The Book Review Column[1]

by William Gasarch
Department of Computer Science
University of Maryland at College Park
College Park, MD, 20742
email: gasarch@cs.umd.edu

Welcome to the Book Reviews Column. We hope to bring you at least two reviews of books every month.

We are looking for reviewers for the following books: (1) *Information flow: The logic of distributed systems* by Barwise and Seligman, (2) *Control flow semantics* by Bakker and Vink. I can be contacted at the email address above.

Review of: **Metamathematics, Machines, and Gödel's Proof**[2]
by Author: N. Shankar
Publisher: Cambridge University Press

Reviewed by K. Purang, University of Maryland at College Park

1 Overview

This book, first published in 1994, is derived from the author's 1986 Ph.D dissertation, one of the main goals of which was to gauge the usefulness of automatic theorem proving technology in constructing and verifying proofs. The proofs chosen for this task were Gödel's first incompleteness theorem and the Church-Rosser theorem of the lambda calculus. The theorem prover used was the Boyer-Moore theorem prover which can be obtained by ftp [1]. A manual for the prover has been published as [2]. The author gives a detailed account of the more important steps leading to the mechanized verification of proofs of these theorems. The theorem prover did not discover the proofs but checked definitions and lemmas (about 2000 for the incompleteness theorem and supplied by the author) that lead to the theorems.

2 Interesting aspects of the proof

While the proofs used by the author have been previously published, there are some interesting aspects to them that we highlight here:

- The Boyer-Moore theorem prover does not support quantifiers. Therefore, the statements of the theorems and the proofs are formulated in a slightly unfamiliar way.
- The author shows the incompleteness of Z2, Cohen's theory of hereditarily finite sets [3]. Z2 has the same expressive power as Peano Arithmetic and was chosen for convenience: Z2 has an ordered pairing operation whereas pairing would have had to be encoded in Peano Arithmetic.

- The metalanguage used is Boyer-Moore logic which is closely related to pure Lisp. Lisp is also used to characterize the class of partial recursive functions.
- The proofs for the Church-Rosser theorem are done using the de Bruijn notation [4]. This is equivalent to the standard notation (and is proved to be so by the Boyer-Moore theorem prover). Using this notation simplifies the problems of substitution and α-equivalence.

3 Chapter summary

Chapter 1: Introduction This is an introductory chapter giving a wide-ranging and necessarily compressed history of logic and theorem proving. This is followed by an overview of the steps leading to the proofs of the incompleteness theorem and of the Church-Rosser theorem. Finally, the Boyer-Moore theorem prover is described. We provide a summary of the description of the Boyer-Moore theorem prover below.

The Boyer-Moore theorem prover is a heuristic theorem prover that can be used to prove properties of pure Lisp programs in a quantifier free logic. The theorem prover exploits the duality between proofs by induction and definitions by recursion. Some of its heuristics enable it to automatically choose the induction schemas needed for proofs by induction. It is to be noted that the Boyer-Moore theorem prover does not generate large proofs but is led to them by the definitions and lemmas supplied by the user. Thus, it is more properly regarded as a proof checker.

The logic used in the Boyer-Moore theorem prover, the Boyer-Moore logic is closely akin to pure Lisp. It deals only with total functions and is based on a first-order logic with equality but without quantifiers. Some features of the logic are:

- The logic contains the logical constants $TRUE$ and $FALSE$ and the only logical operation is the 3-place function IF. A set of basic axioms is defined for the other logical connectives and equality.
- The *shell principle* of the Boyer-Moore theorem prover enables the user to add axioms describing new objects. Natural numbers, for example, are axiomatized by defining a recognizer function $NUMBERP$, a constructor function $ADD1$, a destructor function $SUB1$ and a bottom object $ZERO$, all of which are added in a shell.
- The Boyer-Moore theorem prover accepts new function as axioms only if they are total. These functions must be defined by recursion and the recursion must "bottom out" at some point. The *induction principle* then allows inductive proofs based on these definitions by recursion.

