TACTIC-BASED MODELING OF COGNITIVE INFERENCE ON LOGICALLY STRUCTURED NOTATION

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Eric G. Aaron

August 2000

TACTIC-BASED MODELING OF COGNITIVE INFERENCE ON LOGICALLY STRUCTURED NOTATION

Eric G. Aaron, Ph.D.

Cornell University 2000

Computational (algorithmic) models of high-level cognitive inference tasks such as logical inference, mathematical inference, and decision making can have both theoretical and practical impact. They can improve our theoretical understanding of how people think and also provide practical direction for applications such as automated reasoning systems, systems attuned to user-interaction in decision-critical environments, and computer-aided education. To support those benefits, cognitive models need to be detailed, compositional, based in well-understood mathematics, and, to whatever extent possible, descriptively accurate. We introduce a new, interdisciplinary approach that could be used to develop cognitive models of high-level inference with these properties.

Two significant aspects of this approach are *tactics* and *eyetracking* methods. Tactics are used to express high-level inferences in fully formalized mathematics for automated theorem proving systems; eyetracking methods provide insight into real-time and microcognitive information processing by permitting analysis of the visual attention of people performing cognitive tasks. Combining tactics and eyetracking methods with traditional techniques from applied logic, artificial intelligence, and cognitive science can result in more deeply detailed and accurate cognitive models.

We demonstrate the feasibility of this new approach to modeling by describing its application to a calculational logic system that supports schematic reasoning via metalinguistic operations (such as textual substitution) without resorting to higher-order logic. We discuss several computational, psychological, and pedagogical insights that resulted from this approach, and we present

a detailed, tactic-based model of calculational logic inference. Specific results include: an explanation of calculational logic as a formalized metalogic; a tactic-based implementation of calculational logic inference; some pedagogical observations on the teaching of calculational logic; and experimental results that demonstrate that eyetracking methods can provide insight into theorem proving that could not be achieved by studies of written work alone.

Biographical Sketch

The author received the AB degree in Mathematics from Princeton University in 1992 and the MS in Computer Science from Cornell University in 1995.

To my parents, for making sure I could take any road and enjoy the ride

Acknowledgments

It is only right to begin by acknowledging the influence of Philip Johnson-Laird, who has earned my deepest admiration. He showed me respect and gave me guidance at a critical stage in my undergraduate life; I hope never to forget those lessons. He also introduced me to the possible cognitive science applications of Nuprl, planting the seeds of this dissertation. He is as great a Professor as I have ever known, and he has my unending gratitude.

My primary computer science collaborator on this project was Stuart Allen. I greatly appreciate the immense amount of time he spent discussing technical details with me; this dissertation would not exist without him. Chapter 3 is based on his idea of calculational logic as a formalized metalogic and adapted from a paper [AA99] that Stuart and I co-authored. He was also invaluable in assembling the appendix of this dissertation.

My primary cognitive science collaborator was Michael Spivey, whose energy and expertise made possible the eyetracking experiments reported in this dissertation. Chapter 8 is adapted from a paper [AS98] that Michael and I co-authored.

David Gries and Anil Nerode were supportive faculty advisors, making various stages of my graduate school experience much, much better than they would otherwise have been. I thank them both for their advice and criticism on this dissertation. I also appreciate the other faculty who provided support during this process, including Dexter Kozen, Robert Constable, Claire Cardie, and Frank Keil.

Cognitive Studies at Cornell was another significant part of my graduate school experience. Barbara Lust and Sue Wurster were stupendously supportive of me, and I greatly enjoyed working with them on various projects.

I cannot imagine having completed a project like this without the companionship and support of many great friends. Among those making an incalculable impact: Laura Sabel, whose intellectual camaraderie, technical expertise, and emotional support were utterly invaluable; Anne Vandeventer, who brought treasures to my life that I never can forget; Jennifer Welsh, who brings extraordinary beauty wherever she goes. Thanks also to Scott, Joel, Dexter, Bruce, Mike, Jan, Jim, Ben, Naomi & Joe, John, Will, Sachi, Karen, Elisa, Maribeth, Susannah, Vicky, Liz, Liz, Liz, and many others. Your friendship has added immeasurably to the dissertation process.

Steve: Could you ever know how much you helped me?

To my grandfather, Eli, who was unwavering in his deeply heartfelt support: You are far too wonderful for me to possibly express here. Thank you for everything.

As the last personal acknowledgment, I acknowledge that I cannot adequately thank my parents, Alan and Lynn. Words escape me; emotions and memories do not. Thank you, always and forever.

This research was supported by NSF grant GER-9454149. It was also supported by several awards from the Cornell Cognitive Studies Program.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Computation is an established framework for modeling processes of human cognition such as perception (vision, audition) and inference. The impact of computational *cognitive modeling* has been two-fold; it has helped improve our understanding of how people think[1] while also providing direction for practical applications in arenas such as computer vision, natural language processing, and character animation. Within our particular area of interest —modeling performance on high-level tasks such as logical inference, mathematical inference, and decision making— applications of cognitive modeling include systems better attuned to user interaction in decision-critical environments (e.g., [M$^+$]) and better computer-aided education (e.g., some applications of ACT-R [And93]).

Our research explores a new, interdisciplinary approach to cognitive modeling of high-level inference, combining complementary ideas from applied logic, artificial intelligence, and cognitive science. In this dissertation, we describe an application of this approach to a particular inference task, resulting in several psychological, pedagogical, and computational insights about that inference task as well as a cognitive model with many desirable properties. In this introductory chapter, we elaborate further on both the general approach and the details of its first application.

Working with high-level inference, we want to reason *about* cognitive models as well as *with* them, to prove things about the models as well as use them to predict behavior. We would like to understand the models on varying levels of abstraction, analyzing them on the level of conscious, high-level inferences as well as smaller (perhaps subconscious) inferences. Clearly, we also want the models to describe the things people actually do when performing that kind

[1] Cognitive modeling is not identical to artificial intelligence. Examples of computational cognitive models of high-level inference that have contributed to our understanding of how people think include symbolist/algorithmic approaches like Johnson-Laird's Mental Models theory [JL83] and Rips' PSYCOP [Rip94], as well as several connectionist models of language processing.

of high-level inference. For these reasons, we prefer cognitive models that are *compositional*, *detailed*, based in well-understood mathematics, and, to whatever extent possible, descriptively accurate. By *compositional*, we mean that an inference model can be understood in terms of components that correspond to sub-inferences; in this computational context, we consider compositionality and modularity to be similar concepts. Compositionality permits us to develop, upgrade, and prove things about inference models by working with component sub-structures that correspond to component sub-inferences. By *detailed*, we mean that models should accommodate low-level analysis, perhaps exposing sub-inferences that might not be immediately obvious. Reasons for cognitive models to be mathematically well-understood and descriptively accurate are clear: without those qualities, models lack descriptive and predictive value.

We observed that existing methods in automated theorem proving and eyetracking could be applied to cognitive inference modeling. The practice of using *tactics* (see [GMW79]) in automated reasoning allows fully formalized mathematical inferences to be expressed and manipulated at a level of abstraction away from primitive logical rules. Tactics are intended to capture high-level inferences, including those that people might naturally make in developing a proof, and cognitive modeling seems like a natural application for tactic-based automated reasoning systems. For our research, we used the tactic-based proof development system Nuprl [C$^+$86] as a platform for formalizing a high-level inference task and developing a tactic model of cognitive inference. We discuss Nuprl and tactics further in chapter 4.

By allowing experimenters to observe the gaze and visual focus of people performing cognitive tasks, eyetracking methods have provided significant insight into real-time information processing. They have been used in a wide variety of contexts, including high-level inference tasks like high school geometry [ES96] and military tactical decision making [M$^+$]. Although cognitive science has studied theorem proving in the past (e.g., [Mel94]), the application of eyetracking methods to theorem proving has not been thoroughly explored. Our experiments, described in chapter 8, support the idea that eyetracking can provide insights into theorem proving that studies of written work alone could not. These insights could be integrated into computational models of cognitive inference so that components of models could be experimentally tested for descriptive accuracy prior to implementation.

Just like general cognitive tasks vary in how well they can be analyzed by eyetracking, inference tasks vary in how well they support the use of tactics and eyetracking methods in developing cognitive models. In particular, our approach seems likely to be most beneficial when used to model performance on tasks that involve inference on some *logically structured notation*, i.e., a notation that emphasizes the structure on which inferences are made. Rather than fully tighten this loose definition/explanation, we elucidate it by giving negative and positive examples. Natural language English is *not* logically structured; for instance, untangling the quantifier structure of the sentence "Everyone hates someone" is not trivial. In contrast, successful formal logic notations —be they variations of standard predicate logic notations, pictorial systems such as Venn diagrams, or other expressive systems— are typically logically structured. They would not have gained acceptance had they not emphasized the aspects that people considered when performing inference.

Logically structured notation facilitates two major aspects of our approach to modeling. It makes it more plausible that we might be able to represent that notation in a formalized mathematical system and manipulate it using relatively simple tactics.[2] In addition, because the underlying structure used for inferences is visible, we can reasonably hope to gain cognitive insight by tracking the eye movements of people performing such inferences. For these reasons, we consider only the application of our tactics-and-eyetracking approach to modeling inference on logically structured notation.

To establish the feasibility of our approach, we needed to test it on a particular inference task; we chose theorem proving in calculational predicate logic, a variant of first-order predicate logic described by Gries and Schneider in their college-level textbook "A Logical Approach To Discrete Math" [GS93b]. (We provide a brief, self-contained introduction to calculational predicate logic in chapter 2.) It had several advantages for our purposes. Calculational logic had strong local support, being used for research and taught to students in our Cornell University community. Its significant visual component made it likely that eyetracking studies could yield interesting results, and there was a ready subject population for eyetracking experiments. Overall, it seemed a likely choice for a task on which our approach to cognitive inference modeling could prove useful.

By choosing a particular method/topic to model instead of a more general problem or theory, we were able to concentrate on the planning, semantic analysis, cognitive science, and automated reasoning techniques needed for our major accomplishments. We constructed a formal, compositional foundation for our model, and our calculational logic tactics provide an interesting perspective on areas of relative complexity or simplicity in the established pedagogical presentation of calculational logic. Based on our understanding of the cognitive inference of calculational logic, we implemented a tactic to perform and verify a significant body of calculational logic proof steps. We also demonstrated that eyetracking methods can indeed help develop cognitive inference models on high-level, logical inference tasks like theorem proving.

[2] Imagine the complexity of an automated reasoning system for English —not just some restricted or specialized subset, but the whole language. This seems implausible in part because English does not facilitate recognizing its logical structure.

Even more important than our particular results is the observation that our approach might achieve similar depth and breadth of analysis with other logically structured notations. The overall framework and techniques of our approach lend themselves to scientific, computational studies of issues of general interest. What does it mean for one inference (or method) to be simpler than another? What information is necessary to support a given inference? (Relatedly, what information do people actually *attend to* when making a given inference?) How can we characterize obvious inferences (an important issue when developing interactive environments)? What distinguishes novices from experts? These questions arise in many domains, and we believe that our approach might help answer them in contexts other than the one used for this dissertation. There is nothing unique to calculational logic or Nuprl that enables our research; they were simply convenient systems to use in demonstrating the feasibility of our general approach to modeling high-level logical inference.

Without de-emphasizing our belief that our modeling approach can be applied in many contexts, we also observe that calculational logic and Nuprl are well-suited for this feasibility demonstration. Nuprl's refinement logic embodies a method of problem solving derived from the cognitive models provided by Logic Theorist and GPS [NSS57, NS61, CKB84], and calculational logic inference patterns can be effectively represented in this framework. We further discuss tactics and this cognitive modeling aspect in chapter 9 on page 150.

Here is an overview of this dissertation:

- Chapter 2 is a brief, self-contained introduction to calculational logic and the textbook [GS93b]. Most of the remainder of the dissertation presumes that readers are familiar with this chapter.

- It was not immediately clear from [GS93b] that calculational logic could be readily explained using only conventional semantics. Chapter 3 contains our first major result: an explanation of calculational logic as a formalized metalogic. We later use this explanation as a foundation for our Nuprl formalization of calculational logic.

- Chapter 4 is a brief, self-contained introduction to Nuprl, tactics, and the tools we use to formalize calculational logic and our cognitive inference model.

- Chapters 5 and 6 describe our Nuprl formalizations of the language of calculational logic and the inferences done by people performing calculational logic, respectively. Chapter 6 discusses technical details of the implementation of our tactic model. The issue of how it can be understood as a cognitive model is not addressed until chapter 9.

These chapters also contain insights into the inferences necessary for calculational logic that are not mentioned in [GS93b].

- Chapter 7 contains some observations on pedagogy. Pedagogical analysis was not a primary goal of our research, but because we were formalizing and modeling an inference method taught in the college-level textbook [GS93b], it was not altogether surprising that some education-related observations emerged.

- Chapter 8 describes our eyetracking experiments, which demonstrate that eyetracking methods can provide insight into how people prove theorems that studies of written work alone could not offer.

- Chapter 9 contains a discussion of our tactics as a cognitive model. It also contains concluding remarks, including comments on other, related research.

Chapter 2

An Overview of Calculational Logic

We needed an inference system to study via eyetracking experiments and partially implement in Nuprl; the calculational logic described in [GS93b] was a sensible choice. It was commonly used by students in the Cornell community, thus providing a subject population for eyetracking experiments. Furthermore, because of the stylized format in which they are written, calculational proofs were easier for us to study than natural language proofs.

In this chapter, we provide a brief overview of calculational logic; it is not a complete specification. The goal is simply to familiarize readers with the basic ideas and characteristic patterns present in the book [GS93b]; we will expand upon and clarify these ideas in later chapters. We present this overview in layers. First, we give a very brief introduction to the calculational approach, enough to understand the remaining chapters but lacking many significant details about calculational logic itself. Following that is a more detailed description of calculational predicate logic, including axioms and inference rules as presented in [GS93b].

2.1 A Very Brief Overview

Calculational logic came about as a formalization of general calculational methods, which attempt to emphasize simple syntactic manipulations in problem solving. It emerged from work by computer scientists on the formal development of algorithms (see [Bac95, DS90, GS94] for historical notes and sources). Gries and Schneider present it, along with heuristics and guidelines for students, in a college-level textbook [GS93b] as a general technique for logic and discrete mathematics.

In this section, we review some of the basic elements underlying a particular method of proving equalities using a chain of equality-preserving rewrites. This proof method is based on a fundamental, commonly exploited observation about the language of calculational logic: If E and F are equal expressions, then substituting F for E in an expression does not alter the value of that expression. For example, since $(P \Rightarrow Q) = (\neg P \vee Q)$, we know $(R \wedge (P \Rightarrow Q)) = (R \wedge (\neg P \vee Q))$. This is justified by an inference rule of "substitution of equals for equals" called "Leibniz" in [GS93b]: because the premise $(P \Rightarrow Q) = (\neg P \vee Q)$ is previously proved true, we can substitute the expression on one side of the equality for the expression on the other ("substitute equals for equals"). This substitution process permits the construction of proofs in a formal manner that has the feel of ordinary calculation.

A typical calculational logic proof is based on a series of equality-preserving rewrites, each justified by inference rule Leibniz; users of calculational logic can prove a statement either by transforming the entire expression into a previously established theorem or by transforming one side of an equality into the other side. For example, a calculational proof step that establishes the equality $(R \wedge (P \Rightarrow Q)) = (R \wedge (\neg P \vee Q))$ might have the following form:

$$
\begin{array}{ll}
 & R \wedge (P \Rightarrow Q) \\
= & \quad \langle P \Rightarrow Q \equiv \neg P \vee Q \rangle \\
 & R \wedge (\neg P \vee Q)
\end{array}
$$

where the expression in angle brackets, called a "hint," refers to the premise of the instance of Leibniz that yields the conclusion $(R \wedge (P \Rightarrow Q)) = (R \wedge (\neg P \vee Q))$. (The equivalence sign $\equiv$ in the hint $P \Rightarrow Q \equiv \neg P \vee Q$ represents equality on booleans; it has the lowest precedence of any logical operator in calculational logic.[1] Further elaboration on calculational logic conventions regarding $=$ and $\equiv$ is unnecessary for now.) Hints can be logical statements or names that refer to them. For instance, we could have written "Definition of Implication" as the hint in the example step given above, since that is the name associated with the equivalence $P \Rightarrow Q \equiv \neg P \vee Q$.

The following proof establishes the theoremhood of $p \vee q \equiv p \vee \neg q \equiv p$. We read it as $(p \vee q \equiv p \vee \neg q) \equiv p$ and prove $p \vee q \equiv p \vee \neg q$ equal to p using equality-preserving rewrites. (Boolean equality $\equiv$ is associative on its arguments, so the statement $p \vee q \equiv p \vee \neg q \equiv p$ need not be explicitly parenthesized.)

$$
\begin{array}{ll}
 & p \vee q \equiv p \vee \neg q \\
= & \quad \langle \text{Distr. of } \vee \text{ over } \equiv,\ p \vee (q \equiv r) \equiv p \vee q \equiv p \vee r \rangle \\
 & p \vee (q \equiv \neg q) \\
= & \quad \langle \neg q \equiv q \equiv \mathit{false} \rangle \\
 & p \vee \mathit{false} \\
= & \quad \langle \text{Identity of } \vee,\ p \vee \mathit{false} \equiv p \rangle \\
 & p
\end{array}
$$

[1]The order of precedence for operators is: $=$; $\wedge, \vee$; $\Rightarrow$; $\equiv$. As an illustration of the different precedences of $=$ and $\equiv$, the expressions $P \Rightarrow Q = \neg P \vee Q$ and $P \Rightarrow Q \equiv \neg P \vee Q$ have different meanings; only the second one represents the traditional relation between implication and disjunction.

Each of the three equalities established in the proof above is justified by inference rule Leibniz. The conclusion follows from those component equalities by transitivity of equality, which is also an inference rule of calculational logic.

This section was intended as only a cursory overview of the most basic aspects of calculational logic; the next section provides more technical detail. We do not provide a thorough, formal explanation of the logic until chapter 3.

2.2 A More Detailed Overview

Throughout this dissertation, we use the phrase "calculational predicate logic" to refer to the predicate logic (with equality and function symbols) presented in chapters 3, 8, and 9 of [GS93b]. This does not include many elements of calculational logic that are only peripherally important to our research, such as applications of calculational logic to discrete mathematics (see [GS93b] and [GS94]) and proof methods other than simply using equality-preserving rewrites (see [GS93b, chap. 4]). We do not elaborate on such elements in this chapter. Although we were mindful of extensions and applications of calculational predicate logic —indeed, some design decisions in our formalization of it were purposefully made to accommodate extensions beyond the current scope— this chapter focuses only on aspects of calculational logic that are centrally relevant to our research.

