematical Society Translations — Series 2*. American Mathematical Society, October 1999.

[361] Max Urchs. *Essays on Non-Classical Logic*, chapter Recent Trends in Paraconsistent Logic, pages 219–246. World Science Publishing, 1999.

[362] P. E. van der Vet and N. J. I. Mars. Bottom-up construction of ontologies. *IEEE Transactions on Knowledge and Data Engineering*, 10(4):513–526, 1998.

[363] Anca Vermesan and Frans Coenen, editors. *Validation and Verification of Knowledge Based Systems: Theory, Tools, and Practice*. Kluwer Academic Publishing, September 1999.

[364] Kim Waldén and Jean-Marc Nerson. *Seamless Object-Oriented Software Architecture - Analysis and Design of Reliable Systems*. Prentice-Hall, Inc., 1994.

[365] Guijun Wang, Liz Ungar, and Dan Klawitter. Component assembly for OO distributed systems. *IEEE Computer*, 32(7):71–78, July 1999.

[366] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley Publishing Company, 1999.

[367] Richard C. Waters. Cliché-based program editors. *ACM Transactions on Programming Languages and Systems*, 16(1):102–150, January 1994.

[368] WebDAV implementations, 2001. See website at http://www.webdav.org/projects/, particularly the Jakarta Slide project at http://jakarta.apache.org/slide/index.html.

[369] Stuart Weibel, Jean Godby, Eric Miller, and Ron Daniel. OCLC/NCSA metadata workshop report. http://www.oclc.org:5046/conferences/metadata/dublin_core_report.html, March 1995.

[370] William Weiss and Cherie D'Mello. *Fundamentals of Model Theory*. Department of Mathematics, University of Toronto, 1997. Available via http://www.math.toronto.edu/weiss/.

[371] P. Wendorff. Linking concepts to identifiers in information systems engineering. In S. Sarkar and S. Narasimhan, editors, *Proceedings of the Ninth Annual Workshop on Information Technologies and Systems*, pages 51–56, 1999.

[372] P. Wendorff. A formal approach to the assessment and improvement of terminological models used in information systems engineering. *Software Engineering Notes*, 26(5):83–87, 2001.

[373] E. J. Whitehead and Y. Y. Goland. WebDAV: A network protocol for remote collaborative authoring on the web. In *Proceedings of the Sixth European Conference on Computer Supported Cooperative Work*, pages 291–310. Kluwer Academic Publishing, 1999.

[374] B. Whittle. Models and languages for component description and reuse. *Software Engineering Notes*, 20(2):76–89, April 1995.

[375] B. Whittle and M. Ratcliffe. Software component interface description for reuse. *Software Engineering Journal*, 8(6):307–318, 1993.

[376] N. Wilde and R. Huitt. Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering*, 18(12):1038–1044, 1992.

[377] R. Wille. Concept lattices and conceptual knowledge systems. *Computers and Mathematics with Applications*, 23(6-9):493–515, 1992.

[378] R. Wille. The basic theorem of triadic concept analysis. *Order—A Journal on the Theory of Ordered Sets And Its Applications*, 12(2):149–158, 1995.

[379] R. Wille. Conceptual graphs and formal concept analysis. *Conceptual Structures: Fulfilling Peirce's Dream*, 1257:290–303, 1997.

[380] K. Williamson and M. Healy. Industrial applications of software synthesis via category theory. In *Proceedings of the Fourteenth IEEE International Conference on Automated Software Engineering*, pages 35–43. IEEE Computer Society, 1999.

[381] *The Fourth Workshop on Software Reuse*, November 1991.

[382] Ludwig Wittgenstein. Wittgenstein und der wiener kreis. *Schriften*, 3, 1967.

[383] Ludwig Wittgenstein. *Wittgenstein's Lectures on the Foundations of Mathematics*. Ithaca, 1976.

[384] Fuqing Yang, Hong Mei, Qiong Wu, and Bing Zhu. An approach to software development based on heterogeneous component reuse and its supporting system. *Technological Sciences*, 40(4), August 1997.

[385] D. M. Yellin and R. E. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2), 1997.

[386] Daniel M. Yellin and Robert E. Strom. Interfaces, protocols, and the semi-automatic construction of software adaptors. In *Proceedings of OOPSLA '94*, October 1994.

[387] John W. Yolton, editor. *Theory of Knowledge*. The Macmillan Company, 1965.

[388] D. H. H. Yoon. The categorical framework of open systems. In P. A. Ng, L. C. Seifert, C. V. Ramamoorthy, and R. T. Yeh, editors, *Proceedings of the Second International Conference on Systems Integration*, pages 388–391. IEEE Computer Society, 1992.