The interface to the theorem prover allows the user to add new shells, define functions and prove theorems. The prover produces an annotated transcript of its proof attempts which is especially useful when the proofs fail.

Chapter 2: The statement of the incompleteness theorem The proof of the incompleteness theorem used is based on Rosser's proof [5] and is asserted in Z2 rather than in PA as noted earlier. The logic of Z2 follows that of Shoenfield [6]. Since the incompleteness theorem is asserted in Z2, the author formulates the metatheoretic definition of Z2 in Lisp (the Boyer-Moore logic). Next, a proof checker for Z2 is defined. Then, the incompleteness theorem can be stated. The usual formulation is that the consistency of Z2 implies the existence of a sentence that is neither provable nor disprovable. The consistency hypothesis however, requires quantification which cannot be done in the Boyer-Moore theorem prover. Therefore, the incompleteness theorem instead asserts the

existence of a sentence such that if it is either provable or disprovable, it is both provable and disprovable. That sentence is then explicitly constructed in the proof.

Chapter 3: Derived inference rules Derived inference rules (inference rules for which there is in the theory a deduction of the conclusions from the premises) are important in that they allow bigger steps to be taken in proofs. The major part of this chapter shows the use of the Boyer-Moore theorem prover in proving the tautology theorem (all propositional tautologies are theorems of Z2).

Chapter 4: The representability of the metatheory As in non-mechanized proofs of the incompleteness theorem, the representability of the metatheory in the object language (Z2) is a substantial undertaking. The metatheory of Z2 had earlier been expressed in Lisp. Representability then consists in showing that these Lisp expressions can be evaluated in Z2. This is done by showing that a Lisp interpreter can be represented in Z2. The standard trick is used to represent recursive Lisp functions in Z2 using *traces*. The steps to the representability theorem are therefore the following:

- Define a Lisp evaluator.
- Show that the evaluator can be represented in Z2.
- Show that the metatheoretic Lisp functions for Z2 can be computed in the evaluator.

Chapter 5: The undecidable sentence Finally, the undecidable sentence can be constructed and the incompleteness theorem proved. The following steps are required to construct the undecidable sentence:

- Represent an enumeration of Z2 proofs.
- Define a Lisp predicate to search for the first proof or disproof of a formula (a theorem checker).
- Use the fact that this predicate is representable to construct a sentence that states of itself that it is not provable.

The undecidable sentence that is constructed is parameterized by axioms added to the theory, therefore the incompleteness proof shows that Z2 is essentially incomplete: it is incomplete under addition of finitely many new axioms.

Chapter 6: A Mechanical Proof of the Church-Rosser Theorem The chapter starts with an introduction to the lambda calculus. The Church-Rosser theorem essentially states that reductions have the diamond property (If $X \rightarrow Y$ and $X \rightarrow Z$ then there exists a W such that $Y \rightarrow W$ and $Z \rightarrow W$.) The proof involves the following steps:

- Show that the transitive closure of a relation with the diamond property has the diamond property.
- Define the relation "X walks to Y" such that $X \rightarrow Y$ is its transitive closure.
- Show that walks have the diamond property.

An essential part of this process is the formalization of the lambda calculus in Lisp. As noted above, the standard notation is not the most convenient to use for the mechanized proof however, the de Bruijn notation is used instead. The proof of the Church-Rosser theorem is stated in the standard notation, translated to the de Bruijn notation for the proof and then translated back.

Chapter 7: Conclusions Some of the conclusions the author derives from this work are:

- Most of the time (18 months for the incompleteness theorem) was spent in finding errors in definitions, theorems and proofs. Having an automatic proof checker made this aspect much easier. However, the proof checker cannot guarantee that there are no errors in the proof: the translation of the definitions or statements could have mistakes, for example.
- Proof checking is a creative activity in that the exact formulation of the concepts has a great influence on the working of the proof checker.
- The fact that the Boyer-Moore theorem prover gives commentary as it attempts proofs is useful to either repair the proofs and definitions or to find counterexamples to the theorem one is attempting to prove.