Calculational predicate logic uses notation similar to typical predicate logics, but a few symbols merit discussion. We have already mentioned the use of $\equiv$ as equality restricted to booleans, an associative operation; in contrast, the general equality symbol $=$ applies to all types (including booleans), so it is not associative. Calculational logic exploits this difference, treating $\equiv$ associatively (as illustrated in the previous section) and treating $=$ conjunctionally, so $a = b = c$ is read as $(a = b) \wedge (b = c)$. Note that in the simple binary case for boolean p, q, the expressions $p = q$ and $p \equiv q$ have the same value and are used interchangeably in [GS93b]. It is only in more complex cases where differences emerge: for $p, q, r{:}\mathbb{B}$, $p \equiv q \equiv r$ is not the same as $p = q = r$.

In addition, textual substitution is of primary importance to calculational logic. Indeed, as we suggest in chapter 3, many features of calculational logic can be seen as motivated by a desire to formally reason about textual substitution. We use the notation $E[v := P]$ to stand for capture-avoiding textual substitution, where E and P are logic expressions and v is a variable.

In the expected way, a theorem of calculational logic is either an instance of an axiom[2] or a conclusion of an instance of an inference rule whose premises

[2]We do not yet distinguish between axioms and axiom schemas. We clarify this and other matters in our explanation of calculational predicate logic in chapter 3.

Table 2.1: Axioms of calculational propositional logic

(2.1) **Associativity of $\equiv$:** $((p \equiv q) \equiv r) \ \equiv \ (p \equiv (q \equiv r))$
(2.2) **Symmetry of $\equiv$:** $p \equiv q \equiv q \equiv p$
(2.3) **Identity of $\equiv$:** $\mathit{true} \equiv q \equiv q$

(2.4) **Definition of *false*:** $\mathit{false} \equiv \neg \mathit{true}$
(2.5) **Distributivity of $\neg$ over $\equiv$:** $\neg(p \equiv q) \ \equiv \ \neg p \equiv q$
(2.6) **Definition of $\not\equiv$:** $(p \not\equiv q) \ \equiv \ \neg(p \equiv q)$

(2.7) **Symmetry of $\vee$:** $p \vee q \ \equiv \ q \vee p$
(2.8) **Associativity of $\vee$:** $(p \vee q) \vee r \ \equiv \ p \vee (q \vee r)$
(2.9) **Idempotency of $\vee$:** $p \vee p \ \equiv \ p$
(2.10) **Distributivity of $\vee$ over $\equiv$:** $p \vee (q \ \equiv \ r) \ \equiv \ p \vee q \ \equiv \ p \vee r$
(2.11) **Excluded Middle:** $p \vee \neg p$

(2.12) **Golden rule:** $p \wedge q \ \equiv \ p \ \equiv \ q \ \equiv \ p \vee q$

(2.13) **Definition of Implication:** $p \ \Rightarrow \ q \ \equiv \ p \vee q \ \equiv \ q$
(2.14) **Consequence:** $p \ \Leftarrow \ q \ \equiv \ q \ \Rightarrow \ p$

are previously proved theorems. In the following subsections, we present the axioms and inference rules of calculational predicate logic, starting with its propositional logic component.

2.2.1 Calculational propositional logic

The axioms of the propositional portion of calculational logic are given in Table 2.1, ordered and grouped according to their presentation in [GS93b]. Equivalence is introduced first. Because the first axiom says that equivalence is associative, we may eliminate parentheses from sequences of equivalences that follow. For example, Symmetry of $\equiv$ (2.2) is given as $p \equiv q \equiv q \equiv p$. In addition, Associativity of $\equiv$ reduces the number of axioms that need to be listed. For example, we may parse (2.2) in five ways: $((p \equiv q) \equiv q) \equiv p$, $(p \equiv q) \equiv (q \equiv p)$, $(p \equiv (q \equiv q)) \equiv p$, $p \equiv ((q \equiv q) \equiv p)$, and $p \equiv (q \equiv (q \equiv p))$.

The calculational logic definition of conjunction, called the Golden rule (2.12), exploits this flexibility with $\equiv$. To see that (2.12) is valid, check its

Table 2.2: Inference rules of calculational propositional logic

$$\textbf{Leibniz:}\quad \frac{P = Q}{E[v := P] = E[v := Q]}$$

$$\textbf{Transitivity:}\quad \frac{P = Q,\ Q = R}{P = R}$$

$$\textbf{Equanimity:}\quad \frac{P,\ P \equiv Q}{Q}$$

truth table or else use associativity and symmetry of $\equiv$ to rewrite it as

$$p \equiv q \quad \equiv \quad p \wedge q \equiv p \vee q \quad .$$

Now, it may be recognized as the statement that two booleans are equal iff their conjunction and disjunction are equal.

The inference rules of calculational propositional logic are given in Table 2.2. Inference rule Leibniz is the justification for substitution of equals for equals. Transitivity is used to derive a desired conclusion from a chain of equalities, where each equality in that chain was itself derived from Leibniz. Inference rule Equanimity justifies the previously mentioned proof method of proving a theorem by reducing it to a previously proved theorem. For example, to prove that an expression Q is a theorem, applications of Leibniz and Transitivity might prove Q equal to P, for some previously proved theorem P. Then, Equanimity establishes that Q is itself a theorem.

Here is a simple proof (taken from [GS93b, page 48]) that illustrates the use of these inference rules. It derives the theoremhood of $\neg p \equiv p \equiv \mathit{false}$ by reducing it to Axiom Definition of False (2.4).

$$\begin{array}{ll}
 & \neg p \equiv p \equiv \mathit{false} \\
= & \quad \langle \text{Distributivity of } \neg \text{ over } \equiv,\ \neg(p \equiv q) \equiv \neg p \equiv q \text{, with } q := p \rangle \\
 & \neg(p \equiv p) \equiv \mathit{false} \\
= & \quad \langle \text{Identity of } \equiv \text{, with } q := p \rangle \\
 & \neg \mathit{true} \equiv \mathit{false} \qquad \text{—theorem (2.4)}
\end{array}$$

The above proof is based on a chain of two equalities, each established as an instance of inference rule Leibniz. For instance, the first one, $(\neg p \equiv p \equiv \mathit{false}) = (\neg(p \equiv p) \equiv \mathit{false})$, is the conclusion of an instance of Leibniz with $(\neg p \equiv p) = (\neg(p \equiv p))$ as its premise. That premise is itself a theorem, an instance of Axiom Distributivity of $\neg$ over $\equiv$. (Note the use of $=$ instead of $\equiv$; as previously mentioned, on booleans, they are the same in the binary case.) The other equality in the chain, $(\neg(p \equiv p) \equiv \mathit{false}) = (\neg \mathit{true} \equiv \mathit{false})$, is similarly established using Leibniz.

Then, with those two equalities as premises, inference rule Transitivity establishes $(\neg p \equiv p \equiv \mathit{false}) = (\neg \mathit{true} \equiv \mathit{false})$. Since $\neg \mathit{true} \equiv \mathit{false}$ is a theorem and $(\neg p \equiv p \equiv \mathit{false}) = (\neg \mathit{true} \equiv \mathit{false})$ has just been derived, these form the premises of an instance of inference rule Equanimity, deriving the theoremhood of $\neg p \equiv p \equiv \mathit{false}$, our original goal.

Proofs generally follow the format demonstrated above, and applications of Leibniz and Transitivity are left implicit; their use is understood. Use of Equanimity is indicated by text, as in the above example, stating that the expression at one end of the chain of equalities is a previously proved theorem.

2.2.2 Calculational predicate logic

In its treatment of quantification, the calculational approach in discrete mathematics text [GS93b] utilizes the observation that, in many common contexts, there is a similarity between iterated mathematical operators (such as summation $\sum$ and product $\prod$) and traditional universal and existential quantifiers. It therefore employs a single, general notation for all iterated operators and quantifiers.

Let $\star$ be a binary, associative, and symmetric operator that has an identity.[3] The notation

(2.15) $(\star i \mid R.i : P.i)$

denotes the accumulation of values $P.i$, using operator $\star$, over all values i for which range predicate $R.i$ holds. If range $R.i$ is true, we may write the quantification as $(\star i \mid : P.i)$.

Here are some examples of this notation; **gcd** is the greatest common divisor operator.

$$\begin{array}{rcl} (+i \mid 1 \le i \le 3 : i^2) & = & 1^2 + 2^2 + 3^2 \\ (\wedge x \mid 3 \le x < 7 \wedge \mathit{prime}.x : b[x] = 0) & \equiv & b[3] = 0 \wedge b[5] = 0 \\ (\,\mathbf{gcd}\ i \mid 2 \le i \le 4 : i^2) & = & 2^2\ \mathbf{gcd}\ 3^2\ \mathbf{gcd}\ 4^2 \end{array}$$

The calculational approach in [GS93b] exploits this single notation (2.15) for quantification and other iterated operators by presenting a uniform treatment of bound variables, scope of variables, free variables, and textual substitution that encompasses all of those operators. Furthermore, many primary axioms of calculational predicate logic are given using this notation because they hold for all such iterated operations, not just predicate logic quantifiers (see Table 2.3).

[3]This is actually an oversimplification. For instance, if $\star$ is associative and symmetric but has no identity, then instances of the axioms of Table 2.3 that have a false range do not hold. In addition, we restrict ourselves to instances where the accumulation converges to a value, excluding cases such as $(\equiv x \mid : \mathit{false})$.

Table 2.3: Axioms for quantification

(2.16) **Empty range:** $(\star x \mid \mathit{false} : P) \;=\; (\text{the identity of } \star)$

(2.17) **One-point rule:** $(\star x \mid x = E : P) \;=\; P[x := E]$

(2.18) **Distributivity:** $(\star x \mid R : P) \star (\star x \mid R : Q) \;=\; (\star x \mid R : P \star Q)$

(2.19) **Range split:** Provided $\neg(R \wedge S)$ holds or $\star$ is idempotent,

$$(\star x \mid R \vee S : P) \;=\; (\star x \mid R : P) \star (\star x \mid S : P)$$

(2.20) **Range split:**

$$(\star x \mid R \vee S : P) \star (\star x \mid R \wedge S : P) \;=\; (\star x \mid R : P) \star (\star x \mid S : P)$$

(2.21) **Interchange of dummies:**

$$(\star x \mid R : (\star y \mid Q : P)) = (\star y \mid Q : (\star x \mid R : P))$$

(2.22) **Nesting:** $(\star x, y \mid R \wedge Q : P) = (\star x \mid R : (\star y \mid Q : P))$

(2.23) **Dummy renaming:** $(\star x \mid R : P) = (\star y \mid R[x := y] : P[x := y])$

Note: The usual caveats concerning the absence of free occurrences of dummies in some expressions are needed to avoid capture of variables. Further, some of the axioms require the ranges of quantifications to be finite or $\star$ to be idempotent.

We also add one more inference rule to the calculational system to accommodate this notation:

$$\textbf{Leibniz:}\quad \frac{R \;\Rightarrow\; P = Q}{(\star x \mid R : E[z := P]) = (\star x \mid R : E[z := Q])}$$

We already have an inference rule named "Leibniz," but we also use the name for this rule, which also permits substitution of equals for equals. This one simply specifies a different condition for substitution into the body of a quantifier expression.

Calculational predicate logic requires just a few more axioms that deal specifically with universal and existential quantification —see Table 2.4. When the operator $\star$ is $\wedge$ (corresponding to universal quantification) or $\vee$ (existential quantification), calculational logic instead uses the more conventional notation $\forall$ or $\exists$ to represent the operator.

Table 2.4: Additional axioms for predicate calculus

(2.24) **Trading:** $(\forall x \mid R : P) \equiv (\forall x \mid : R \Rightarrow P)$

(2.25) **Distributivity of $\vee$ over $\forall$:**

$$P \vee (\forall x \mid R : Q) \equiv (\forall x \mid R : P \vee Q)$$

(2.26) **(Generalized) De Morgan:** $(\exists x \mid R : P) \equiv \neg(\forall x \mid R : \neg P)$

2.3 Intended Purpose of this Overview

The description of calculational logic in this chapter is admittedly shallow —for instance, we acknowledged failing to adequately distinguish the notions of axiom and axiom schema— but we trust that readers can get a general feel for the moves of calculational logic despite this imperfection. Such a passing familiarity is sufficient to understand the design and execution of the eyetracking experiments described later, in chapter 8. In those experiments, we simply analyzed the eye movements of people as they were constructing calculational proofs. Some familiarity with the methods of calculational logic is essential background for that chapter; a deep mathematical or semantic understanding is unnecessary.

While the information present in this chapter provides the background for the eyetracking portion of the research in this dissertation, the information *not* present here —the unanswered questions and unresolved issues— serves to motivate our formalization of calculational logic. After the cursory introduction in this chapter, readers may wonder about the validity of some of the inference rules, or how cleanly calculational logic incorporates a metalinguistic operation like textual substitution. Answers and clarifications on issues like these are in chapter 3, and our Nuprl implementation of calculational logic inference follows from the framework in that chapter.

Chapter 3

Justifying Calculational Logic by a Conventional Metalinguistic Semantics

3.1 Introduction

The calculational approach to predicate logic represents an alternative to higher-order logic for escaping the restrictions of first-order logic. It combines selected aspects of metamathematics and standard predicate logic (with equality and function symbols) in a single, formal system. It also presents the challenge of justifying its atypical metalinguistic features; for instance, calculational logic is intended to reason about textual substitution, but a conventional object language does not permit assertions about substitution. In this chapter, we demonstrate that a conventional metalinguistic semantics can provide an adequate foundation for the purposefully non-standard language of calculational logic. In addition, we enumerate the non-standard elements of the language of the calculational logic of [GS93b]. We conclude that, although calculational proofs may seem unusual and highly heuristic, the fundamental inferences of calculational logic seem quite straightforward in the context of our metalinguistic explanation, and they can be readily justified by conventional methods.

Typically, logicians are formal in their description of first-order predicate logic (with equality and function symbols) but informal in their use of metalanguage to describe properties of first-order logic, even when applying the metalanguage to show some formula is a theorem, as in [HC68]. For example, the following metatheorem,

(3.1) The universal quantification $(\forall x)x = e \;\Rightarrow\; P$, where x does not occur free in expression e, is equivalent to $P[x := e]$.

is generally stated in English and would be proved informally, if at all. (Here $P[x := e]$ denotes a copy of formula P in which each free occurrence of x is replaced by expression e in the usual capture-avoiding fashion.)

In the calculational approach to logic as presented in [GS93b] —see also the IPL issue [Bac95], which is devoted to this approach— such metatheorems are written in a formal notation as if they were part of an extended first-order predicate logic. For example, (3.1) is treated as an axiom:

(3.2) **One-point rule:** Provided x does not occur free in e, $((\forall x)x = e \Rightarrow P) \equiv P[x := e]$.

The calculational predicate logic in [GS93b] is designed to permit formal reasoning about metalinguistic operations such as $P[x := e]$, so its proofs incorporate both metamathematical manipulation and traditional predicate logic steps. However, calculational logic systems developed thus far have not been provided the kind of theoretical, formal foundation that is needed in order to be sure that the systems are technically correct. The purpose of this chapter is to provide such a foundation.

To illustrate the use of the calculational approach, we provide an example of the kinds of properties that can be easily expressed and proved. Consider, first, the following statement about integer arithmetic:

$$\textstyle\sum_{i=2}^{10} 2i = \sum_{j=0}^{8} 2(j+2),$$

which in the calculational notation of [GS93b] is written as

$$(+i \mid 2 \leq i \leq 10 : 2i) = (+j \mid 0 \leq j \leq 8 : 2(j+2)).$$

(Thus, the range of dummy i appears between " $\mid$ " and " : ," and the expression being "accumulated" appears after " : ".) The metatheorem that justifies this equality is rarely stated. But in [GS93b], it is given explicitly and concisely as:

(3.3) **Change of dummy:** Provided $\neg occurs(“j”,“R, P”)$ and function f has an inverse, $(+i \mid R : P) = (+j \mid R[i := f(j)] : P[i := f(j)])$.

The syntactic condition $\neg occurs(Vs, Es)$ is true exactly when no variable on list Vs occurs free in any expression on list Es.

Furthermore, in [GS93b], the theorem is stated in terms of a general operator $\star$ instead of the addition operator $+$; the theorem holds for various binary operators $\star$ that are associative, symmetric, and have an identity. (We do not claim here that it holds for all such binary operators, because we do not know a semantics for infinite iteration in general; we restrict $\star$ to $\vee$ and $\wedge$ for the purposes of this chapter.)

This theorem and its proof (Figure 3.1, taken verbatim from [GS93b]) incorporate many features of the calculational approach. The theorem is not only succinctly and formally stated, it is also succinctly and memorably proved. By memorable, we mean that at each step in developing the proof, the form of the proof practically determines the next step; there is no need for tricks or mnemonics, the proof's form itself is a guide to its construction.

The proof of Theorem Change Of Dummy also integrates the primary notational and mathematical novelties of the calculational predicate logic in [GS93b], non-standard aspects such as a generalized treatment of quantification, formal reasoning about textual substitution, formal reasoning about syntactic properties such as free occurrences of variables in expressions, and a proof format that leaves many inferences implicit. Because of this, we chose this proof as a concrete example to help illustrate our metalinguistic foundation for calculational predicate logic. In the next section, we introduce and motivate our metalinguistic reading of this proof.

In later sections, we give a complete, technical exposition of our formal foundation, applying it to explain the particular component propositions of the Change of Dummy proof as well as other aspects of the calculational logic presented in chapters 3, 8, and 9 of the discrete mathematics textbook [GS93b].

3.2 An Example Metalinguistic Explanation

The proof of Change of Dummy shown in Figure 3.1 is a chain of seven equalities, where each equality is justified using inference rule "substitution of equals for equals" (called "Leibniz" in [GS93b]). In English, this inference rule is: if $X = Y$ is a theorem, then so is $P = Q$, where Q is the result of replacing some occurrences of X in P by Y. For example, the first line of the proof of Change of Dummy is P, the third line is Q, and the second line refers to the premise $X = Y$ (within braces "$\langle$" and "$\rangle$"). In this case, the premise is the instance $(\star x \mid x = f.y : P) = P[x := f.y]$ of axiom One-point rule.

The transitivity of equality-derivability —known simply as "Transitivity" in [GS93b]— is an inference rule of calculational logic. (Informally, the rule states that if $A = B$ and $B = C$ are theorems under the same conditions, so is $A = C$; we will be more precise about what this means later in the chapter.) Using it six times, we conclude that the formula on the first line of the proof equals the formula on the last line.

This calculational system is quite different from conventional predicate logics. First, it relies heavily on inference rule substitution of equals for equals, instead of modus ponens. More importantly, it extends a conventional first-

Theorem Change of Dummy. Provided $\neg occurs(\text{“}y\text{”}, \text{“}R, P\text{”})$ and function f has an inverse, $(\star x \mid R : P) = (\star y \mid R[x := f.y] : P[x := f.y])$.