The author also presents a critique of the Boyer-Moore theorem prover. An important point is that the Boyer-Moore theorem prover allows many trivial proofs to be automatically done, however, when its heuristics fail, it becomes difficult to control the theorem prover.

4 Opinion

The proofs presented in this book are not novel, and it is not likely to give the reader new insight into either Gödel's incompleteness theorem or the Church-Rosser theorem. It is not a primer to automated theorem proving either.

This book will be of greatest benefit to people interested in current tools that provide assistance in constructing complex mathematical proofs. In that case the reader would benefit from reading the book in conjunction with using the Boyer-Moore theorem prover to do the proofs described in the book. The scripts for doing that are available with the standard distribution of the Boyer-Moore theorem prover [1]. From this perspective, the book can be seen as a detailed road-map to the scripts for the proofs of Gödel's first incompleteness theorem and the Church-Rosser theorem that are provided in the Boyer-Moore theorem prover distribution.

References

[1] R. S. Boyer and J. S. Moore. *Nqthm-1992*. ftp://ftp.cs.utexas.edu/pub/boyer/nqthm/nqthm-2nd-edition.tar.Z

[2] R. S. Boyer and J. S. Moore. *A Computational Logic Handbook*. Academic Press, 1988.

[3] P. J. Cohen. *Set theory and the continuum hypothesis*. Benjamin, 1966.

[4] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indagationes mathematicae*, 34(5), 1972.

[5] J. B. Rosser. Extensions of some theorems of Gödel and Church. *Journal of symbolic logic*, 1936.

[6] J. R. Shoenfield. *Mathematical logic*. Addison-Wesley, 1967.

Review of **Reasoning About Knowledge** by
Ronald Fagin, Joseph Halpern, Yoram Moses and Moshe Vardi[3]
Published by MIT Press

Review by Alexander Dekhtyar, University of Maryland
dekhtyar@cs.umd.edu

"Knowledge is a deadly friend when no one sets the rules"
Pete Sinfield, King Crimson

1 Overview

How much do you know ? How much do you know about what you know ? How much do you know about what someone else knows ? How much that someone else knows about what you know about what she knows, and how much do you know about that ?

We started with a very simple question about knowledge, and almost immediately the chain of questions lead to complicated and hard to comprehend ones. Classic logic teaches us how one can reason about truth and falsehood of statements, but can it help us if we want to reason about our (or someone else's) knowledge ? And if the answer to the last question is "yes", then how can we achieve this goal ?

These are just a few of the questions, that served as motivation for the book *Reasoning About Knowledge*. In the book authors discuss the extentions of classical logic that encapsulate the desired meaning of the statements such as "I know X", "I know that Alice knows X" and "Everyone knows that I know X".

In the first chapters the book provides a number of motivating examples which lead the reader to an informal understanding of the authors' concept of knowledge. After that the notion of knowledge is being formalized by introduction of modal operators for knowledge, common knowledge and distributed knowledge into the classic Propositional logic. Kripke structures are introduced as models at the same time and the obtained modal logics are then studied closely with respect to their soundeness and completeness properties, as well as the complexity of decision procedures.

Once the general background is layed out, authors start discussing a variety of applications of knowledge-based reasoning to other problems in such fields as AI and Communications. A number of chapters of the book are devoted to discussing multi-agent systems and their behavior. A concept of Common Knowledge and its relation to Agreement and Cooperation between agents is also discussed.

The more detailed summary of the book is provided below.

2 Summary of Contents

The following example, called *The Muddy Children Puzzle* is introduced in **Chapter 1** and used throughout the book. We have n children and k out of them have muddy foreheads. The children cannot see their own foreheads but they can see the forheads of every other child. Their father comes and tells them that some of the children have muddy foreheads and asks each child to identify whether his/her forehead is muddy or not. On each round, each child can either state that he/she knows whether his/her forehead is muddy or not, or say "I don't know". Assuming that the

children are intellegent and can can reason logically, how many rounds would it take for children with muddy foreheads to guess that ?.