Proof. We start with the right side of $(\star x \mid R : P) = (\star y \mid R[x := f.y] : P[x := f.y])$ and show it is equal to the left side.

$$
\begin{array}{ll}
 & (\star y \mid R[x := f.y] : P[x := f.y]) \\
= & \quad \langle \text{One-point rule (8.14)} \\
 & \quad \text{—Quantification over } x \text{ has to be introduced. The One-} \\
 & \quad \text{point rule is the } \textit{only} \text{ theorem that can be applied at first.} \rangle \\
 & (\star y \mid R[x := f.y] : (\star x \mid x = f.y : P)) \\
= & \quad \langle \text{Nesting (8.20) —Moving dummy } x \text{ to the outside} \\
 & \quad \text{gets us closer to the final form.} \rangle \\
 & (\star x, y \mid R[x := f.y] \wedge x = f.y : P) \\
= & \quad \langle \text{Substitution (3.84a) —} R[x := f.y] \text{ must be removed} \\
 & \quad \text{at some point. This substitution makes it possible.} \rangle \\
 & (\star x, y \mid R[x := x] \wedge x = f.y : P) \\
= & \quad \langle R[x := x] \equiv R; \text{ Nesting, } \neg occurs(\text{“}y\text{”}, \text{“}R\text{”}) \\
 & \quad \text{—Now we can get a quantification in } x \text{ alone.} \rangle \\
 & (\star x \mid R : (\star y \mid x = f.y : P)) \\
= & \quad \langle x = f.y \equiv y = f^{-1}.x \text{ —This step prepares for the} \\
 & \quad \text{elimination of } y \text{ using the One-point rule.} \rangle \\
 & (\star x \mid R : (\star y \mid y = f^{-1}.x : P)) \\
= & \quad \langle \text{One-point rule (8.14)} \rangle \\
 & (\star x \mid R : P[y := f^{-1}.x]) \\
= & \quad \langle \text{Definition of textual substitution —} \neg occurs(\text{“}y\text{”}, \text{“}P\text{”}) \rangle \\
 & (\star x \mid R : P)
\end{array}
$$

Figure 3.1: Proof of Theorem Change of Dummy. Details of the cited premises One-point rule (8.14), Nesting (8.20), and Substitution (3.84a) are presented in section 3.7.

order language by incorporating elements that are typically accounted for as metamathematics about the logic. This can be seen even in Theorem Change of Dummy itself: the notation for textual substitution —a metamathematical concept— appears in what is purported to be a formula of the logic.

In light of this, a metalinguistic reading of the proof seems like the simplest way to explain it. The theorems and proofs in calculational logic are not typical predicate logic theorems and proofs; they are metatheorems and metaproofs *about* theorems and proofs in a typical predicate logic.

To arrive at this conclusion, we need to formalize and thus understand this calculational system in a more precise manner than is done in [GS93b]. We begin by defining a first-order predicate logic that we call the *object language*. The object language does *not* contain metamathematical concepts like substitution. Its purpose is only to serve as a concrete object level for the metalinguistic development that follows.

Second, we formalize the *data language*, so named because it represents the actual data that users manipulate when stating or proving theorems in [GS93b]. In the data language, expressions may be *object-expression valued*. For example, a variable P in the data language might range over expressions of the object language, and data-level expressions like $P \wedge Q$ would be object-expression valued, denoting the object-language expression constructed by applying the object-level conjunction operator to the object expressions denoted by P and Q. The other signs for logical operators of our first-order object language are represented in the data language in a similar fashion.

We can illustrate the difference between object language and data language using two expressions that appear to stand for function application in Theorem Change of Dummy. First, consider the expression $f.y$, a data-language level term. f and y are variables in the data language; they are not themselves a function and an argument. Instead, $f.y$ denotes $Ap_O(f;y)$, the object-level term constructed by the object-level function application operator Ap_O and the object expressions referred to by f and y. That is, the data-level expression $f.y$ is object-expression valued. The data-level variable f does not range over data-level functions, but object expressions.

Now consider the expression $occurs(\text{“}x\text{”}, \text{“}P\text{”})$, which is intended to have the meaning “each variable in list x (of variables) occurs free in at least one expression in list P (of expressions)”. In [GS93b], the two arguments of occurs are quoted in order to make clear that the arguments are not the values of x and P but the lists of variables and expressions themselves. However, at the data-language level, we write this simply as $occurs(x, P)$, where x is a term that stands for a list of object-language variables and P is a term that stands for a list of object expressions. The boolean-valued expression $occurs(x, P)$ then works in the ordinary way, applying function $occurs$ to the terms x and

P. It does not denote an object-level term the way $f.y$ does in the above paragraph.

Similarly, the (inherently metalinguistic) textual substitution operation $E[V := P]$ is an object-expression valued operation in the data language. With data-level variables E, V, P referring to object expression e, list of object variables vs, and list of object expressions ps, respectively, $E[V := P]$ denotes an object expression: a copy of e in which all free occurrences in e of the variables on vs have been replaced by the corresponding expressions on ps using simultaneous, capture-avoiding substitution.

With this introduction, we are ready to begin defining the object language and then the data language.

3.3 The Object Language

The design of the object language should be broad enough to include the fundamental forms on which the abstractions of the data language are built as well as general enough to be expanded to accommodate other aspects (i.e., other than predicate logic) of the calculational approach to discrete mathematics in [GS93b]. In the object language, we present a predicate logic in which each model contains a domain of discourse, a typing environment, and a valuation.

Throughout, $\mathbb{B}$ denotes the set $\{$ False , True $\}$.

3.3.1 Object language syntax

(3.4) **Definition.** OV is a denumerable class of *object variables* —identifiers used in the object language as propositional variables, predicate symbols, function symbols, and variables over individuals. As in ordinary informal practice, we make no *syntactic* distinction between quantifiable variables and propositional variables, or indeed between these and uninterpreted function and predicate symbols. Such distinctions will be introduced with typing environments in section 3.3.2 on object language semantics.

(3.5) **Definition.** OE is the class of *object expressions*, consisting of the object variables in OV, the constant $\mathbf{f}_O$, and all instances of the distinct operations $Ap_O(E1; E2)$, $E1 \Rightarrow_O E2$, $E1 =_O E2$, and $(\forall_O x.E1)$, where $E1, E2 \in OE$ and $x \in OV$.

Ap_O is intended to stand for function application. We express multi-place predicates and functions by currying with Ap_O. For example, an application of two-place predicate P to $E1$ and $E2$ would be $Ap_O(Ap_O(P; E1); E2)$.

The infix operator $=_O$ stands for equality. Note that, as in [GS93b], equality is applied to both individual and propositional expressions.

(3.6) **Definition.** We also introduce the following useful syntactic abbreviations:

- $\neg_O P == P \Rightarrow_O f_O$
- $t_O == \neg_O f_O$
- $P \wedge_O Q == \neg_O(P \Rightarrow_O \neg_O Q)$
- $P \equiv_O Q == (P \Rightarrow_O Q) \wedge_O (Q \Rightarrow_O P)$
- $P \vee_O Q == \neg_O P \Rightarrow_O Q$
- $\exists_O x.P == \neg_O(\forall_O x.\neg_O P)$

We use standard conventions for parenthesization with well-understood connectives.

The Gries & Schneider text [GS93b] treats quantifiers in a more complex and abstract manner than is represented above. It is not necessary to build this into our object syntax; we discuss it more fully in a later section.

In fact, it barely matters what the object syntax is. The choice of primitive operators for the object language presented here was not implicit from the logic in [GS93b], and the method of inference we aim to explain does not work directly on object expressions at all, but on a metalanguage about object expressions. We chose this particular object language only for concreteness of example.

3.3.2 Object language semantics

The semantics of the object language is given with respect to a domain of individuals D, over which we define quantification.

(3.7) **Definition.** The types of values that object (sub)expressions may denote have the forms

$$D(n) \rightarrow D \text{ or } D(n) \rightarrow \mathbb{B}$$

where (for $A \in \{D, \mathbb{B}\}$) $D(0) \rightarrow A$ is A and $D(n+1) \rightarrow A$ is $D \rightarrow (D(n) \rightarrow A)$. That is, $D(n) \rightarrow A$ is the type of curried n-ary A-valued functions.

We let $K(D)$ be the collection of all these types $D(n) \rightarrow A$, for $A \in \{D, \mathbb{B}\}$.

(3.8) **Definition.** A *model* of the object language is a triple $\langle D, \sigma, V \rangle$ where

- D, a *domain of discourse*, is a non-empty collection of values.
- $\sigma{:}OV \to K(D)$ is a *typing environment* function that assigns a type in $K(D)$ to every object variable.
- $V{:}(\Pi x{:}OV.\ \sigma(x))$ is a *valuation* function; the binding Π notation allows us to express that it assigns a value from type $\sigma(x)$ to each variable x.

So, a model consists of a domain and an assignment of types and values to variables. Where assignments to variables are concerned, we use superscripts and subscripts to represent updating. For instance, for valuation V, V_d^x is the same as V except that it assigns d to variable x. We also follow this convention for typing environments and other objects throughout the chapter.

We now define a relation $Osem$ to give the semantics of object expressions.

(3.9) **Definition.** For a domain of individuals D, typing environment $\sigma{:}(OV \to K(D))$, valuation $V{:}(\Pi x{:}OV.\ \sigma(x))$, $e{:}OE$, $T{:}K(D)$, and $v{:}T$, we recursively define $Osem$ by the clauses below, where $P, Q \in OE$. We intend $Osem(D, \sigma, V, e, v, T)$ to hold exactly when expression e has type T and value $v \in T$ under model $\langle D, \sigma, V \rangle$.

To simplify notation, we elide the arguments D, σ, V from an $Osem(\cdots)$ expression where it is obvious what is meant. In clauses where these terms are directly manipulated, they are included. We also treat each clause in the definition as closed by universally quantifying over any apparently free variables.

1. $\forall x{:}OV.\ Osem(x, V(x), \sigma(x))$
2. $Osem(Ap_O(P; Q), F(a), T)$
   if $Osem(P, F, D \to T)$ and $Osem(Q, a, D)$.
3. $Osem(f_O,\ \text{False}\ , \mathbb{B})$.
4. $Osem(P \Rightarrow_O Q, q \text{ if } p, \mathbb{B})$ if $Osem(P, p, \mathbb{B})$ and $Osem(Q, q, \mathbb{B})$.
5. $Osem((\forall_O x.\ P), (\forall d{:}D.\ g(d)), \mathbb{B})$ if $\forall d{:}D.\ Osem(\sigma_D^x, V_d^x, P, g(d), \mathbb{B})$.
6. $\forall T{:}\{D, \mathbb{B}\}, p, q{:}T.$
   $Osem(P =_O Q, p = q, \mathbb{B})$ if $Osem(P, p, T)$ and $Osem(Q, q, T)$.

This semantics has a typical form, although some of its features are not usually encountered in the language of a first-order predicate logic. Here, we offer a few brief explanatory notes.

Clause 1 reflects our decision to allow any identifiers to be used with any type; typing is assigned by σ . This formalizes the practice in [GS93b], which seems practical and natural.

Clause 2 stipulates that the object language sign for function application denotes actual function application (similarly for the usual boolean functions related to clauses 3-5). Note that clauses 1 and 2 can be applied to expressions of various function types, while the other clauses can be applied only to boolean or individual-valued expressions.

Clause 6 is unusual for first-order predicate logic because it allows equalities between booleans as well as between individuals. This is persistently exploited by the methods of calculational logic in order to apply equality lemmas and rules to equivalences (biconditionals). Note that the only difference between $P =_O Q$ and $P \equiv_O Q$ is that the latter has a value only when its arguments have boolean values.

Despite its few novelties, the object language is sensible in standard ways, as exemplified by the following theorems, whose straightforward proofs we omit.

(3.10) **Theorem.** $Osem(D, \sigma, V, e, v, T)$ defines v as a partial function of the other arguments. That is, $Osem(e, v, T) \Rightarrow Osem(e, v', T) \Rightarrow v = v'$.

The above theorem was a design goal for the definition. It also happens that $Osem$ defines type T as a partial function of the arguments other than T and v , but we might reasonably choose to extend the language to one that is "polymorphic" with respect to types. We might introduce elements with multiple types, such as a nil element if we used list types, or a constant that for each function type stands for the identity function on that type. Or, we might introduce types with shared values, such as $\mathbb{N}$ and $\mathbb{Z}$. All of these would be reasonable extensions if we were to include more of the discrete mathematics of [GS93b] in our metalinguistic framework.

(3.11) **Theorem.** Textual substitution is semantically equivalent to updating environments. That is, $Osem(\sigma, V, G^x_E, v, T)$ iff $Osem(\sigma^x_{T'}, V^x_d, G, v, T)$ and $Osem(\sigma, V, E, d, T')$ for some d, T'.

(3.12) **Theorem.** Only free variables affect the value of a term: for any term E, if two models agree on all free variables in E and E has a value in one of the models, then E has the same value in the other model.

(3.13) **Theorem.** Change of bound variables preserves value: for any object expression G, if G has a value in a model, then G' has the same value, where G' is G under a renaming of bound variables.

3.4 Object-Level Theoremhood

Our data language is directly about the theoremhood of expressions in an object language such as the one we defined. As part of the subject matter of the data language, we develop an appropriate notion of object-level theoremhood by defining a class of *object theorems* with respect to *assumptions* α. That is, we define a derivability relation

$$(\epsilon)\ \alpha \vdash P$$

where conclusion P is an object expression, α is a list of object expressions, and ϵ is a *type expression assignment* function (which we will soon describe precisely). In this section, we present a little language of type expressions, define object-level derivability, and discuss its relation to the data language.

3.4.1 Type expressions

We formulate object theoremhood with respect to *type expression* assignments. Type expressions are syntactic elements that we interpret as types; for a domain D, type expressions refer to types in $K(D)$.

Formally, the class OT of object type expressions consists of the pairs $(n)r$, where $n \in \mathbb{N}$ and r is one of $\{bool, ind\}$ (standing for the ground types of booleans and individuals). The semantics for these simple expressions is an interpretation into $K(D)$:

- $[(n)bool]_D = D(n) \rightarrow \mathbb{B}$
- $[(n)ind]_D = D(n) \rightarrow D$

Note that we interpret type expressions as types only in the context of a domain D, but we commonly elide the understood subscript D. In addition, for simplicity, we write $(n)r$ as r in cases where n is 0.

A *type expression assignment* ϵ is a function in $OV \rightarrow OT$. For type expression assignment ϵ, we let $[\epsilon]_D$ be the corresponding type assignment, eliding D when understood. That is, for object variable x, $[\epsilon].x$ is $[\epsilon.x]$ under the above semantics for type expressions. Thus, for simplicity's sake, we may establish properties in terms of either type expression assignments or typing environments, but we also consider them established in terms of the other through this correspondence.

We define the assignment of type expressions to object expressions under ϵ as the strongest relation $(\epsilon)\ A :: T$ that satisfies the following clauses. (We elide the type expression assignment when it is just ϵ.)

1. $\forall x{:}OV.\ x :: \epsilon(x)$
2. $\forall r{:}\{bool, ind\}.\ Ap_O(A; B) :: (n)r$ if $A :: (n+1)r$ and $B :: ind$
3. $\mathbf{f}_O :: bool$
4. $A \Rightarrow_O B :: bool$ if $A :: bool$ and $B :: bool$
5. $(\forall_O x\ |{:}\ A) :: bool$ if $(\epsilon^x_{ind})\ A :: bool$
6. $\forall r{:}\{bool, ind\}.\ A =_O B :: bool$ if $A :: r$ and $B :: r$

This definition follows the form of the clauses of the object language semantics. For notational convenience, we also extend type expression assignment to multiple expressions: $(\epsilon)\ \alpha :: T$ is defined as $\forall A \in \alpha.\ (\epsilon)\ A :: T$.

3.4.2 Definition of object-level theoremhood

The purpose of theoremhood (derivability) is to pick out a usefully large but effectively recognizable subclass of valid expressions, doing so without referring to the semantics. We express this in a semantic constraint on derivability (which we will not prove here)

$$\forall \epsilon{:}OV \rightarrow OT.((\epsilon)\ \alpha \vdash P) \ \Rightarrow\ valid(\epsilon, \alpha\ \Rightarrow\ P)$$

where $valid(\epsilon, Q)$ iff for any domain D and valuation V such that $\langle D, [\epsilon], V\rangle$ is a model, $Osem([\epsilon], V, Q, \mathrm{True}, \mathbb{B})$. We also use $\alpha\ \Rightarrow\ P$ as iteration of implication over antecedent lists, e.g., $a1, a2, a3 \Rightarrow P$ is $a1 \Rightarrow a2 \Rightarrow a3 \Rightarrow P$. Since $(\epsilon)\ \alpha \vdash P$ depends on the values of ϵ only on variables that occur free in α or P, the fact that the domain of ϵ is infinite does not prevent effective recognizability of theoremhood.

There is a noteworthy similarity between our motivations for choosing a particular definition of the object language and a particular definition of object-level derivability. Recall that, in many ways, the details of our particular choice of object language are of little consequence to the data language. Similarly, although a sensible set of object theorems is a necessary foundation for the metatheorems established by calculational logic, the data language and users of its inference methods remain generally insulated from any particularities of the definition of the object theorem set. The metalogical approach of [GS93b] emphasizes the data language forms of inference about object theoremhood and de-emphasizes the details about how the class of object theorems is defined. Therefore, using (and justifying) calculational logic does not require any particular definition of an object theorem set.

Nonetheless, to have a concrete example to refer to when discussing object-level derivability, we present a definition of object theoremhood; it is not the only possible adequate definition. Our set of object language theorems satisfies the above semantic constraint (we omit the proof) and contains the theorems of a traditional predicate logic. We adopt a few conventions for notational simplicity: when we elide the type expression assignment, as in $\alpha \vdash P$, it is ϵ; $P[x := A]$ denotes capture-avoiding textual substitution; P `cbv` Q means that P and Q are related by mere change of bound variables.

(3.14) **Definition.** We define *object theoremhood* $(\epsilon)\ \alpha \vdash P$, for $\epsilon{:}OV \rightarrow OT$, $\alpha{:}OE\,\mathrm{List}$, $P{:}OE$, as the strongest relation satisfying the following clauses:

1. $\alpha \vdash P$ if $P \in \alpha$ and $\alpha :: \mathit{bool}$
2. $\alpha \cup \{A\} \vdash P$ if $\alpha \vdash P$ and $A :: \mathit{bool}$
3. $\alpha \vdash Q$ if $\alpha \vdash P \Rightarrow_O Q$ and $\alpha \vdash P$
4. $\alpha \vdash P \Rightarrow_O Q$ if $\alpha \cup \{P\} \vdash Q$
5. $\alpha \vdash P$ if $\alpha \vdash P'$ and P `cbv` P'
6. $\alpha \vdash P[x := A]$ if $\alpha \vdash (\forall_O x.P)$ and $A :: \mathit{ind}$
7. $\alpha \vdash P$ if $\alpha \vdash (\forall_O x.P)$ and $\neg(x \text{ free in } P)$
8. $\alpha \vdash (\forall_O x.P)$ if $(\epsilon^x_{ind})\ \alpha \vdash P$ and $\neg(x \text{ free in } \alpha)$
9. $\alpha \vdash P$ if $\alpha \vdash \mathbf{f}_O$ and $P :: \mathit{bool}$
10. $\forall r{:}\{\mathit{bool}, \mathit{ind}\}.\ \alpha \vdash P[x := B]$ if $\alpha \vdash P[x := A]$ and $\alpha \vdash A =_O B$ and $A, B :: r$ and $(\epsilon^x_r)\ P :: \mathit{bool}$
11. $\forall r{:}\{\mathit{bool}, \mathit{ind}\}.\ \alpha \vdash A =_O A$ if $A :: r$ and $\alpha :: \mathit{bool}$
12. $\alpha \vdash A =_O B$ if $\alpha \vdash A \equiv_O B$

Most of the clauses correspond to expected elimination and introduction forms for operators in a natural deduction style for sequents: clauses 3 and 4 are elimination and introduction for implication; clauses 6 and 8 are elimination and introduction for quantification over individuals; clause 9 is false-elimination (there is no false-introduction); clauses 10 and 11 are elimination and introduction for equality, with both boolean and individual expressions permitted as the equands.

We comment briefly on the other clauses:

- Clauses 1 and 2 embody generic facts about assumptions.
- Clause 5 admits change of bound variables as an inference. While typically derivable, it is not worth the trouble to derive. Combinations of clauses 3, 4, and 5 permit change of bound variables on hypotheses.
- Clause 7 entails the existence of individuals (elements of the domain), making $\exists_O x.\mathbf{t}_O$ derivable even when $\epsilon{:}OV \to OT$ assigns *bool* to all variables; all the other clauses are valid even for the empty domain. In fact, clause 7 would be a special case of clause 6 if only there were an individual-type expression for every ϵ. Indeed, in formulations of predicate calculus in which each variable has a syntactically fixed type (equivalent to our fixing ϵ for our entire formulation), there are always individual-type variables. In our formulation, however, any variable may be assigned any type, and there are no individual-type constants.
- Clause 12 has many effects. It makes possible the proof of non-trivial equalities on propositional arguments. Indeed, it establishes material equivalence as the equality for propositional values, thus prohibiting multiple "true" or "false" values. Also, because of the way that we defined $A \equiv_O B$, the standard interpretation of "bool" is forced under our definitions of theoremhood. If $A \equiv_O B$ were built in as a primitive operator, then all these clauses would be intuitionistically valid. Because $A \equiv_O B$ is $\neg_O((A \Rightarrow_O B) \Rightarrow_O \neg_O(B \Rightarrow_O A))$, however, we can derive the characteristic theorem of classical logic, $\neg_O\neg_O P \Rightarrow_O P$ (let A be P and B be $\mathbf{t}_O$).

3.4.3 Notes on object-level derivability

If we were directly generating object-level proofs in accord with our definition of derivability, the role of ϵ might cause concern for at least two reasons: it is an infinite function; and it specifies a type for every variable, failing to exploit the polymorphism of equality that we troubled to admit. We do not, however, intend to build a class of object proofs. Instead, we use derivability by making claims and arguments about it in a metalanguage, and we interpret [GS93b] as also doing so, in a very restricted formalized metalanguage.

As mentioned above, the fact that ϵ is infinite does not prevent effective recognizability of theoremhood. Let us now consider the polymorphism concern. Here is a simple, contrived, book-style assertion:

Assuming $Y =_O X$ and $V =_O U$, $X =_O Y \wedge_O U =_O V$.

It is polymorphic, independently, in both the type of X and the type of U. We interpret it as:

$\forall r1, r2 : \{$ bool , ind $\}$. $(\epsilon)\ \alpha \vdash (X =_O Y \wedge_O U =_O V)$
if $Y =_O X \in \alpha$ and $V =_O U \in \alpha$ and $(\epsilon)\ X, Y :: r1$ and $(\epsilon)\ U, V ::$
$r2$

where we implicitly quantify over the obvious types. The polymorphism of object language variables is expressed using variables of the metalanguage to range over type expressions.

3.5 Quantification and Iteration

One of the advances in [GS93b] is its treatment of quantification, which goes beyond the typical structure reflected in our object syntax. Recall that the object language quantifiers take only one variable. In the data language, however, we want to include iterations of these quantifications over lists of variables. Before beginning our formulation of the data language, we informally introduce notations for these iterated object language quantifiers.

We define iterated quantification by induction over lists of variables, as follows.

$$(\forall_O \bar{x} \mid R : P) = \begin{cases} R \Rightarrow_O P & \text{if } \bar{x} \text{ is the empty list} \\ (\forall_O y.\ (\forall_O \bar{z} \mid R : P)) & \text{if } \bar{x} = y.\bar{z} \end{cases}$$

$$(\exists_O \bar{x} \mid R : P) = \begin{cases} R \wedge_O P & \text{if } \bar{x} \text{ is the empty list} \\ (\exists_O y.\ (\exists_O \bar{z} \mid R : P)) & \text{if } \bar{x} = y.\bar{z} \end{cases}$$

Similar to the practice in [GS93b], when R is $\mathbf{t}_O$, we may omit it from the notation; for example, $(\exists_O \bar{x} \mid : P)$ is an abbreviation for $(\exists_O \bar{x} \mid \mathbf{t}_O : P)$.

The common form of quantifiers is abstracted to a function of four variables, $(\star^M xs \mid R : B)$, where variable $\mathtt{M}$:$\{\wedge^M, \vee^M\}$ is a kind of quantifier index. By convention, $(\wedge^M xs \mid R : B)$ and $(\vee^M xs \mid R : B)$ are written as $(\forall_O xs \mid R : B)$ and $(\exists_O xs \mid R : B)$, respectively. They are also defined inductively on variable list xs.

The motivating idea is that $(\star^M v \mid R : B)$ is the iteration of a binary operator $x \ \star^M y$ on the values that B takes for v satisfying R. For a paradigm, consider the interpretation of $(\Sigma v \mid R : B)$ as $(+v \mid R : B)$, the iteration of $x + y$. For instance, assume predicate $R.x$ holds only on individuals i_0 and i_1. Then, as expected, $(\Sigma v \mid R.v : f.v) = f(i_0) + f(i_1)$. Correspondingly, in the treatment of quantifiers in [GS93b], $(\exists_O v \mid R.v : f.v) = f(i_0) \vee_O f(i_1)$, and similarly for $\forall_O$.

Part of extending the formalization beyond predicate logic would be to give a semantics to $(\star^M xs \mid R : B)$ for various other operators $\mathtt{M}$. (Indeed, summation might be part of a sensible extension.) We require that $\mathtt{M}$ be commutative and associative and that B assumes values other than the identity of $\mathtt{M}$ only finitely often on R. In case $\mathtt{M}$ does not have an identity, we further require that R be non-empty; the development in [GS93b] states that a quantifier expression $(\star^M xs \mid R : B)$ with an empty range gets the value $\mathtt{id(M)}$, the identity of $\mathtt{M}$.

When the related binary operator and identity are defined with respect to the quantifier, useful properties can be stated abstractly. For $\mathtt{M} = \wedge^M$, $P \ \star^M Q$ is $P \wedge_O Q$ and $\mathtt{id(M)}$ is $\mathbf{t}_O$. Similarly, for $\mathtt{M} = \vee^M$, $P \ \star^M Q$ is $P \vee_O Q$ and $\mathtt{id(M)}$ is $\mathbf{f}_O$. Using this informal metalanguage, we may state uniformities such as:

$$((\star^M xs \mid R : P) \ \star^M (\star^M xs \mid R : Q)) =_O (\star^M xs \mid R : P \ \star^M Q).$$

The calculational style of [GS93b] is designed to exploit such properties.

3.6 The Data Language

We use the term *data language* for the expressions of [GS93b] that are directly manipulated by users. In a more conventional development, the data language would simply be some notation for proofs whose constituent propositions would be about numbers, lists, sets, functions, or some other basic domain of interest. We instead propose that the data language of the book should be understood as a restricted, formalized metalanguage about theoremhood of an object language. In our opinion, the arguments of [GS93b] (at least after the introduction of quantification) are most easily read as rather straightforward arguments about the structure of expressions and derivability. Indeed, our data language does not directly refer to the object language semantics, only to a class of object theorems. This reflects a critical distinction: the job of argument about object-level theoremhood belongs to the data language; the job of addressing mathematical domains of principal interest (lists, numbers, etc.) belongs to the object language.

This, the data level, is the level at which several atypical features of the calculational logic of [GS93b] are explained; the object language and object-level derivability were only precursors to the data language, not domains in which the details of the *user level* of calculational logic could be discussed. In this section, we formally present the syntax and semantics of the data language. We here restrict ourselves to calculational predicate logic, but we anticipated eventual extensions to a broader range of the topics in the discrete mathematics text [GS93b], which influenced some of our design decisions.

3.6.1 Data language syntax

We intend the data language to adequately represent the language actually used in the book to discuss object theoremhood. Therefore, its syntax is significantly more involved than that of our simple, standard object language. Atypical features of calculational logic can be seen in the generalized treatment of quantification and the relations involving object language syntax.

(3.15) **Definition.** DT is a class of type expressions used in data language quantifiers. It comprises the constants OV, OE, OT, $\mathbb{N}$, $\{\ \mathrm{bool}\ ,\ \mathrm{ind}\ \}$, and $\{\wedge, \vee\}$, and it is closed under the operators T List and $T \times T'$. The overloaded notation —OV, OE, etc. denote both types and type expressions— should cause no confusion in our explanation. Context can distinguish types from type expressions when relevant.

(3.16) **Definition.** DV is a denumerable class of identifiers used as *data-language variables*. The data language has no function or predicate variables, only variables over (several different types of) individuals. DE is the class of *data-language expressions*, comprising DV and all instances of the following (where $a, b, c, \star, \epsilon, \alpha \in DE$, $x \in DV$, and $t \in DT$):

- Various standard logical operations (including identity): $\mathit{if}\ a\ \mathit{then}\ b$; $a\ \&\ b$; $a\ \mathit{or}\ b$; $\mathit{not}\ a$; a is b; for $x : t.\ a$; $a\ \mathit{iff}\ b$.
- Constants and operations for lists and pairs: $[\,]$; $a \cdot b$; $\langle a, b\rangle$.
- A predicate denoting list membership: $a \in b$.
- Constants representing object variables: abc (for $abc_O \in OV$).
- Various constants and operations for constructing object expressions: $a(b)$; $\mathbf{t}$; $\mathbf{f}$; $a \Rightarrow b$; $a \Leftarrow b$; $a = b$; $\neg a$; $a \equiv b$; $a \wedge b$; $a \vee b$; $(\star a \mid b : c)$; $a \star b$; $\mathtt{id}(\star)$; $a[b := c]$.
- Constants for the quantifier indices $\{\wedge^M, \vee^M\}$: $\wedge, \vee$. (We may write $\forall$ for $\wedge$ and $\exists$ for $\vee$.)

- Constants and an operation for denoting object type expressions: ⌜bool⌝ , ⌜ind⌝ , $(a)b$.
- A constant for each natural number: e.g., 99.
- Operations for various relations involving object syntax:
  - Representing object theoremhood: $(\epsilon)\ \alpha \vdash a$.
  - Representing typing of object expressions: $(\epsilon)\ a :: b$.
  - Updating of environments: ϵ^a_b . Note that we overload this notation, using it for updating in both the informal metalanguage and the formalized data language.
  - Representing assumptions in statements about object theoremhood: assuming $(\in \alpha)\ a,\ b$.
  - Representing that an object variable is free in an object expression: a free in b .

3.6.2 Semantics of data language types

Before we use the type expressions of DT in our data language semantics, we present a few preparatory notes.

(3.17) **Definition.** The semantics of the type expressions in DT is straightforward. For $t \in DT$, $[t]$ is defined as follows:

$$
\begin{array}{rcl}
[OT] & = & OT \\
[OE] & = & OE \\
[OV] & = & OV \\
[\{\ \text{bool}\ ,\ \text{ind}\ \}] & = & \{bool, ind\} \\
[\{\wedge, \vee\}] & = & \{\wedge^M, \vee^M\} \\
[\mathbb{N}] & = & \mathbb{N} \\
\forall t{:}DT.([t\ \text{List}] & = & [t]\ \text{List}) \\
\forall t1, t2{:}DT.([t1 \times t2] & = & [t1] \times [t2])
\end{array}
$$

We remind readers about our overloaded notation, previously discussed in definition 3.15. We do not expect it to cause confusion.

(3.18) **Definition.** We also employ a subtype relation on DT. For $t1, t2 \in DT$, we define $t1 \subseteq t2$ as follows:

$$OV \subseteq OE$$
$$\forall t : DT.\ t \subseteq t$$
$$\forall t1, t2 : DT.(t1 \text{ List}) \subseteq (t2 \text{ List}) \text{ if } t1 \subseteq t2$$
$$\forall t1, t2, t3, t4 : DT.(t1 \times t3) \subseteq (t2 \times t4) \text{ if } t1 \subseteq t2 \text{ and } t3 \subseteq t4$$

Naturally, this subtype relation on the notations of DT represents actual subtyping, i.e.,

$$\forall t1, t2 : DT.\ t1 \subseteq t2 \ \Rightarrow\ [t1] \subseteq [t2]\,.$$

3.6.3 Data language semantics

We define the data language semantics using essentially the same methods as for the object language. There are differences, such as multiple types of individuals and subtyping, but we could easily adopt these devices in the object language, too, if we were to extend it.

In the definitions that follow, `Type` refers to a collection of collections of values.

(3.19) **Definition.** A *model* of the data language is a pair $\langle \gamma, V \rangle$ where

- $\gamma : DV \rightarrow \texttt{Type}$ is a *typing environment* function.
- $V : \Pi x{:}DV.\ \gamma(x)$ is a *valuation*, a function that assigns to each variable x a value in the type $\gamma(x)$.

(3.20) **Definition.** The data language semantics is given as a recursive definition of $Dsem(\gamma, V, A, v, T)$ for $\gamma \in DV \rightarrow \texttt{Type}$, $V \in (\Pi x{:}DV.\ \gamma(x))$, $A \in DE$, $T \in \texttt{Type}$, $v \in T$. We intend $Dsem(\gamma, V, A, v, T)$ to hold exactly when expression A has type T and value $v \in T$ under model $\langle \gamma, V \rangle$. (As with the object language semantics, we elide the arguments γ and V where it is obvious what is meant.) It is defined according to the following clauses, where variables range over data expressions unless otherwise specified:

1. $Dsem(A, a, [t2])$ if $Dsem(A, a, [t1])$ and $t1 \subseteq t2$.
2. $\forall x{:}DV.\ Dsem(x, V(x), \gamma(x))$.
3. $Dsem(\mathit{if}\ P\ \mathit{then}\ Q, q \text{ if } p, \mathbb{B})$ if $Dsem(P, p, \mathbb{B})$ and $Dsem(Q, q, \mathbb{B})$, and similarly for the other connectives.

4. $Dsem(A \text{ is } B, a = b, \mathbb{B})$ if $Dsem(A, a, T)$ and $Dsem(B, b, T)$.

5. $\forall g{:}[t] \to \mathbb{B}.\ Dsem((\text{for } x : t.\ a), (\forall v{:}[t].g(v)), \mathbb{B})$
   if $\forall v{:}[t].\ Dsem(\gamma^x_{[t]}, V^x_v, a, g(v), \mathbb{B})$.

6. $\forall T{:}\texttt{Type}.\ Dsem([\,], \text{nil}, T \text{ List})$; note the type polymorphism.

7. $Dsem(A \cdot B, \text{the prepending of } a \text{ to } b, T \text{ List})$
   if $Dsem(A, a, T)$ and $Dsem(B, b, T \text{ List})$.

8. $Dsem(\langle A, B\rangle, \langle a, b\rangle, T1 \times T2)$ if $Dsem(A, a, T1)$ and $Dsem(B, b, T2)$.

9. $Dsem(abc, abc_O, OV)$; similarly for other object variable literals.

10. $Dsem(\mathbf{t}, \mathbf{t}_O, OE)$ and $Dsem(\mathbf{f}, \mathbf{f}_O, OE)$.

11. $Dsem(A \equiv B, a \equiv_O b, OE)$ if $Dsem(A, a, OE)$ and $Dsem(B, b, OE)$;
    similarly for other OE-constructors.

12. $Dsem((\star X \mid R : P), (\star^M x \mid r : p), OE)$
    if $Dsem(X, x, OV \text{ List})$ and $Dsem(R, r, OE)$
    and $Dsem(P, p, OE)$ and $Dsem(\star, \texttt{M}, \{\wedge^M, \vee^M\})$.

13. $Dsem(A[X := B], a^x_b, OE)$
    if $Dsem(A, a, OE)$ and $Dsem(B, b, OE)$ and $Dsem(X, x, OV)$,
    where a^x_b is a capture-avoiding substitution function on object expressions. (Fully specifying a textual substitution function for our purposes would be tedious and unnecessary.)

14. And similarly for other operators that denote object expressions, quantifier indices, object type expressions, numeric constants, etc.

Before continuing with the semantics for the remaining elements of data language syntax —the operators that denote relations involving object syntax— we introduce an auxiliary device. To simplify the data language, we have avoided introducing function types. So, in our data language, we will use lists of pairs to refer indirectly to assignments of object type expressions to object variables. We introduce here a simple "ϵ to $OV \to OT$" notation that maps a list ϵ to a function; unmatched variables are mapped to $bool \in OT$ so we may consider the resulting function as total.

- $\forall \epsilon{:}(OV \times OT) \text{ List}.(\epsilon \text{ to } OV \to OT) \in OV \to OT$.

- $\forall x{:}OV, t{:}OT, \epsilon{:}(OV \times OT) \text{ List}.((\langle x, t\rangle, \epsilon) \text{ to } OV \to OT)(x) = t$.

- $\forall x, y{:}OV, t{:}OT, \epsilon{:}(OV \times OT) \text{ List}.$
  $((\langle x, t\rangle, \epsilon) \text{ to } OV \to OT)(y) = (\epsilon \text{ to } OV \to OT)(y) \text{ if } \neg(x = y)$.

- $\forall x{:}OV.([\,]\text{ to } OV \to OT)(x) = \mathit{bool}$.

We now proceed with the remaining clauses of the data language semantics.

15. $\mathit{Dsem}((\epsilon)\ \alpha \vdash P, (\mathit{vts}\text{ to } OV \to OT)\ \mathit{as} \vdash p, \mathbb{B})$
    if $\mathit{Dsem}(\epsilon, \mathit{vts}, (OV \times OT)\text{ List})$ and $\mathit{Dsem}(\alpha, \mathit{as}, OE\text{ List})$
    and $\mathit{Dsem}(P, p, OE)$.

16. $\mathit{Dsem}((\epsilon)\ A :: B, (\mathit{vts}\text{ to } OV \to OT)\ a :: T, \mathbb{B})$
    if $\mathit{Dsem}(\epsilon, \mathit{vts}, (OV \times OT)\text{ List})$ and $\mathit{Dsem}(A, a, OE\text{ List})$
    and $\mathit{Dsem}(B, T, OT)$.