Chapter 1 provides the introduction to the field of *Epistemology* (the study of knowldege) and a short overview of the book. Together with that, authors introduce the *Muddy Children Puzzle*, which would serve both as an example of informal reasoning about knowledge that authors want to formalize and as a test for expressibility of the systems constructed in later chapters. As we read the book we will find out that the *Muddy Children Puzzle* possesses almost all the properties dicussed later in the book.

Chapter 2 introduces the Possible Worlds Model to represent reasoning about knowledge. A possible world is just a set of statements that an agent considers possible based on her knowledge. Kripke structures are introduced and authors discuss how the Possible Worlds Model has to be updated in order to incorporate reasoning about Common Knowledge and Distributed Knowledge. Common knowledge for a group of agents is defined as follows: "X is common knowledge among agents in group G" is equivalent to the following statements "Every agent in group G knows X" and "Every agent in group G knows that every other agent in G knows X", etc....). Also we say that "Group G of agents possesses distributed knowledge of X" if the combined knowledge of all agents in G will have X as a consequence.

Authors proceed by constructing the model for the Muddy Children Puzzle. The properties of the Possible Worlds Model for knowledge representation are discussed in the rest of the chapter.

Chapter 3 is devoted to (1) proving the main soundness and completeness results and (2) discussing the computational complexity of the decision procedure for the modal logics constructed. First the axiomatizations are given for different models of knowledge in previous chapter and then their correspondent models are established with appropriate soundness and completeness theorems. It turns out that the modal logics introduced for representing knowldege are well-known modal logics T, $S4$, $S5$ and $KD45$. Each of the logics represents one of the views of what "knowledge" is and what are its properties.

In the rest of the chapter Decidability of the propsed logics is stated and proven, logics that incorporate Common and Distributed knowledge are introduced and the complexity of the decision problem for each logic in the chapter is described (although only NP-completeness results for $S5$ and $KD45$ are actually established). The last section of the chapter discusses the First-order extensions of the logics discussed previously.

Chapter 4 discusses the notion of a multi-agent system. First the notion of a multi-agent system as a set of agents acting on their own behalf and environment "surrounding" the agents and the notions of both local and global states of the system are defined. The authors argue that in most cases the reasoning in multi-agent systems is actually reasoning about knowledge of agents. A number of examples supporting the point is provided and the formal methods for incorporating knowledge and time into multi-agent systems are introduced (the latter being the introduction of temporal logics). The rest of the chapter is devoted to detailed examples of a various multi-agent systems, such as Knowledge Bases, Message Passing Systems and Game Trees. Most of these examples are widely used in later chapters.

Chapter 5 continues the study of multi-agent systems. One of the first examples of a multi-agent system constructed in **Chapter 4** showed how difficult it can be to describe even a small multi-agent system in its entirety. This chapter focuses on the means by which multi-agent systems change their states and on the means of describing these state changes. A notion of a protocol for an agent is introduced as a function that determines a set of possible actions for an agent based on current state. Another important notion, context, representing the "global" restrictions on the behavior of the system is introduced too. Last sections of the chapter discuss formal ways to

represent protocols, such as **programs**. A program representing a protocol for an agent is defined as a **case**-statement.

Chapter 6 discusses the effects of Common Knowledge in multi-agent systems. A notion of Cooperation between agents in multi-agent systems is introduced and studied on a number of examples, such as Coordinated Attack. As a result of this study, a number of important results about Common Knowledge in multi-agent systems are established, such as:

- Cooperation in a multi-agent system is impossible without Common Knowledge
- In the multi-agent systems where communication between agents is assumed unreliable, it is impossible to achieve Common Knowledge.

Another interesting negative result with respect to Common Knowledge is that in a multi-agent system with Common Knowledge agents cannot "agree to disagree", i.e. perform different actions based on the same Common Knowledge assumption.

Finally, a Cooperation problem called Simultaneous Byzantine Agreement (SBA) is presented and the methods for finding the protocols for agents that lead to achievement of SBA in a multi-agent system are described.