17. $\mathit{Dsem}(\epsilon^X_B, (\langle x, T\rangle, \mathit{vts}), OV \times OT\text{ List})$
    if $\mathit{Dsem}(\epsilon, \mathit{vts}, (OV \times OT)\text{ List})$ and $\mathit{Dsem}(X, x, OV)$
    and $\mathit{Dsem}(B, T, OT)$.

18. $\mathit{Dsem}((\text{assuming } (\in \alpha)\ P,\ Q), p \in \mathit{as}\ \Rightarrow\ q, \mathbb{B})$
    if $\mathit{Dsem}(\alpha, \mathit{as}, OE\text{ List})$ and $\mathit{Dsem}(P, p, OE)$ and $\mathit{Dsem}(Q, q, \mathbb{B})$.

19. $\mathit{Dsem}(X\text{ free in } A, \text{variable } x \text{ occurs free in some member of } a, \mathbb{B})$
    if $\mathit{Dsem}(X, x, OV)$ and $\mathit{Dsem}(A, a, OE\text{ List})$.

This is a conventional first-order semantics, and things are generally as one would expect. For instance, clauses 1 and 2 state that subtypes and data language models behave in the expected ways, clause 6 gives the expected (polymorphic) meaning to the empty list, etc. We omitted many clauses that would be included in an exhaustive presentation; these omitted clauses, however, do not require any additional semantic machinery.

Clauses 3 and 4 are examples of our treatment of logical operators typically written in natural language, like "if ... then" and "is," metalinguistic operators used to discuss mathematics. We claim that the calculational logic of [GS93b] can be read as a formalized metalanguage, however, with these operators as part of the data language. There is also a temptation to confuse OE-constructors with data-level logical operations. For instance, the OE-constructor $A \ \Rightarrow\ B$ is different from the data-level logical operation *if* P *then* Q; the former is an OE-valued operation on expressions of type OE and the latter is an operation on $\mathbb{B}$. Similar distinctions apply for the OE-valued $A \wedge B$ and the $\mathbb{B}$-valued $P\ \&\ Q$, etc.

Our syntax and semantics omitted a few expressions that appear in [GS93b] that are easily definable in terms of the language we presented. For instance, (X free in A) is extended to $\mathit{occurs}(V{:}OV\text{ List}, E{:}OE\text{ List})$, a data-level predicate that holds when some V' in V is such that (V' free in E). It is typically negated as $\mathit{not\ occurs}(V, E)$ to indicate that none of the variables referred to by elements of V occur free in any expression referred to by an element of

E. Updating of type expression environments is also implicitly extended to lists: for $X{:}OV$ List and $r{:}OT$, ϵ_r^X denotes a copy of $\epsilon{:}(OV{\times}\, OT)$ List with all variables on X updated to type expression r. It can be easily defined in terms of the single-variable version. We believe it is clear how to incorporate such simple extensions into our formal framework, and we consider them to be in our data language for the purpose of examples in section 3.7.

Relatedly, we have not been explicit about certain conventions in [GS93b] that are merely a matter of display. For instance, clause 18 of the semantics indicates that "assuming" is essentially implication, as expected. In practice, it is used only in the context of object theoremhood judgments and the "$(\in \alpha)$" notation is elided; it specifies that a particular expression is included among the assumptions, making our notation look like that of the book [GS93b]. We can use the more general "*if* ... *then*" instead of the "assuming" display form, if it improves readability. As another example, a three element list might be written in the standard notation "a, b, c," and a singleton list is written simply as its element. All essential details of the formation of lists are present in that representation and expressible in the data language. Similarly, when the types of f and x are clear from context, we may write $f.x$ instead of $f(x)$ for the OE-valued constructor of object-language function applications. Notational conversions of this sort occur only for easily understood standard notation, and they should not pose difficulties for readers. Unless otherwise explained, standard notations have their expected meanings.

3.7 Using the Data Language

Our goal in this chapter is to demonstrate how calculational predicate logic (as presented in chapters 3, 8, and 9 of [GS93b]) could be easily read as a restricted, formalized metalanguage. We have formalized this data language; the outline in this chapter is the basis for the formalization in chapter 5. We do not here attempt to formalize the actual proof structure used in [GS93b]; we expect that the conventional semantics of our data language makes it clear that one can do so.

In this section, we provide concrete examples of how material from the text can be expressed using our data language; the names and numbers labeling our examples are taken directly from the text [GS93b]. Our motivation for providing these examples is two-fold. Foremost, we demonstrate that our data language is adequate for calculational logic by using it to express various theorems and inference rules that were chosen to illustrate the full range of the calculational logic language; in particular, we formalize the component propositions of the proof of Theorem Change Of Dummy, so readers can compare our metalinguistic treatment with the text from [GS93b]. Equally importantly, we provide examples of the propositions and inferences used in calculational logic. By showing detailed representations of the propositions, readers can see that inferences from one to another could be formalized in conventional ways.

3.7.1 Theorems

We present data language expressions for a few objects classified as theorems in [GS93b]. In [GS93b], typically only the notation that looks like standard predicate logic is presented as the theorem; for instance, the first theorem below, **Identity of** $\vee$, is simply given as $P \vee \mathbf{f} \equiv P$. Here, we give full explanations of the meanings of these common (in [GS93b]) theorems, brief examples to illustrate the scope and adequacy of our formalized data language. In particular, we include the theorems cited in the proof of Theorem Change of Dummy (Figure 3.1) among our examples.

- (3.30) **Identity of** $\vee$**.** *For* $\epsilon : (OV \times OT)$ List, $\alpha{:}OE$ List, $P{:}OE$.
  if $(\epsilon)\ P ::$ bool & $(\epsilon)\ \alpha ::$ bool *then* $(\epsilon)\ \alpha \vdash P \vee \mathbf{f} \equiv P$.

- (3.35) **Golden Rule.** *For* $\epsilon{:}(OV \times OT)$ List, $\alpha{:}OE$ List, $P, Q{:}OE$.
  if $(\epsilon)\ P, Q ::$ bool & $(\epsilon)\ \alpha ::$ bool
  then $(\epsilon)\ \alpha \vdash P \wedge Q \equiv P \equiv Q \equiv P \vee Q$.
  (Recall that the boolean operator $\equiv$ is associative.)

- (3.84a) **Substitution.**
  For $\epsilon{:}(OV \times OT)$ List, $\alpha{:}OE$ List, $r{:}\{$ bool , ind $\}$, $z{:}OV, e, f, P{:}OE$.
  if $(\epsilon)\ \alpha ::$ bool & $(\epsilon)\ e, f :: r$ & $(\epsilon^z_r)\ P ::$ bool
  then $(\epsilon)\ \alpha \vdash (e = f) \wedge P[z := e] \equiv (e = f) \wedge P[z := f]$.

- (8.14) **One-point rule.**
  For $\epsilon{:}(OV \times OT)$ List, $\alpha{:}OE$ List, $\star{:}\{\wedge, \vee\}$, $x{:}OV, E, P{:}OE$.
  if not occurs(x, E) & $(\epsilon)\ \alpha ::$ bool & $(\epsilon)\ E ::$ ind & $(\epsilon^x_{\text{ind}})\ P ::$ bool
  then $(\epsilon)\ \alpha \vdash (\star x \mid x = E : P) = P[x := E]$.

- (8.20) **Nesting.**
  For $\epsilon{:}(OV \times OT)$ List, $\alpha{:}OE$ List, $\star{:}\{\wedge, \vee\}$, $r{:}\{$ bool , ind $\}$,
  $x, y{:}OV, P, Q, R{:}OE$.
  if not occurs(y, R) & $(\epsilon)\ \alpha ::$ bool & $(\epsilon^{x,y}_{\text{ind}})\ P, Q, R ::$ bool
  then $(\epsilon)\ \alpha \vdash (\star x, y \mid R \wedge Q : P) = (\star x \mid R : (\star y \mid Q : P))$.

- (9.2) **Trading.** *For* $\epsilon{:}(OV \times OT)$ List, $\alpha{:}OE$ List, $x{:}OV, P, R{:}OE$.
  if $(\epsilon)\ \alpha ::$ bool & $(\epsilon^x_{\text{ind}})\ P, R ::$ bool
  then $(\epsilon)\ \alpha \vdash (\forall x \mid R : P) \equiv (\forall x \mid : R \Rightarrow P)$.

3.7.2 Inference rules

The statements identified in [GS93b] as calculational logic inference rules express a relationship between object-level theoremhood judgments; typically, they have the form "Provided certain syntactic constraints hold on P and Q, if P is an object theorem, then Q is an object theorem," where P, Q refer to object expressions. Statements like these are readily expressed using the data language. Here, we present data-language re-statements of three calculational logic inference rules, a "Leibniz" rule that permits textual substitution into the body of quantifiers, the "Transitivity" rule of equality-derivability employed in the proof of Theorem Change of Dummy, and an $\equiv$-elimination rule called "Equanimity".

- **Leibniz.**
  For $\epsilon{:}(OV \times OT)$ List, $\alpha{:}OE$ List, $\star{:}\{\wedge, \vee\}$,
  $\quad Z{:}OV, X{:}OV$ List, $A, B, P, R{:}OE, r{:}\{$ bool , ind $\}$.
  if not occurs(X, α) &
  $\quad (\epsilon^{X}_{\text{ind}})\ \alpha\ \vdash\ (R\ \Rightarrow\ A = B)$ &
  $\quad (\epsilon^{X}_{\text{ind}})\ A, B :: r$ &
  $\quad (\epsilon^{X,Z}_{\text{ind},r})\ P ::$ bool
  then $(\epsilon)\ \alpha\ \vdash\ (\star X \mid R : P[Z := A]) = (\star X \mid R : P[Z := B])$

- **Transitivity.**
  For $\epsilon{:}(OV \times OT)$ List, $\alpha{:}OE$ List, $P, Q, R{:}OE$.
  if $(\epsilon)\ \alpha \vdash P = Q$ &
  $\quad (\epsilon)\ \alpha \vdash Q = R$
  then $(\epsilon)\ \alpha \vdash P = R$

- **Equanimity.**
  For $\epsilon{:}(OV \times OT)$ List, $\alpha{:}OE$ List, $P, Q{:}OE$.
  if $(\epsilon)\ \alpha \vdash P$ &
  $\quad (\epsilon)\ \alpha \vdash P \equiv Q$
  then $(\epsilon)\ \alpha \vdash Q$

3.7.3 Proof content

One of the critical requirements for the data language is that it be adequate for expressing the component propositions of calculational proofs in [GS93b]. Here, as a demonstration of adequacy, we present the central component propositions of the proof of Theorem Change of Dummy (see Figure 3.1). Other proofs in [GS93b] can be read similarly.

For the first proposition, we are explicit in universally closing the proposition and listing all the components of the antecedent, including the "assuming" construct. For other propositions, we omit these details for a more concise presentation, but the statements are to be considered closed under the appropriate universal quantification and guarded by the same antecedent. We can judge data language adequacy in part by their similarities to the notation in [GS93b] also used in Figure 3.1. Note that the similarities would be even greater if we systematically abbreviated $(\epsilon)\ \alpha \vdash P$ to simply P.

This subsection (3.7.3) also provides the most direct support for our claim that, although we do not formalize a proof structure for calculational logic in this chapter, it could be done in a straightforward way. Using inference rules and theorems like those previously presented as data-language expressions, there are no unusual elements in the inferences from one proposition to the next. A detailed account of these inferences, however, is unnecessary to support the goals and claims of this chapter.

In the following, note that $\star$ is a variable of the data language.

1. *For* $\epsilon{:}(OV \times OT)$ List, $\alpha{:}OE$ List, $\star{:}\{\wedge, \vee\}, P, R{:}OE, f, f^{-1}, x, y{:}OV.$
   assuming $(\in \alpha)\ (\star x, y \mid \mathbf{t} : x = f.y \equiv y = f^{-1}.x)$,
   if $(\epsilon^{x}_{\text{ind}})\ R, P ::$ bool & $(\epsilon)\ \alpha ::$ bool & $(\epsilon^{x,y}_{\text{ind}})\ f, f^{-1} :: (1)$ind &
   not occurs$(y, [R, P])$ & *not* $(x$ is $y)$ & *not occurs*$([x, y], \alpha)$
   then $\epsilon\ \alpha \vdash (\star y \mid R[x := f.y] : P[x := f.y]) =$
   $(\star y \mid R[x := f.y] : (\star x \mid x = f.y : P))$

2. $\epsilon\ \alpha \vdash (\star y \mid R[x := f.y] : (\star x \mid x = f.y : P)) =$
   $(\star x, y \mid R[x := f.y] \wedge x = f.y : P)$

3. $\epsilon\ \alpha \vdash (\star x, y \mid R[x := f.y] \wedge x = f.y : P) =$
   $(\star x, y \mid R[x := x] \wedge x = f.y : P)$

4. $(\epsilon)\ \alpha \vdash (\star x, y \mid R[x := x] \wedge x = f.y : P) = (\star x \mid R : (\star y \mid x = f.y : P))$

5. $(\epsilon)\ \alpha \vdash (\star x \mid R : (\star y \mid x = f.y : P)) = (\star x \mid R : (\star y \mid y = f^{-1}.x : P))$

6. $(\epsilon)\ \alpha \vdash (\star x \mid R : (\star y \mid y = f^{-1}.x : P)) = (\star x \mid R : P[y := f^{-1}.x])$

7. $(\epsilon)\ \alpha \vdash (\star x \mid R : P[y := f^{-1}.x]) = (\star x \mid R : P)$

3.8 Conclusion and Discussion

By formalizing a metalinguistic interpretation of the language of the calculational logic of [GS93b], we have provided:

- A sensible, thorough reading for the language of calculational logic, using only conventional semantic methods;

- Meanings for the propositions manipulated by users of calculational logic, providing evidence that there is nothing unusual about the inferences needed for calculational logic;

- A framework in which one can evaluate the extent to which calculational logic is non-standard.

We conclude the chapter by discussing each of these issues and how we have addressed them. For this section, it will be helpful if readers have read some of the examples of the data language (in section 3.7), but it is not necessary to be thoroughly familiar with that section.

3.8.1 Justifying calculational logic

Using the examples from section 3.7 as support, we believe that the technical content of this chapter is an adequate basis for justifying calculational logic. We explained the language of calculational logic using only standard, well-understood semantic methods. The semantics that we gave resulted in a sensible reading of the predicate logic in [GS93b], and it can be extended to account for more of the discrete mathematics in that text.

There may be ways to formally justify calculational logic that are fundamentally different from ours. (There are certainly other ways to give an object language and a definition of object theoremhood, but these descriptions are not fundamental to our justification.) We simply intended to give one such justification, some foundation to the method.

It was not immediately clear to us that such a reading could be given at all. The combination of elements of object language and metalanguage in the formal logic of [GS93b] made it seem possible that calculational logic was entirely heuristic, a method for constructing arguments that would not withstand the detailed investigation of formalization. This possibility can now be discounted.

3.8.2 Calculational logic inferences

The inferences of calculational logic may seem odd, based on the typical calculational proof structure presented in [GS93b]. We provided evidence to the contrary; in fact, they are very straightforward and can be easily understood. Any confusion about the inferences may have arisen from confusion about the meanings of the propositions involved, a failure to distinguish object-level from meta-level. The inference from proposition P to proposition Q can indeed be difficult to understand without knowing what P and Q really are.

It would be both tiresome and outside the scope of this chapter to fully formalize calculational logic inference. For instance, we would need to formalize all the facts about how to deduce the correctness of a typing claim " $(\epsilon)\ A :: T$ " from other typing claims, and we would need to formalize many facts about object theoremhood to support deductions of one object theoremhood claim $\alpha\ \vdash P$ from other similar claims. Such facts can now be readily grasped, expressed, and applied using standard methods, however, given our clarification of the propositions underlying calculational logic. Indeed, we describe our formalization and implementation of a substantial body of calculational logic inferences in chapter 6 of this dissertation.

The proof structure of calculational logic can also be readily understood, in the context of the fundamental inferences, and we do not discuss it here.

3.8.3 Is calculational logic non-standard?

When interpreted in a traditional framework for first-order predicate logic (with equality and function symbols), the calculational logic in [GS93b] is readily seen to be non-standard in the following ways:

1. Identifiers do not have fixed types.
2. $P = Q$ is a proposition even for propositions P, Q .
3. There is (limited) type polymorphism. For example, $A = A$ is a theorem whether A is an expression of type boolean or individual.
4. Quantifier expressions are treated abstractly in two ways:
   (a) the choice of underlying quantifier ($\forall$ or $\exists$)
   (b) the number of quantified variables (i.e., not necessarily just one variable)
5. Textual substitution $E[V := P]$ appears in theorems.

We interpret calculational logic in a different way: as a metalogic, in which (for instance) the operation $P \vee Q$ refers not to the ordinary disjunction operation on truth values but to a term-constructor, denoting an expression that itself stands for disjunction in a separate object language. Under this interpretation, we account for the first two of the features enumerated above as slightly non-standard choices for the object language. In contrast, the remainder are explained as phenomena of the formalized metalanguage. As this chapter demonstrates, all can be readily formalized using only conventional metalinguistic semantic techniques.

With the clarifications provided by our formalization of calculational logic, we also exposed several elements not apparent in the presentation in [GS93b]. They are listed in our data language syntax among the operations involving object syntax: representations of object theoremhood, typing of object expressions, updating environments, and representing assumptions in object theoremhood judgments. Some of these operations may seem non-standard in the context of conventional first-order logics, but in the context of calculational logic —intended to permit reasoning about the typically metalinguistic operation of textual substitution— they are readily seen as formalizations of common or necessary aspects of inference. Furthermore, we have shown that these operations, too, can be formalized in straightforward ways.

Interestingly, even though an operation representing object-level theoremhood is not explicitly present in formulas designated as theorems in [GS93b], it is explicitly used in statements designated as metatheorems. Readers of [GS93b] may notice that theorems and metatheorems seem to be smoothly integrated in the same logical system. Our interpretation of calculational logic accounts for that by suggesting they are essentially the same: all the theorems in the book are metatheorems, about theoremhood in some other object language. It is therefore unsurprising that these two apparently disparate levels can be effectively integrated; they are not different levels at all.

From the calculational proof structure given in [GS93b], it initially seemed that there could be a considerable need for non-standard semantic methods underlying the unusual appearance of calculational logic. In fact, no unconventional semantic devices are necessary to interpret calculational logic as a metalogic. Perhaps most satisfyingly to supporters of the calculational method, a straightforward metalinguistic formalization of calculational logic renders even its non-standard elements readily explained by standard semantic methods.

Chapter 4

An Overview of Nuprl

The Nuprl proof development system provides a pre-existing platform for developing our model of calculational logic inference. This chapter is an overview of Nuprl, intended both as an introduction for the uninitiated and, for readers familiar with Nuprl, a brief summary of the Nuprl features that are most important for our model of calculational logic inference. For a history of the system and a more complete introduction, see [C+86] and [Jac94].

4.1 Implementing Mathematics in Nuprl

In this section, we introduce some important features of Nuprl that we used in implementing the data language. Our definitions were essentially built from Nuprl's type system and expressed by Nuprl *terms*, so we briefly discuss types and terms. We also introduce Nuprl's system of *display forms*, which preserves the useful distinction between mathematical concepts and notation; for many Nuprl objects, their display forms —which describe how the objects are to be displayed in various contexts— are defined separately from the objects themselves. We exploit display forms in several important ways.

We do not yet consider concepts particular to definitions of inference methods or other meta-level programs in this section; we discuss them in the next section.

4.1.1 Nuprl types

In its standard semantics, Nuprl is based on an intuitionist type theory similar to that of Martin-Löf (see [ML82, All87, ACHA90]). This has a few significant consequences for us as developers. For instance, booleans and propositions are not the same, and we cannot generally do *if-then* branching on propositions; we need to prove that the truth of a proposition is decidable before we can branch on it in an *if-then* context. In addition, readers may recall from chapter 3 that we do not have a function for object-level semantics in the data language. Because calculational logic has a classical semantics and Nuprl does not,[1] such a meaning function could not be implemented directly in our Nuprl-based data language. This does not, however, impede our project, because calculational logic is based around the syntactic property of object-level theoremhood, not the semantic property of object-level truth. Calculational logic is syntactically oriented, and Nuprl is extremely flexible with respect to syntax. Semantic mismatches between the two systems do not affect us.

Indeed, we do not need to consider most of the details of Nuprl's type theory in this introduction. Essentially, we simply defined functions and other mathematical objects in a lambda calculus within a sophisticated type system; that level of understanding should suffice for most of our readers. Nuprl's type theory, however, is much deeper and more broadly applicable than we represent here. See [C$^+$86] for more information.

We now discuss some types and related functions, to provide some necessary background for forthcoming chapters and a feel for how we use Nuprl.

Disjoint union A union operation $+$ is one way to combine types: if $T1$ and $T2$ are types, then we can express the notion that a term t is in one of $T1$ or $T2$ by saying it is in the union of the types, i.e. $t \in T1 + T2$. The type system of Nuprl uses a *disjoint union* to combine types, so given an element of $T1 + T2$, it must be possible to determine which component t is in, $T1$ or $T2$. To accomplish this, Nuprl uses the term constructors `inl` and `inr`; for $t1 \in T1$, `inl(`$t1$`)` is in $T1 + T2$, and for $t2 \in T2$, `inr(`$t2$`)` is in $T1 + T2$. The respective inverse operations are `outl` and `outr`: `outl(inl(t))` is `t`, and similarly for `outr`. We elide the parentheses from `inl`, `outl`, etc. when it improves readability.

We take this opportunity to introduce two ways in which union types are used in our calculational logic implementation. For one, in our type *OE* for object expressions, we conceptually separate object variables from non-object variable expressions; we reflect this by using a disjoint union type of the general form (*variables*) + (other expressions),[2] so we can syntactically determine whether or not any object expression is an object variable. Another use of union types is for a function that looks up values related to keys in a table; we can use a union type (*success type*) + (*failure type*) as the lookup function return type, for any *success type* and *failure type* we may choose. When the lookup succeeds on a key, it returns

[1] The standard semantics of Nuprl is non-classical. There are non-standard semantics for Nuprl that are classical, such as [How96]. We worked with the standard Nuprl semantics.

[2] We give a full explanation of type *OE* in the next chapter, merely using it here as a motivating example.

`inl(` s `)` for some $s \in$ *success type*; when it fails, it returns `inr(` f `)` for some $f \in$ *failure type*.

Cartesian product The expected pair constructor is present: $\langle a, b\rangle \in A \times B$. In contrast, the primitive Nuprl function for pair decomposition may be unfamiliar to readers: the form is `spread(` $p; u, v.b$ `)`, where p is a pair and b is an expression in variables u and v; `spread(` $\langle p, q\rangle; u, v.t$ `)` $= t[p, q/u, v]$. So, for instance, the standard first and second component projections of a pair can be represented as `spread(` $p; u, v.u$ `)` and `spread(` $p; u, v.v$ `)`, respectively.

Dependent types Dependent types are used to create compound types in which one component type depends on a particular value in another component type. For instance, consider a function f on integers that returns an integer on odd inputs and returns a $\mathbb{Z} \rightarrow \mathbb{Z}$ function on other inputs; we would represent the type of f as $x{:}\mathbb{Z} \rightarrow F(x)$, where $F(x) =$ if x is odd then $\mathbb{Z}$ else $(\mathbb{Z} \rightarrow \mathbb{Z})$.

A similar dependent type notion applies to products: if A is a type and B is a type-valued function on A, then an element of $x{:}A \times B(x)$ would be a pair $\langle y, z\rangle$ where $y \in A$ and $z \in B(y)$.[3]

Recursive types Nuprl's type theory can also represent recursive types. For example, consider the recursive structure of unlabeled binary trees with integer leaves; in Nuprl, it can be defined as `rec(` $node.\mathbb{Z} + node \times node$ `)`, where variable *node* is bound in the union type expression. Its elements include `inl 5`, `inr <inl 3, inl 7>`, and `inr <inr <inl 2, inl 4>, inl 6>`.

The relationship between a Nuprl type T and its members is expressed with assertions of the form $A \in T$ or $A = B \in T$. $A \in T$ expresses that A is a member of T; $A = B \in T$ expresses that A and B are members of T and equal in T.[4] In this dissertation, the form $A = B \in T$ refers only to Nuprl equality.

These are just some of the elements of Nuprl's type theory; clearly, it is a very expressive system. All the concepts above are used in our type *OE* of object expressions, but our type definitions do not generally use that much expressive capacity.

In a way, types in Nuprl are the basis for all our work, not just definitions of types needed for implementing calculational logic. Nuprl uses the propositions-as-types correspondence known as the Curry-Howard isomorphism, so its general theorem proving emerges directly from proof rules for its type theory.

[3] Other common notations for $x{:}\mathbb{Z} \rightarrow F(x)$ and $x{:}A \times B(x)$ are $\Pi x{:}\mathbb{Z}.F(x)$ and $\Sigma x{:}A.B(x)$, respectively.

[4] Type enters into the expressions of Nuprl equality because different types may have different equalities. For example, 5 and 10 are equal in $\mathbb{Z}_5$ but not equal in $\mathbb{Z}$.

In addition, Nuprl's lambda calculus —the familiar formalism extended to Nuprl's type system— is the basis for our mathematical definitions. Despite this, readers need not understand most of the concepts in Nuprl's type theory to understand this dissertation. We generally work with familiar constructs at a level of abstraction away from the type theory (constructs for recursion, case splits, etc.), and we generally explain our work that way.

4.1.2 Terms and term structure

The Nuprl data structure *term* is used for a variety of purposes. For instance, all Nuprl propositions and expressions in Nuprl's type theory are represented as terms. The mathematical/logical objects we define for our data language —as distinguished from inference methods and other higher-level procedures— are also represented by terms, so we briefly discuss Nuprl terms before presenting the data language implementation.

We do not give a full definition of Nuprl term structure, omitting many details that are not directly relevant to this dissertation. Our concise description captures the general feel of Nuprl's use of the general-purpose data structure term, and that should be sufficient.

Nuprl terms have roughly the following structure:

(4.1) $\mathit{opid}(s_1, \ldots, s_n)$.

The parts of a term are:[5]

- *opid* is the *operator identifier*; we call this feature an opid. Often, opids serve as the names of terms in our development. For instance, the opid of our object-level theoremhood predicate is *Othm* and the opid of our object-level conjunction constructor is *oand*. Readers should generally be unconcerned with low-level details such as Nuprl opids, but we do refer to them in describing our implementation in the next chapter, so we introduce them here.

- s_i is *bound-term* i of the term. Each bound-term itself has a complex structure: $s_j = x^j_1, \ldots, x^j_{a_j}.t_j$, where each of the x's is a *variable* and t_i is itself a term. This bound-term binds free occurrences of variables $x^j_1, \ldots, x^j_{a_j}$ in t_j ; this is the standard notation for variable binding in Nuprl, also used above in contexts such as the pair-decomposition function, `spread(` $p; u, v.t$ `)`.

[5]It is possible to supply other atomic parameters to operators, but we do not generally do so.

We discuss terms frequently in this dissertation without using Nuprl term notation to display them. In general, we opt for the common, intuitive notations permitted by the flexible system of Nuprl display forms, described next.

4.1.3 Display forms

Although Nuprl terms have a uniform syntax, their appearances on page or screen can differ greatly from that syntax. This is one of the strengths of Nuprl: the display forms for a term are defined at a level of abstraction away from the term and its syntax. For instance, consider pair-decomposition operation `spread` mentioned above. Displaying it in uniform term syntax can get somewhat clunky. Instead, it has an abbreviated typical display form: `spread(pair;u,v.body)` is displayed as `let <u,v> = pair in body`, or simply as `(pair/u,v.body)`, according to the user's preferences. This significantly improves readability.

As another example, consider the common propositional logic operators. In a typical Nuprl session, the logical conjunction operator would be input by typing the word 'and', corresponding to its opid and therefore its representation in the uniform term syntax presented above. But it is displayed using the character &, a convenient abbreviation; the argument slots are structured so that & appears to be an infix operator in the expected way. Similar mechanisms are used for disjunction, implication, etc., making formulas much easier to read than they would be if logical operations were expressed in the prefix/English term syntax.

This allows for the systematic, unambiguous overloading of notation: we may associate the same display form with several operators, but Nuprl manipulates the unambiguous underlying terms. We exploit this in our work, using (for instance) the symbol $\Rightarrow$ for two different infix operations: the *OE*-constructor for object-level implication and the logical/propositional implication operator in Nuprl. There is no ambiguity when entering the expressions into Nuprl —they have different names— but they look the same on the screen. We avoided overloading symbols such as & in chapter 3, but in the remainder of this dissertation, we use display forms to overload traditional logical symbols such as $\Rightarrow$ and $\vee$. This helps us correspond to both conventional Nuprl notation, in which the symbols stand for propositional operations, and the notation in [GS93b], in which they are *OE*-constructors, without any actual ambiguity. In this dissertation as in a Nuprl session, context informally disambiguates their usages.

Nuprl users can alter display forms; for instance, a change from & to $\wedge$ could be easily made. This is relevant when considering our display forms for operations relating to quantification in calculational logic, one area where we intentionally diverge from the notation in [GS93b]. For instance, the book's notation for the general form of quantification is $(\star X \mid R : B)$, whereas we use $(\star_b\, X \mid R : B)$, making explicit the indexing argument b.[6] Users who prefer that the b be elided (or perhaps want the star to be a different kind, or other modifications) could make that change. When entering the term, the b would still need to be accounted for —the term structure of the quantification wouldn't change— but it could simply be erased from the display form. We feel that this indexing is an important feature to emphasize, and using a star as a variable instead of the b would make things harder to read. This is not a conceptual divergence from [GS93b], however, and users of our calculational logic system could make the quantification display forms suit their tastes.

This system of display forms also permits case-dependent notation, where the same operator can be displayed in different display forms depending on its operand. This has numerous applications, including the ability to elide default values, which we exploit in managing calculational logic quantification. Recall that, in chapter 3, we defined the two expected predicate logic quantifiers $(\forall X \mid R : B)$ and $(\exists X \mid R : B)$ as instances of the general form $(\star_b\, X \mid R : B)$ for calculational logic quantification, based on whether argument b indicated universal or existential quantification. We implemented that directly in Nuprl using our display form for $(\star_b\, X \mid R : B)$. When argument b is filled by the constant indicating universal quantification, we simply *display* $(\star_b\, X \mid R : B)$ as $(\forall X \mid R : B)$. We do not have a separate object for the specific universal quantifier form; it is just a different notation for the same object, given a particular value for b. (Similarly for existential quantification, of course.) When the slot for b is filled by a variable in the general quantification form —say, as part of a lemma statement that quantifies over the possible kinds of calculational logic quantifiers— we use its standard display form, explicitly displaying argument b.

This practice not only makes our implementation more readable, it also encodes the desired relationship between the related quantifier forms. We do not need to define separate operators and prove relationships between them. We have only one operator, corresponding to the definition of our data language, which looks different in different contexts.

There is also a facility in Nuprl for associating user-defined *input commands* with a display form; for instance, we could type `CLquant` to get the general form for calculational logic quantification, no matter what the opid of that operator is. Combining this with the representation of default values, we can directly input operators with certain default values filled in; for instance, we could type `CLall` to get the calculational logic universal quantifier form, i.e. the general form with a particular value filled in for b. It is somewhat similar

[6] The $\star_b$ notation is what we use in our Nuprl implementation to represent the theoretical concepts expressed using $\star^M$ notation in chapter 3. Changes in notation as we move from a theoretical outline (chapter 3) to an actual implementation should not be seen as troubling.

to the effect of a macro: it is as if we typed `CLquant` and entered a value for b, all by typing `CLall`.

The value of this tremendously powerful system in our work is worth noting. In developing a language that must have the same appearance as a pre-existing notation, the flexibility to assign display forms to specific cases, access them directly, and alter display forms without affecting the underlying mathematics all make our lives as developers much easier. We can test designs, recover from notation errors, establish shortcuts, use post-hoc mnemonic names, etc., all without significant cost. It permits us all the mechanical and logical benefits of Nuprl's uniform term syntax without constraining the apparent notation of our terms. It truly separates the underlying meaning of expressions from their appearance, a tremendous virtue.

For this reason, we are somewhat loose with some aspects of notation and meta-notation in this dissertation. We may be careless with list notation in our descriptions, for instance, displaying a singleton list as its element or otherwise dropping the brackets that indicate a list in Nuprl. Such small differences in appearance between our descriptions and our direct inclusions of Nuprl notation should not confuse readers (the typing of expressions will often disambiguate cases when there is some doubt). These differences, after all, are only matters of display, not about the underlying mathematics.

We do not describe how to create Nuprl display forms. Instead, we have discussed only the most important aspects and how they are used in our implementation. For more details on display forms, see [Jac94, All98, MA94].

4.2 Nuprl ML, Tactics, and Proofs

Inference modeling uses a different set of tools from those used for formalizing the mathematical language of calculational logic. Nuprl inferences (such as those used in implementing calculational logic inference) are at a meta-level to the mathematics formalized using the concepts introduced in section 4.1. To implement inferences and various meta-level auxiliary functions, Nuprl uses a dialect of the programming language ML —the metalanguage of the Edinburgh LCF system (see [GMW79])— as its general-purpose metalanguage. Once we have introduced *Nuprl ML* and the related concepts of *tactics* and *tacticals*, we will discuss a few details of Nuprl inferences and proofs.

As with section 4.1, this is a very brief overview of an intricate system. For a more balanced and thorough introduction to the use of metalanguage in Nuprl, see [MLm93, Jac94, C$^+$86].

4.2.1 Nuprl ML

We use Nuprl ML for two distinct but complementary purposes: creating tactics for carrying out inferences and creating auxiliary functions for doing general tasks that are not well-suited for expression as inferences, such as

- simple list manipulation,
- heuristic guessing (without proof) whether two expressions might be equal,
- heuristic guessing (without proof) the type of an expression in an environment.

The second and third examples above reflect that we may want to use heuristics that are not as computationally expensive as tactics to determine if a tactic is even worth calling in a context. Running a small, non-tactic ML program is often far less time-consuming than running a tactic, so we can use ML to implement heuristics to guide tactic inferences.

Although tactics are ML programs, we tend to consider them separately from general ML programs that do not involve tactics. Indeed, we may use the phrase "ML programs" to refer to only those non-tactic programs. In this section, we discuss the ML language in general, independent of any explicitly tactic-related constructs. We discuss tactics in section 4.2.2.

In Figure 4.1, we present a subset of the ML syntax given in [MLm93, section 3.1]. See [MLm93] for more details on both these syntax equations and their associated semantics.

To make this section a somewhat more self-contained introduction, here are a few key points:

- It may be difficult to see how the local declaration construct (given under **Expressions**) is used, so we provide an example. Consider this contrived definition of a factorial-like function f:

  ```
  letrec f n =
     let non_zero i = (i > 0) in
        if non_zero n then n · (f n − 1) else 1
  ```

  (We ignore any issues about restricting values of n.) This illustrates the local declaration of function `non_zero` in the declaration of function f as well as the constructs for function application and recursive function declaration.

Declarations d

d	::=	$\texttt{let}\ b$	ordinary variables
	|	$\texttt{letrec}\ b$	recursive functions

Bindings b

b	::=	$p\texttt{=}e$	simple binding
	|	$p_1\ p_2 \ldots p_n\ \texttt{=}\ e$	function definition

Patterns p

p	::=	var	variable
	|	$p_1\texttt{.}p_2$	R list cons
	|	$p_1\texttt{,}p_2$	R pairing
	|	$\texttt{[}p_1\texttt{;}p_2 \ldots \texttt{;}p_n\texttt{]}$	list of n elements (n may be 0)

Expressions e

e	::=	ce	constant
	|	var	variable
	|	$e_1 e_2$	L function application
	|	$e_1\texttt{.}e_2$	R list cons
	|	$e_1\texttt{@}e_2$	R list append
	|	$e_1\texttt{=}e_2$	L equality
	|	$\texttt{not}\ e$	negation
	|	$e_1\texttt{\&}e_2$	R conjunction
	|	$e_1\texttt{,}e_2$	R pairing
	|	$\texttt{failwith}\ e$	failure with string
	|	$\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$	conditional
	|	$e_1\ \texttt{?}\ e_2$	failure trap
	|	$e_1\texttt{;}e_2 \ldots \texttt{;}e_n$	sequencing
	|	$\texttt{[}e_1\texttt{;}e_2 \ldots \texttt{;}\ e_n\texttt{]}$	list of n elements (n may be 0)
	|	$d\ \texttt{in}\ e$	local declaration
	|	$\backslash\ p_1 p_2 \ldots p_n\texttt{.}\ e$	abstraction

Figure 4.1: ML syntax equations

- Only functions can be defined with `letrec`. For example, `letrec x = 2-x` is syntactically incorrect.
- All the variables occurring in a pattern must be distinct. On the other hand, a pattern can contain multiple occurrences of the wildcard '`()`'.

4.2.2 Tactics

The *tactic* structure of Nuprl (based on [GMW79]) is one of the primary reasons it seemed practical to model cognitive inference in an automated reasoning system. (See [CKB84] for elaboration on the idea of tactics used in Nuprl and their representation of mathematical thinking.) Given a collection of pre-specified primitive inferences, a tactic is a program for reducing a proof goal to premises by iteration of these primitive inferences. Essentially, a tactic is a program for constructing such an inference tree; which tactics are applied depends on how one wants to generates subgoals from the proof goal.[7] Executing a tactic gives rise to an inference step; the premises of that inference are the unproved leaf-premises of the primitive proof tree.

The execution of a tactic might raise an exception or fail to terminate. For a program to count as a tactic, however, if it terminates without exception, it must be guaranteed to generate subgoals justifiable by the primitive rules.

In order to freely write heuristic tactics, there must be some practical criterion for recognizing tactics. In ML, this criterion is type checking to the type of tactics.