Chapter 7 continues the study of Knowledge-based reasoning and focuses on programs for protocols. The Muddy Children Puzzle again provides an example of when the program has to include formulas refering to knowledge. This leads to the definition of Knowledge-Based programs and to a study of their properties. It turns out that unlike previously considered programs, knowledge-based programs allow for multiple representations by multi-agent systems, or in some cases have no representation at all. The authors proceed to establish a sufficient condition for a Knowledge-based program to have unique representation.

Then, the Knowledge Bases are revisited (as they were described in **Chapter 4**). Now a more general situation is considered, where some updates to the KNB may include information about knowledge. After that, a Knowledge-based program for SBA (see **Chapter 6**) is constructed and discussed.

At the end of the chapter a new problem, Sequence Transition is introduced and discussed. A knowledge-based program for Sequence-Transimission problem is constructed and is analyzed.

In **Chapter 8** the authors turn to formalizing the concept of evolving knowledge. Axiomatizations are given for the temporal extention of modal logics described in **Chapter 4**, together with model-theoretic semantics and proofs of soundness and completeness. A variety of axiomatizations is dicussed with respect to both synchronous and asynchronous systems. At the end of the chapter authors provide a summary of the complexity results for decision procedures for different logics considered.

Chapter 9 deals with Logical Omniscience. Logical Omniscience can be described as a requirement for an agent to know all the tautologies, and all the logical consiquences of the true formulas. This property was one of the key assumptions on the knowledge in previous chapters. However, as authors argue, in many cases the achievement of logical omniscience is unrealistic. Is it possible then to somehow formalize knowledge-based reasoning without logical omniscience ?

The authors provide a number of methods of doing so, including syntactic and semantic alterations of validity function, a change of the notion of "truth" and a change of the notion of "possible world". As yet another possible solution, a concept of awareness is introduced to successfully limit the omniscience (according to this concept, an agent can draw logical conclusions only about information she is *aware* of). Finally, authors discuss reasoning in an inconsistent world (i.e., when an

agent knows a fact and its negation, but because of lack of logiac omniscience is unable to realize that).

In **Chapter 10** the authors discuss rturn to knowledge-based programs (see **Chapter 4** and **Chapter 7**) but now the object of interest is the computational aspect. As in many systems agents have to take actions based on their knowledge, this knowledge has to be computable, in order for the appropriate action to take place. But in this case, we obtain yet another approach to knowledge-based reasoning, this time based on the ability of the agent to compute its knowledge (the authors call this *Algorithmic knowledge*). Its properties are studied throughout the chapter.

Finally, **Chapter 11** studies Common Knowledge in detail. As it had been shown previously (see **Chapter 6**) in some situations common knowledge cannot be achieved in finite amount of time. In this chapter authors study the reasons of that, and propose a model of Common Knowledge as a fix-point. Together with that, approximations of common knowledge in multi-agent systems are described. The chapter ends with a number of examples of application of coordinated attack problem.

3 Style

Although some of the notions introduced and studied in the book are rather complicated, the authors made every attempt to explain the intuition behind them on some easy-to-understand examples. This, together with authors' tendency to discuss not only the formalisms and proof details, but also some philosophical aspects of the results make the book quite readable even for people who are not familiar with the area. Perhaps the only necessary prerequisite for successful understanding of the material presented in the book for a reader is previous knowledge of Propositional Calculus. On the other hand, such a style can make it hard to find quickly a certain point in the text.

This is not to say that the book is not formal enough, or does not contain deep enough results. All these are present, but we can either find them between very careful explanations, or as addeniums at the end of the chapters. So, while an unexperienced reader can easily skip the proofs, specialists in the field will be able to see them if needed.

Every chapter is accompanied by an extensive set of problems of various difficulty related to the material presented in it and detailed bibliographical notes. The latter makes it easier to navigate through an extensive bibliography at the end of the book, while the former suggests that the book can be used as a text book for a graduate or even an undergraduate course in the area.

4 Opinion

The book seems to be an excellent introduction into the area of reasoning with knowledge. Its style makes it relatively easy to read, and its contents easy-to-understand.