The labor of justifying inference is split between justifying "tactichood" and checking the primitive inference rules. (Presumably, the primitive inference rules are formulated to make such verification feasible. For example, they tend to be schematic.) The justification of complex forms of inference via tactics is in terms of the generic criterion for being a tactic and is not schematically described.

Here are a few simple tactics in Nuprl that pertain to points of interest in our development:

- `Id` is the identity tactic; applying it to a proof goal leaves it unchanged. As we illustrate shortly, it is useful in constructing complex tactics from simple ones.
- `FailWith` takes a character string as an argument. When applied to a proof goal, it fails and outputs its string argument as an error message. In conjunction with a failure-trapping mechanism, it permits developers

[7] We discuss tactics from a more cognitive science oriented perspective in chapter 9, page 150.

to output more informative, customized failure messages than Nuprl's generic ones.

- `AUTO` is perhaps the most important tactic in Nuprl. It performs a wide variety of simple inferences, such as typing judgments, some trivial equalities, and some trivial proofs from hypotheses. In essence, `AUTO` represents the class of inferences considered too obvious for Nuprl's users.

  In implementing calculational logic inference, we did not use `AUTO` in our tactics. It is very high-level, and tactics built around it can be fragile because someone may want to change `AUTO` and the class of obvious inferences it represents without affecting other, more specific tactics. Therefore, we presume that users of our calculational logic tactics will invoke `AUTO` themselves.

- There are several varieties of chaining lemmas in Nuprl. For instance, `BackThruLemma` is a simple backchaining tactic, whereas `BackThru` is a more complex one, wrapping `BackThruLemma` in levels of pattern-matching and other mechanical intelligence. There are also other tactics for forward chaining, backward chaining, and lemma instantiation, and we use some varieties of them extensively.

  For reasons similar to those expressed regarding `AUTO`, above, we used only the lowest-level chaining tactics in our calculational logic implementation. Indeed, higher-level chaining tactics may themselves call `AUTO`, so the identical reasons apply.

Although all of Nuprl ML can be used to create new tactics from old ones, for common applications, it is often simpler to use Nuprl's language of *tacticals*, which are functions for composing tactics. We made extensive use of tacticals in creating our calculational logic tactics, but their use as programming constructs is so simple that we do not spend much space here even introducing them. Here are a few simple examples of tacticals; see [Jac94] for a more complete list.

- `REPEAT` is analogous to the common programming language looping construct: for tactic T, `REPEAT` T repeatedly runs T on the subgoals resulting from previous applications until no progress is made.

- `ORELSE` is the failure-trapping tactical: for tactics $T1$ and $T2$, $T1$ `ORELSE` $T2$ tries running $T1$; if it fails, it runs $T2$ instead. With the `FailWith` tactic mentioned above, this can be used for customizing error messages.

- `Try` T tries to run tactic T on a proof goal, but if T fails, it leaves the proof goal unchanged; it is defined as T `ORELSE Id`. This is extremely useful in tactic development.

- $T1$ `THEN` $T2$ first runs $T1$ and then runs $T2$ on all subgoals generated by $T1$. The combination of a tactic T with `AUTO`, written as T `THEN AUTO`, has an abbreviated alternative display form: T ..... This abbreviated form comes up in example (4.3) in section 4.2.3 below.

- `Complete` T is useful for determining if tactic T will finish a proof; it runs T and fails if T generates any subgoals.

Tactics can get increasingly complex; there is a full programming language for use in constructing them. Some complex inference patterns could also be captured in lemmas; a lemma can represent the result of a chain of inferences just as a tactic can. In this way, there is an interesting division of inferential labor, as it were, between tactics and lemmas: increased reliance on one of them can facilitate less dependence on the other. In our implementation, we tended to use lemmas where it seemed helpful, rather than create more complicated tactics. This was, however, simply a design decision. If our important inferences had been more clearly expressed by tactics than lemmas, we would have decided differently.

4.2.3 Proving things in Nuprl

Nuprl's inference style is based around *sequents*. A sequent is written as $H_1, \ldots, H_n \vdash C$, where C is the *conclusion* of the sequent and each H_i is either a *hypothesis* or a *declaration* of a variable with its type. Normally, Nuprl displays sequents vertically, with explicit numbers for the hypotheses:

$$
\begin{array}{ll}
(4.2) & 1.\ H_1 \\
& \vdots \\
& n.\ H_n \\
& \vdash C
\end{array}
$$

We may refer to the hypotheses and the conclusion of a sequent as *clauses*. In addition, by *goal* (or *proof goal*) we may refer to either a full sequent or only its conclusion.

Our introduction to Nuprl proofs is a simplification of the more detailed one in [Jac94] and other sources, intended primarily to permit readers to understand the rest of this dissertation. We see proofs as tree structures in which every node has a sequent component and a tactic component; the tactic component of a node may be empty or otherwise ill-formed. The children of a node N are the subgoals generated by the tactic of N applied to the goal of N. If a node has no children, its tactic fully solves its proof goal.

As an example of how tactics generate subgoals, consider the following contrived example for backchaining:

(4.3)
```
1. propn1 : Prop
2. propn2 : Prop
⊢  ¬(propn1 ∨ propn2)
by  BackThru: Thm* ∀A,B:Prop. ¬A & ¬B ⇒ ¬(A ∨ B) ....
 \
  ⊢  ¬propn1  by  <TACTIC>
  ---
  ⊢  ¬propn2  by  <TACTIC>
```

It backchains through a simple lemma, then calls tactic `AUTO` —which is represented in abbreviated form by the four dots after the backchained lemma— to handle routine manipulation (proving typing subgoals, etc.). Note that antecedents in the lemma became subgoals after the tactic. We will refer again to this pattern of antecedents becoming subgoals in the forthcoming chapter explaining our implementation of calculational logic inference.

In the expected way, a proof is complete when all of its nodes have the expected properties: all variables in every sequent are bound in that sequent; all nodes have a tactic; every goal is proved by its tactic, assuming provability of its children; etc.

Proof goals in Nuprl also have a *label*, which roughly indicates their classification or purpose. For instance, goals corresponding to the primary inferences generally have the (normally elided) default label `main`; other labels, such as `assertion` and `rewrite subgoal`, mark proof goals that arise in particular circumstances with which most readers of this dissertation need not be concerned. As developers, we must take some minimal care to manage our tactics correctly on these labels. With one exception, discussed immediately below, users of the system never encounter them.

The one kind of non-`main` proof goal that is significant to users of our calculational logic system is *well-formedness* goals, which are used to establish that Nuprl expressions have the necessary types; they have the label `wf`. Well-formedness goals are critical and pervasive in Nuprl, because everything must be type-correct. They are typically handled automatically by Nuprl's `AUTO` tactic. Often, when Nuprl users want to invoke a tactic *TAC*, they wrap it in `AUTO`, instead applying ***TAC*** `THEN AUTO`. It is a strength of the tactic system of Nuprl —and its `AUTO` tactic in particular— that Nuprl users are often insulated from typing judgments and other simple inferences. Further, when `AUTO` does not solve a well-formedness subgoal, it is often a useful indication of user error; when a user fails to assign a correct type to a variable, for instance, it comes up in the form of an unsolved `wf` goal.

In our implementation of calculational logic inference, we assume that our tactics will be wrapped in tactic `AUTO`, so we do not try to solve well-formedness subgoals ourselves. We do solve all other subgoals that may emerge, no matter what their label, but users of our calculational logic tactics who do not also call `AUTO` may well be greeted with a message from Nuprl informing them that there are hundreds of unsolved subgoals remaining. In our experience, these are all `wf` subgoals. Well-formedness subgoals that remain after `AUTO` have the same value as in any other Nuprl context: they frequently indicate user error. Our tactics do nothing to obscure this.

4.3 Concluding Remarks

As mentioned in this chapter, specific features of Nuprl guided our design decisions in several ways. Prominently, Nuprl's type theory influenced our choice for type *OE*, which affected the rest of our development. Nuprl's many degrees of abstraction —between syntax and display, between term-level and meta-level, etc.— also affected the overall structure of our mathematics and our implementation of calculational inference, as did the relative simplicity of expressing some procedures as tactics and others as general ML programs.

We conclude this chapter with a few comments on how we used tactics. One of the primary aspects of our design philosophy for tactics has already come up in our discussion of `AUTO`, but it bears repeating: it was our conscious goal to keep our tactics as low-level as possible, to build them from component tactics that are themselves as close as possible to Nuprl's primitive rules. One result of this is that our tactics never called `AUTO` directly, but we also never used high-level tactics for chaining, expression decomposition, etc. This results in a more robust system, because Nuprl's high-level tactics are more likely to be changed from version to version (or customized from user to user) than low-level ones, and our dependence on these more variable tactics is minimized.

In addition, because our implementation is part of a feasibility demonstration for our overall cognitive modeling method, we frequently used an exceptional tactic that is not mentioned on the list in section 4.2.2: `Fiat` is a tactic that, when applied, signals Nuprl to treat an incomplete proof as if it were complete. Indeed, it embodies the idea of "proving by fiat," resulting in Nuprl's accepting a proof goal as fully solved without further justification. `Fiat` could even prove `False`, which makes it extremely dangerous to use. We used it regularly to speed up our implementation of calculational logic inference —instead of spending time proving obvious properties of our implementation, we simply stated them as lemmas and used them in our tactics, a philosophy we refer to as *state-and-use* in future chapters. If there are errors in our development, they probably arise from gaps in reasoning that are artificially filled with `Fiat`.

In fact, we nearly overlooked such an error. We created a heuristic ML program `termeql` to tell if two Nuprl terms are equal modulo a simple equivalence relation that we did not want to take the time to implement formally in Nuprl; then, we wrote a tactic that essentially claimed that if we wanted to prove a goal $A = B \in T$, the truth of `termeql` A B was sufficient. Instead of formally working through all the reasoning for such a tactic, we used `Fiat`, and that resulted in a false tactic: it failed to account for all the typing information needed to prove such a goal. We believe that no such errors currently exist in our system, but as a matter of full disclosure, we felt compelled to mention their possibility due to our use of `Fiat`.

Chapter 5

Implementing the Data Language

5.1 Introduction

Some of the data language given in chapter 3 is already present in Nuprl: for the standard logical operations of the data language, we use the Nuprl propositions $\Rightarrow$, $\&$, $\neg$, etc.; we also use Nuprl's own list construction and pairing operations, and Nuprl's types for natural numbers ($\mathbb{N}$), pairing (the product type $T1 \times T2$), and lists. We introduced by definitions the rest of the data language into Nuprl, including the fundamentals of our calculational logic object language and how we manipulate it. In this chapter, we describe our Nuprl implementation of the data language —from the object language on which it operates to the complex propositions it contains for *OE*-typing and object-level theoremhood— providing all the details needed in advance of our explanation of calculational logic inference, given in the next chapter.

In general, our implementation is a reflection of the data language in chapter 3, capturing the same essential features of calculational logic. That chapter gave only a theoretical outline of how one could formalize the predicate logic in [GS93b], however, not a description of an actual implementation. Because of the constraints of a practical context, our Nuprl implementation differs from the theoretical outline in several ways, ranging from different display forms for operations mentioned in chapter 3 to additions of new operations to the data language. At the conclusion of this chapter, we hope readers will be comfortable with our Nuprl definitions and display conventions —even those that differ from chapter 3— in preparation for their use in the formalization of Leibniz inference that follows. In addition, we made several design decisions as system developers that significantly shaped our implementation, and we hope readers will be comfortable with this aspect of the Nuprl implementation as well. One of these design decisions, organizing definitions around the notion of a *user level*, affects several of our definitions, so we discuss the user level before getting to the implemented data language.

5.1.1 The user level

One of our considerations was to distinguish the definitions intended to aid us in developing the system from those intended for direct use in expressing the mathematics of [GS93b]. Some of our implemented definitions are on the developer-level (or system-level), and some are on the *user level*. For example, in some contexts, we have several closely related definitions —such as different *OE*-typing predicates, one defined on a single *OE* and another defined on a list of *OE*s — that are all part of our implementation. We want to distinguish user-level constructs, those intended for users of our calculational logic system, from those intended only as auxiliary, behind-the-scenes elements.

For this purpose, we consider a user of our system to be someone who wants to use our formalized data language and inference tactics to state and reason about the calculational predicate logic in [GS93b]. People whose aim is to develop the data language or implement calculational logic inference procedures (whether tactics or simply plain ML code) are *not* considered users. So it is not necessarily the most fundamental elements of our implemented language, the ones on which our definitions are based, that are considered "user-level". Indeed, we must choose elements that are general enough to handle all the intended uses. For instance, in the *OE*-typing example in the preceding paragraph, the *OE*-typing predicate on a single expression would not handle necessary list-cases —it is natural to state propositions such as $(\epsilon)\ P, Q, R ::$ bool in the data language of [GS93b]— so that would not suffice for the user level of *OE*-typing predicates. So, we decided that the list-based *OE*-typing predicate would be the only one for users of the system; users must state all *OE*-typing propositions using this list-based form. This avoids over-populating the user level, making a cleaner system for users and system developers alike.

In this chapter, we explicitly point out which elements of the data language are considered user-level, when it may be ambiguous. This is particularly relevant for *OE* quantification, *OE* textual substitution, and propositions on *OE*s .

5.2 Nuprl Types for Calculational Logic

As with any typed language, the foundation for our data language is its types. We implemented Nuprl types for concepts such as object expressions (*OE*), object variables (*OV*), and object-level type expressions (*OTS*). As discussed in chapter 4, we carried out these definitions in Nuprl's existing type system. Some of our design choices were necessitated by the Nuprl environment, while others were made primarily to simplify the implementation.

5.2.1 Notes on object expressions and object variables

For perspective on some of these definitions, it is helpful to consider what we mean by an object expression, i.e., what objects we want the class OE to contain. As a first (very rough) approximation, OE can be described by the following grammar, where $IDENT$ represents a class of identifiers that we temporarily take as object language variables:

$$(5.1)\quad OE := IDENT \mid \mathit{false} \mid OE(OE) \mid OE \Rightarrow OE \mid OE = OE \mid \forall\, IDENT.\ OE$$

As a refinement of this, we decided to conceptually separate object variables: for us, an object expression is either an object variable or a function on object expressions. (We consider *false* a constant function.) Implementing this required using Nuprl's disjoint union type $T1 + T2$ in our definition. A second approximation to the definition of OE would be

$$(5.2)\quad OE == IDENT + (\text{some recursive structure in } OE)$$

where the recursive structure captures the syntax expressed in (5.1). We give the exact details of the recursive structure shortly, after presenting further preliminaries.

Having decided upon this form for object expressions, considerations arose about defining our class of object variables. In particular, it seemed natural that the class OV of object variables would be a subtype of OE. We could not, however, simply use $IDENT$ as OV. Within the above framework of equation (5.2), members of $IDENT$ are not members of OE; members of a Nuprl type $A + B$ have one of two forms, $\texttt{inl}(a)$ or $\texttt{inr}(b)$, where $a \in A$, $b \in B$, and `inl` and `inr` are tags to represent which argument of the union type to consider. Thus, terms $\texttt{inl}(id : IDENT)$ are elements of OE, and we base OV on them. We defined the object variable constructor $Ovar(id : IDENT)$ to be just $\texttt{inl}(id)$ and the type of object variables OV to be $\{Y{:}OE \mid \exists id{:}IDENT.\ Ovar(id) = Y\}$. We chose this formulation particularly to highlight the subtyping relationship $OV \subseteq OE$.

In many treatments of logical languages, these notions of object variable and identifier appear to be the same. Our Nuprl types of OV and $IDENT$ are different, but they are also easily identified with each other, with $Ovar$ injecting identifiers into object variables in the obvious way. For the sake of computational simplicity, we sometimes exploit this structural identity, defining some operators that conceptually relate to object variables by specifying their actions on identifiers.

5.2.2 Important Nuprl types for calculational logic

In this section, we present some of the more important Nuprl types that are introduced in our implementation. In a way, they provide an overview of the whole project by highlighting the key concepts. This list is thorough, but not absolutely complete —some simple, application-specific definitions will be delayed until forthcoming sections when they become relevant.

Along with the types, we discuss related constructs, such as auxiliary functions (e.g., the update functions for type expression assignments) and type-specific constructors and destructors that make the language easier to use. We omit straightforward or otherwise uninteresting details —for instance, we also implemented decidable equalities on types such as *IDENT* and *OV* and used them in mundane ways in our development.

identifiers We defined $IDENT == Atom \times \mathbb{N}$. Thus, identifiers are character strings followed by (i.e. paired with) some natural number. The presence of $\mathbb{N}$ is central to our methods for creating fresh variables, used in implementing capture-avoiding substitution. We also implemented the obvious constructors and destructors on *IDENT*: Operators $ident1$ and $ident2$ are projections of the atom and number of an identifier, respectively; operator $identform(a : Atom, n : \mathbb{N})$ constructs $\langle a, n\rangle{:}IDENT$.

object variables Recall that OV is $\{Y{:}OE \mid \exists id{:}IDENT.Ovar(X) = Y\}$. As support, we implemented the straightforward destructor function $identofovar{:}(OV \rightarrow IDENT)$ and two varieties of recognizer functions, respecting Nuprl's use of the mathematical distinction between Boolean values and propositional values: $is\_a\_ovar(oe{:}OE)$, which is just an alias of Nuprl's `isl` function, returns a value of type $\mathbb{B}$; $IsOvar(oe{:}OE)$, which uses the set-type definition of OV given above, returns a value of type $\mathbb{P}$.

We also implemented OV-constructor `ov_lit` to create OV-literals: for any character string, it creates the OV consisting of that string (treated as an element of *IDENT*) paired with the number 0. When the $\mathbb{N}$ argument in an OV is 0, we typically elide it, so `ov_lit(p)` displays as `p`. OV-literals do not play a big role in our implementation of calculational logic —character strings are not object variables, they are metalinguistic expressions that stand for object variables— but sometimes [GS93b] implicitly suggests that readers treat letters as object variable literals. To permit this direct reading, constructor `ov_lit` is a necessary part of formalizing the conventions of [GS93b] in Nuprl. Still, it is atypical to escape the metalinguistic basis of calculational logic in this way. In fact, we did not use any OV-literals in the examples we created to test our definitions and develop our model of calculational logic inference.

OPIDS As mentioned in chapter 4, each Nuprl term has an opid that designates the operator from which that term was formed. We use a similar system, implementing the five-element type *OPIDS* to identify the five fundamental operators in object language syntax. Following common Nuprl practice for such finite types, we define *OPIDS* as $\mathbb{N}_5$, and we refer to its elements by constants *opid_false*, *opid_ap*, *opid_imp*, *opid_eq*, and *opid_all*, where a term with opid *opid_false* is the object language constant for false, a term with opid *opid_ap* is an object language function application, etc. There is no ambiguity between Nuprl opids and the opids we implemented for our object language.

NonOV An object expression that is not an object variable is characterized by the presence of an opid and a subterm-function. To capture this class, we defined the set type $\mathit{NonOV} == \{Y : \mathit{OE} \mid \neg(\exists X : \mathit{IDENT}.\mathit{Ovar}(X) = Y)\}$, the complement of OV with respect to OE.