Not only that, but also the book seems to provide some relavant information about a number of fields, that initially seem to be not realated to reasoning about knowledge. For example, chapters 2 and 3 of a book can be easily recommended as an introduction into modal logic for a novice, which is, albeit quite specific, nevertheless provides good intuition and a good insight into the area. Chapter 8 deserves the same characteristic as an introduction to temporal modal logics. Chapter 4 and later chapters introduce formal description of multi-agent systems and discuss different formalisms of their behavior, which seems very relevant in light of today's developement of collaborative agent technologies and their applications.

Review of **Isomorphisms of Types: from λ-calculus to information retrieval and language design**[4]
by Roberto Di Cosmo
Series: Progress in Theoretical Computer Science
Publisher: Birkhäuser, 1995

Reviewer: Christopher League, Yale University
league-christopher@cs.yale.edu

1 Overview

The λ-calculus is an elegant computational model used extensively in the formal semantics of programming languages. It was first described by Alonzo Church in 1932, as a foundation of mathematics based not on *sets*, but directly on *functions*. λ-calculus was, in part, the inspiration for the programming language Lisp, and its typed versions have become the foundation for a family of modern type-safe languages that includes Haskell and ML.

In its original untyped form, the λ-calculus is an extremely simple rewrite system with the concise syntax $e ::= x | e_1\ e_2 | \lambda x.e$, where x ranges over variable names, $e_1\ e_2$ is called an *application*, and $\lambda x.e$ is called an *abstraction*. An abstraction is analogous to a *function*; x is the *formal parameter* and e is the *body*. Effectively, there is just one rewrite rule, called β-reduction: $(\lambda x.e_1)\ e_2 \rightarrow_\beta e_1[e_2/x]$, where $e_1[e_2/x]$ means that we *substitute* e_2 for occurrences of x in the expression e_1. Voilà — Turing equivalence.

Actually, there are a few technicalities we are glossing over, having to do with renaming bound variables and such. And in modern usage we generally augment the system with natural numbers, arithmetic, and boolean logic as primitives, although these can be completely encoded in the raw calculus.

This book is about *types* (a familiar notion in both mathematical logic and programming languages) and a family of *typed λ-calculi*. Armed with these formalisms, Di Cosmo explores various *isomorphisms* of types, which then prove useful in building a software engineering tool to classify and retrieve software components! Specifically,

- Chapters 1-3 present a quick introduction to the various typed λ-calculi, sketch some isomorphisms, and then prove important results about confluence, decidability, and normalization.
- Chapters 4-5 contain the meat of the theoretical contribution of this work, presenting complete theories for first- and second-order isomorphic types.
- Chapters 6-7 demonstrate the utility of these isomorphisms in a software engineering application, and sketch other potential applications and extensions.

2 Summary of Contents

Chapter 1 starts with some motivations for *types*, taken from mathematical logic and programming languages. In set theory, types are one way to prevent Russel's paradox, by appropriately restricting set-building operations. Similarly, in modern programming languages, types prevent the programmer from accidentally (or intentionally) writing code that would be unsafe or nonsense.

Next, the reader is taken on a whirlwind tour of typed λ-calculi. In this world, every formal parameter and every expression has a type, whether explicit or implicit. For example, $\lambda x : A.e$ is an abstraction for which the parameter must be of type A. This is a more complicated system, with a series of typing rules such as the following:

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash g : A}{\Gamma \vdash f\ g : B} \quad \text{(APPL)}$$

This says that **if** f is a function from some domain A to some range B, and g is an expression of type A, **then** the application $f\ g$ has type B. Here, Γ denotes an *environment*, a series of type declarations for variables.

Most *type signatures* will look familiar as set operations from discrete math. The binary type constructors $\to$ and $\times$ correspond exactly to mathematical functions and cartesian products, respectively. $A \to B$ is the type of a λ-abstraction with a formal parameter of type A and a body of type B. As for product types, we introduce syntax for pairs: if the expression e_1 has type A and e_2 has type B, then $\langle e_1, e_2 \rangle$ has type $A \times B$.

The classic flavors of typed λ-calculi are classified by whether the type annotations are explicit or implicit, and whether polymorphic types are allowed. In a polymorphic system, we allow universally-quantified *type variables*. For example, the identity function, $\lambda x.x$, could have type $INT \to INT$ or $REAL \to REAL$; it doesn't *care* what type the argument is. Nevertheless, a monomorphic system would require a *different* identity function for each type. With polymorphism, the type of the identity function is simply $\forall\alpha.\alpha \to \alpha$, where α is a *type variable*.

The most complex calculus used in the book is $\lambda^2\beta\eta\pi*$. The superscript 2 means that it is second-order. The β means it has the usual rule for β-reduction, described above. η stands for the *extensionality* axiom, namely $\lambda x.M\ x = M$, provided that x is not free in M. *pi* means we have products (pairs) and a few rules that come with them. And finally, the $*$ means we have a special *unit type* called $\mathbf{T}$ and the lone expression $*$ of type $\mathbf{T}$.

Chapter 2 presents some well-known results about confluence, normalization, and decidability. Confluence, the Church-Rosser property, says that "any two reduction sequences starting from the same term have a common reduct." There are often multiple possible *reduction strategies*, for example:

$$(\lambda x.a\ x)\ ((\lambda y.b\ y)\ c) \to_\beta a\ ((\lambda y.b\ y)\ c) \to_\beta a\ (b\ c)$$

compared with

$$(\lambda x.a\ x)\ ((\lambda y.b\ y)\ c) \to_\beta (\lambda x.a\ x)\ (b\ c) \to_\beta a\ (b\ c)$$

An expression is said to be in *normal form* when further β-reduction is not possible. Confluence guarantees a unique normal form for any expression, regardless which path we take in evaluating it. (Although some paths may be exempt by not terminating; consider $(\lambda x.x\,x)\,(\lambda x.x\,x)$, for which there is no normal form.)

Furthermore, a system is called *weakly normalizing* if for any expression, there exists a path (a sequence of reductions) that leads to normal form, and *strongly normalizing* if for any expression, *all* paths lead to the normal form.

Chapter 3 presents proofs of two theorems from Chapter 2, specifically strong normalization for $\beta\eta^2\pi*$ and $\beta^2\eta^2\pi*$. If the reader is already satisfied with these two theorems, then this chapter probably could be skipped.

Chapter 4, *First-Order Isomorphic Types*, marks the beginning of the theoretical contribution that makes software component retrieval possible. It begins by introducing a theory of equality $Th^1_{\times T}$ and a type-reduction relation $\mathcal{R}^1$:

1. $A \times (B \times C) > (A \times B) \times C$
2. $(A \times B) \to C > A \to (B \to C)$
3. $A \to (B \times C) > (A \to B) \times (A \to C)$
4. *etc.*

This relation suggests an obvious way to derive a *type normal form* for any type. The remainder of Chapter 4 applies this notion to $\lambda^1\beta\eta\pi*$ and proves completeness of $Th^1_{\times T}$.

Chapter 5, *Second-Order Isomorphic Types*, extends the same ideas to the second-order calculus, $\lambda^2\beta\eta\pi*$. This is a considerably more complex theoretical endeavor, and the author thoughtfully moved the more lengthy proofs into an appendix. He proves completeness of Th^2_T, the theory of second-order isomorphisms and, importantly, decidability of determining whether types A and B are isomorphic.

Chapter 6, *Isomorphisms for ML*, presents the motivating application for the theoretical framework established thus far. ML is a general-purpose programming language based on the typed lambda calculus. There is a formally defined dialect called Standard ML, but the variants used by Di Cosmo are called CAML and Caml-Light, developed at INRIA.

ML is a *type-safe* language, which means that programs which pass the compile-time type-checking cannot "go wrong" at runtime; i.e., no core dumps. This property effectively prevents a large class of programming errors. Moreover, the ML compiler *infers* the most general type for every expression and variable, so the programmer is not (usually) burdened with type annotations.

Di Cosmo describes a software library search system which has been implemented for CAML, based on isomorphisms of types. In strongly-typed functional languages such as ML and Haskell, the type of a function (or module) is considered to be a partial specification of its behavior. Therefore, a type signature may be used as a key to search libraries of existing software for a module that might accomplish a certain task.

However, there are generally several equivalent ways to specify a function, so syntactic equivalence of type signatures is unsuitable. Consider the variety of names and types chosen for the same piece of functionality in various languages (reproduced from page 37):

Language	Name	Type
ML (LCF)	`itlist`	$\forall\alpha.\forall\beta.(\alpha \to \beta \to \beta) \to \alpha\,\text{list} \to \beta \to \beta$
CAML	`list_it`	*same*
Haskell	`foldl`	$\forall\alpha.\forall\beta.(\alpha \to \beta \to \alpha) \to \alpha \to \beta\,\text{list} \to \alpha$
SML of New Jersey	`fold`	$\forall\alpha.\forall\beta.(\alpha \times \beta \to \beta) \to \alpha\,\text{list} \to \beta \to \beta$
Edinburgh SML	`fold_left`	$\forall\alpha.\forall\beta.(\alpha \times \beta \to \beta) \to \beta \to \alpha\,\text{list} \to \beta$

So, if a programmer were searching for this function in a library, exact matches of the type signature are too strict. Isomorphic types are a much more appropriate notion of equality.

A particular isomorphism for ML types, (**Split**), suggests an extension to the Hindley-Milner type inference algorithm traditionally used by ML. The extended system would infer the same type for two equivalent expressions that would be given different types by H-M. The example involves the functions PAIR $= (\lambda x.\,\langle x, x\rangle) : \forall\alpha.\alpha \to \alpha \times \alpha$ and ID $= (\lambda x.x) : \forall\alpha.\alpha \to \alpha$. When we apply PAIR to the argument ID, under H-M the result has type $\forall\alpha.(\alpha \to \alpha) \times (\alpha \to \alpha)$. But this is unnecessarily restrictive; it says that both identity functions in the pair must be evaluated at the same type. H-M plus **Split**) would infer $\forall\alpha.\forall\beta.(\alpha \to \alpha) \times (\beta \to \beta)$ which is slightly more general.

Finally, Chapter 7 very briefly sketches some future perspectives and related work. One important extension is for the search tool automatically to generalize type signatures provided by the

user as keys. For instance, if I am looking for a function to "flatten" a list of integer lists, I might enter the type INT LIST LIST-¿ INT LIST. Most implementations of FLATTEN would have the more general type 'a list list -¿ 'a list, and these two types are not isomorphic. (This example is trivial, but for others it could be less obvious how to generalize.)

Another suggestion is to apply this idea to tools for searching object-oriented libraries. Unfortunately, types become more complicated in the presence of inheritance (subtyping), and the isomorphism results do not hold in the presence of state.

3 Style

The first chapter is the most well-written. Not only does it present a friendly (albeit quick) introduction to types and typed λ-calculi, but it intuitively sketches the motive and results of the remaining chapters.

As for those, one might do well to read the first few pages of each chapter before sinking into the details. Motivation and definitions are generally presented up front, before descending into all the requisite formalisms and proofs.

4 Opinion

This book has some interesting insights, plus the noble goal of trying to apply programming language theory directly to a software engineering tool.

A more pragmatic reader, however, may not be convinced that Hindley-Milner types are *enough* of a specification of a function's behavior to be effective for retrieving useful modules from large, real-world software libraries. The FOLDL example is fine, but consider that *all* predicates on natural numbers (EVEN , ODD, PRIME, PERFECTSQUARE, POWEROFTWO) have exactly the *same* type, namely INT→. I do not deny that isomorphic types are an improvement over searching by name, but Di Cosmo does not try to convince us that it's a *significant* improvement for real-world libraries. (Every functional programmer already knows how FOLDL works in her favorite dialects.) Still, the ideas have been implemented successfully into CAML, which will make a nice proof of concept.