We defined the *NonOV* constructor $\mathit{mknonov}(\mathit{opid}, \mathit{fn\_of\_OE})$ to be `inr` $\langle \mathit{opid}, \mathit{fn\_of\_OE}\rangle$, the composition of the Nuprl `inr` tag and the pairing operator; an example of its use is given in the discussion of type OE that follows.

object expressions Type OE is implemented as the recursive type

$$(5.3)\quad \mathit{OE} == \mathtt{rec}(\mathit{OE}.\ \mathit{IDENT}+ \\ (i : \mathit{OPIDS} \times (j : \mathbb{N}_{oesbtms(i)} \rightarrow \mathit{OE} \times \mathit{PossBV}(i;j))))$$

As previously discussed, elements of Nuprl type OE are either members of OV, with the form $\mathit{Ovar}(id)$, or elements of NonOV, constructed by $\mathit{mknonov}(\mathit{opid}, \mathit{subterm\text{-}function})$. The *subterm-function* portion is a function from a finite type with as many elements as there are OE subterms of a term with opid *opid* —represented by $\mathbb{N}_{oesbtms(i)}$ in the definition of type OE— to the subterms themselves.

The return type of this subterm-function is actually $\mathit{OE} \times \mathit{PossBV}(i;j)$, which merits a brief explanation. In case the term is a quantifier $(\forall x.P)$, it will have both a subterm P and a variable x bound in that subterm, and this function will return the pair $\langle P, x\rangle$. In other cases, there will be no binding variable, and the pair returned will be $\langle \mathit{subterm}, \cdot\rangle$, where placeholder $\cdot$ indicates the absence of a binding variable for this subterm. Further details on PossBV follow in subsection 5.3.2.

We frequently use the fact that any element of OE is also in type $\mathit{IDENT} + (i : \mathit{OPIDS} \times (j : \mathbb{N}_{oesbtms(i)} \rightarrow \mathit{OE} \times \mathit{PossBV}(i;j))))$, which follows from one unrolling of the recursive definition of OE. We capture that fact in straightforward theorems relating the types: each type is a subtype of the other, and terms equal in one type are equal in the other.

object-level type expressions As Nuprl terms, all object expressions have Nuprl type *OE*. As discussed in chapter 3, however, there is also a different kind of typing for object expressions. Given a class of individuals, an object expression stands for exactly one of the following: an individual, a boolean value, an individual-valued function, or a boolean-valued function. We designed *object type expressions* that refer to the classes of values for which an *OE* can stand; we then defined a Nuprl type OTS whose members are these object type expressions. Here are the definitions that support Nuprl type OTS:

object-level ground types We introduced a two-valued type $OGT = \{IND_O, \mathbb{B}_O\}$, where IND_O and $\mathbb{B}_O$ are atomic constants representing the types of individual-valued and boolean-valued object expressions.

Nuprl type *OTS* Type OTS of object type expressions is defined to be $OGT \times \mathbb{N}$; we defined the OTS-constructor $(n{:}\mathbb{N})gt{:}OGT$ as simply the pair $\langle gt, n\rangle$. The $\mathbb{N}$ element is used to represent the functional arity of terms. To illustrate, a boolean-valued function with four arguments would have type expression $(4)\mathbb{B}_O$, and an individual would have type expression $(0)IND_O$.

object type expression assignments We use the name *type expression assignment* (we may abbreviate it to *type assignment*, when unambiguous) for a structure of type $OV \rightarrow OTS$ that represents the assignment of type expressions to variables. As discussed in chapter 3, even though type assignments are defined on OVs, we think of type assignments as specifying type expressions for all object expressions.

Typically, type assignments are not fully specified; proofs and definitions are implemented by quantifying over type assignments. To specify values for particular variables, we use type assignment updates, which we discuss fully below. If necessary, we treat type assignments as total functions by having a default mapping of OVs to $(0)\mathbb{B}_O$ unless otherwise specified.

type assignment updates We specify type assignment functions by updates: ϵ^x_τ specifies that x is assigned type expression τ and all other variables are the same as in ϵ. There are three kinds of type assignment updates, used in different practical contexts.

- One type assignment update takes a single object-variable x and a type expression τ, updating x to type expression τ. Because of its specificity and simplicity, we as system developers adopt this as our standard form when creating tactics to reason about updates. We have implemented tactics to rewrite the other two kinds (described immediately below) to this one in straightforward ways.

Although we prefer it as developers, this is not a user-level construct: in normal practice, there are many instances where a list of object variables is updated, and this update cannot accommodate that operation in a simple way. This illustrates a concrete difference between user-level and developer-level goals.

- A second type assignment update takes a list x of object variables and a single type expression τ, updating all the variables on x to τ. We defined it in the straightforward way from the preceding (single-OV) update.

  This is the user-level update. It can easily be used to express the same information as in the single-variable case by using singleton lists, but it is also general enough for its other important uses: For instance, if the quantifier $(\forall X \mid R : P)$ (where $X{:}OV\,\mathrm{List}$) is assigned bool under ϵ, the natural typing information to infer about R is $(\epsilon^{X}_{\mathrm{ind}})\ R ::$ bool , using precisely this kind of type assignment update.
- The other type assignment update takes a list x of object variables x and a list τ of type expressions. We defined this one, too, in the straightforward way from the single-OV update.

  This kind of update emerged as a useful concept for simplifying certain aspects of our reasoning as developers of inference methods, and in chapter 6 we discuss its use in instances where the other two kinds of update would be inadequate. The extra degree of abstraction gained by taking lists τ is of no benefit in representing the mathematics in [GS93b], however, so we do not complicate matters by introducing it to users.

type assignment agreement We implemented a predicate to capture the concept of two type assignments agreeing on an object variable: $(\epsilon, \epsilon'$ agree on $x)$ holds exactly when type assignments ϵ and ϵ' assign the same type expression to $x{:}OV$. This structure is useful in reasoning about *OE*-typing: if two type assignments agree on all free variables in an $E{:}OE$, then they assign the same type to E. We return to this idea in chapter 6, when we describe our methods for solving *OE*-typing proof goals.

{ ind , bool } Due to the polymorphism of object-language equality over booleans and individuals, { ind , bool } is a particularly important type. We often use it when proving properties involving equality, and since the three Leibniz rules are about object-level equality, it comes up frequently and prominently. We use Nuprl's set type to define it as a subtype of *OTS* in the straightforward way.

environments For a type T, an environment $env(T) == (IDENT \times T)$ List associates identifiers with values of T. We use environments to manage the lists used in implementing textual substitution of calculational logic. We implemented two functions to support environments:

lookup $lookup\_env(id{:}IDENT, env{:}env(T))$ is a general lookup function for environments of any type T, returning an element of $T + \texttt{Unit}$. ($\texttt{Unit}$ is a Nuprl type for which $\cdot$ is its only member. We use it here only to indicate lookup failure.) It is a simple list-recursive function on env, finding the first pair on env for which the first element is id, say $\langle id, t\rangle$, and returning $\texttt{inl}(t)$. If no such pair exists, it returns $\texttt{inr}(\cdot)$.

update Because *lookup_env* always returns the first match (if any), the environment update functions simply add to the head of an environment. We have both a general $updates\_env(l{:}env(T),\ T{:}env(T))$ function that prepends one environment to the head of another and a specific $update\_env(var{:}IDENT,\ val{:}T, env{:}env(T))$ function for updating by one element, defined for any type T.

zip For any type T, function $zip\_env(I{:}IDENT \text{ List}, L{:}T \text{ List})$ creates a T-environment by joining the two lists: element i of the environment is the pair formed from element i of I and element i of L.

5.3 The Object Language

Our Nuprl implementation reflects the metalinguistic interpretation of [GS93b] discussed in chapter 3. In particular, we read proofs and propositions in calculational logic as being about the *object-level theoremhood* (or *object theoremhood*) of expressions in an *object language*. We need only a relatively standard predicate logic for our object language, although there are some intricacies in the details of our implementation. In this section, we present the implementation and some of the auxiliary operators we introduced to increase its utility.

5.3.1 Object expression constructors

We covered simple object variables and constructor $Ovar$ in the previous discussion of type *OV*. We also introduced type $OPIDS$, the five opid constants that represent the five fundamental functions in *OE* syntax, and *OE*-constructor $mknonov$. Here, we elaborate on that introduction to object expressions.

fundamental *OE* syntax We introduced *OE* constructors for the five fundamental elements of the syntax of object expressions: *ofalse*, *obap(p,q)*, *oimp(p,q)*, *oeq(p,q)*, and *oforall(x,p)*; we display them as $\mathbf{f}$, $p(q)$, $p \Rightarrow q$, $p = q$, and $(\forall x \mid: p)$, respectively. As an example of an *OE* constructor, consider

$$(5.4)\quad p \Rightarrow q == \\ \mathit{mknonov}(\mathit{opid\_imp}, \lambda\, \mathtt{posn.\ If\ posn} = \mathtt{0\ then}\ \langle \mathtt{p}, \cdot \rangle\ \mathtt{else}\ \langle \mathtt{q}, \cdot \rangle).$$

It contains an *opid* , a subterm-function, and the *mknonov* operator. Other *OE* constructors are similar; we note differences later, where appropriate.

***OE* false** The *OE* constant for false, $\mathbf{f}$, is $\mathit{mknonov}(\mathit{opid\_false}, \lambda posn.0)$. The subterm function $\lambda posn.0$ may seem odd, but, in fact, because its intended domain $\mathbb{N}_{oesbtms(opid\_false)} = \mathbb{N}_0$ is empty, any choice of subterm function would suffice.

***OE* universal quantification** The *OE* constructor for the basic universal quantifier form $(\forall p \mid: q)$ is $\mathit{mknonov}(\mathit{opid\_all}, \lambda posn.\langle p, q \rangle)$. Similarly to the case for $\mathbf{f}$, by using type $\mathbb{N}_{oesbtms(opid\_all)} = \mathbb{N}_1$ as a guard in applications, we are free to implement its subterm-function as a constant function. In addition, recall what the pair $\langle p, q \rangle$ as the return value in the subterm-function indicates: the variable represented by p is bound in the *OE* represented by q .

There is significantly more complexity to our implementation of quantification, reflecting the usage and conventions of calculational logic. Due to the necessary detail, we discuss the implementation of general quantifier forms in the next section.

defined elements of *OE* We build the rest of our object language from the fundamental object language syntax, defining the *OE* constructors $\mathbf{t}$, $\neg$, $\vee$, $\wedge$, $\exists$, and $\equiv$ in straightforward ways.

5.3.2 Quantification

We have previously described universal and existential quantification in the calculational logic of [GS93b] and our metalinguistic interpretation of it. Based on that description, we implemented our term constructors for universal and existential quantifiers of the object language in the expected way.

In addition, recall that there are added complexities to the treatment of quantification in [GS93b]. In chapter 3, we described the generalization of standard predicate logic quantifiers to a general form $(\star^M X \mid R : P)$, which can accommodate accumulator operations such as $\sum$ and $\prod$ as well as the quantifiers $\forall$ and $\exists$. (Recall, too, that X is a list of dummy variables; it may be a singleton list.) Our description, however, covered only the general form and the two standard quantifiers, because our goal was an implementation of predicate logic that could be readily extended to other areas of the discrete mathematics in [GS93b]; we did not formalize an implementation of those other areas. Using a two-element type *Qind* of quantifier indices (and constants $\wedge$ and $\vee$ to refer to those indices), we implemented quantification according to that description. We present the details here.

For a foundation, we used the fundamental OE-constructors for universal and existential quantifiers to implement $(\star_{b:Qind}\ x{:}IDENT \mid: p{:}OE)$, defined by cases on b: $(\star_\wedge\ x \mid: p) = (\forall x \mid: p)$; $(\star_\vee\ x \mid: p) = (\exists x \mid: p)$. This is more specific than the general form in two ways: the binding slot is restricted to a single identifier instead of a sequence of dummies, and there is no expression R restricting the range of the quantifier, corresponding to $R = \mathbf{t}$. It is not intended for the user level; it is used solely in defining the general (user-level) quantification.

From this, $(\star_{b:Qind}\ X{:}IDENT\ \text{List} \mid R{:}OE : P{:}OE)$ is defined by induction on X:

$$(5.5)\quad ((\star_b\ X \mid R : P) = \begin{cases} base(b, R, P) & \text{if } X \text{ is the empty list} \\ ((\star_b\ h \mid: (\star_b\ t \mid R : P)) & \text{if } X = h.t \end{cases}$$

where $base(b,r,p)$ is also defined by cases on b: $base(\wedge, r, p) = r \Rightarrow p$; $base(\vee, r, p) = r \wedge p$. The *base* construct is not intended for the user-level. It is simply an auxiliary form for the general definition of quantification.

We use this general, user-level quantifier form $(\star_b\ X \mid R : P)$ to define specific OE quantifiers: in concept and in Nuprl display, $(\star_\wedge\ X \mid R : P)$ is $(\forall X \mid R : P)$ and $(\star_\vee\ X \mid R : P)$ is $(\exists X \mid R : P)$. As mentioned in chapter 4, the 3-ary user-level universal and existential quantifiers are essentially just display forms of the general 4-ary quantifier.

To represent the cognate binary operators and relevant identity elements, we use the infix operator $(P{:}OE)\ \star_{(b:Qind)}\ (Q{:}OE)$ and $u(b : Qind)$, respectively. They are defined in the expected way by cases on b; for instance, $P\ \star_\wedge\ Q = P \wedge Q$ and $u(\wedge) = \mathbf{t}$. These definitions are direct reflections of the mathematics previously given.

It should be reasonably clear how to extend this system of definitions to mathematics beyond predicate logic; for instance, *Qind* could be extended to a 4-element type, adding the constants $\times$ and $+$, etc.

5.3.3 Object expression destructors and related functions

fundamental *OE* destructors Recall that, if an object expression is in *NonOV*, it is essentially an opid and a subterm-function. The *OE*-destructor $\mathit{opid\_of\_OE}(\mathit{oe}{:}\mathit{NonOV})$ returns the opid of an object expression oe; $\mathit{fn\_of\_OE}(\mathit{oe}{:}\mathit{NonOV})$ returns its subterm-function.

number of subterms of *OE*s The $\mathbb{N}$-valued function $\mathit{oesbtms}(\mathit{opid})$ is the number of subterms of an *OE* with opid $\mathit{opid}{:}\mathit{OPIDS}$. This is commonly used in a type $\mathbb{N}_{\mathit{oesbtms}(\mathit{opid})}$ to specify a property P for all subterms of any object expression: e.g., $\forall \mathit{opid}{:}\mathit{OPIDS}, \mathit{posn}{:}\mathbb{N}_{\mathit{oesbtms}(\mathit{opid})}.\ P$.

This $\mathbb{N}_{\mathit{oesbtms}(\mathit{opid})}$ type comes up frequently in Nuprl well-formedness subgoals. To prove properties about definitions of *OE*-constructors, we often need to prove something about subterms, such as how many subterms a given *OE*-constructor has. Thus, this *oesbtms* function is very important to us as developers, though not to users.

subterms of *OE*s The *OE*-valued function $\mathit{nthsbtm}(\mathit{oe}{:}\mathit{NonOV}, n)$ returns subterm n of oe. The restriction of oe to type NonOV ensures that oe has a subterm-function; we treat requests for the subterms of *OV*s as ill-typed. There is also a type restriction on n — n must be in $\mathbb{N}_{\mathit{oesbtms}(\mathit{opid\_of\_OE}(\mathit{oe}))}$ — to ensure that n is actually a value for which expression oe has a subterm. For instance, asking for the fifth subterm of an object-level implication would be considered ill-typed.

bound variables The $\mathbb{N}$-valued function $\mathit{oebv}(\mathit{opid}{:}\mathit{OPIDS}, \mathit{posn}{:}\mathbb{N})$ returns the number of variables that become bound in subterm position *posn* in an *OE* with opid *opid*. In our language so far, *oebv* returns only 0 or 1. Like quantification and some other implemented features, its syntax accommodates extensions to a more general language than the one we use.

Type-valued function $\mathit{PossBV}(\mathit{opid}{:}\mathit{OPIDS}, \mathit{posn}{:}\mathbb{N})$ calls *oebv* to determine if a binding slot is associated with position *posn* of an object expression with opid opid. (We use the singular —"a slot"— because *oebv* returns only 0 or 1.) If so, PossBV returns the type of that variable, which in our application is always IDENT; if not, it returns `Unit`. The primary use of PossBV is in the return type of subterm-functions of *OE*s, $\mathit{OE} \times \mathit{PossBV}(i;j)$.

5.4 Substitution

Capture-avoiding textual substitution on *OE*s is central to calculational logic. We cannot simply use Nuprl's native substitution to perform this, because OE terms have a wholly different binding structure. In this section, we discuss the capture-avoiding substitution on OE that we implemented, both the general idea behind its design and the details of its implementation.

Calculational logic often uses capture-avoiding simultaneous substitution, so we defined the general substitution and then defined the common special case of the substitution of one expression for one variable as a special case of it, using singleton lists. In broad terms, our approach to computing the capture-avoiding substitution of $P{:}OE$ List for $V{:}IDENT$ List in $E{:}OE$, written $E[V := P]$, is as follows:

- Identify a number n so that, for any $id{:}IDENT$, $ident2(id) > n$ implies both that $Ovar(id)$ does not occur free in E or P and that $Ovar(id)$ does not appear on V.
- Rename the identifiers in bound variables of E to new identifiers for which $ident2$ is greater than n; call the result of the renaming E'.
- Having now eliminated the possibility of capture, perform a capture-permitting simultaneous substitution of P for V in E' This is done by a straightforward recursive descent through E', rewriting any free variable with identifier in list V to the corresponding expression in list P.

Here are the operations we implemented to support this capture-avoiding substitution on *OE*s .

greatest-number computations The $\mathbb{N}$-valued function $hnum\_aux(x{:}OE)$ returns the greatest number that appears in any identifier (in free or bound variables) of x. Using this, we defined the $\mathbb{N}$-valued function $hnum(L{:}OE \text{ List})$, which returns the highest identifier-number that appears in a list of *OE*s. Similar functions on $IDENT$ Lists and other types were defined in straightforward ways, culminating in the $\mathbb{N}$-valued function $big\_ident\_num(E{:}OE, V{:}IDENT \text{ List}, P{:}OE \text{ List})$, which computes the number to use as a lower bound for fresh variables with respect to E, V, P in the substitution $E[V := P]$.

renaming variables *OE*-valued function $rename\_bd\_vars(E, n, rewrites)$ returns a copy of $E{:}OE$ in which all bound variables have been renamed so that their ident-numbers are greater than $n{:}\mathbb{N}$. Its initial call is with $rewrites{:}env(IDENT) = [\,]$, and it performs a recursive descent through E. If it comes across a quantifier subterm $(\forall\langle x,m\rangle \mathrel{|}: P)$, where $\langle x,m\rangle \in \mathit{IDENT}$, it returns $(\forall\langle x,n+1\rangle \mathrel{|}: \mathit{rename\_bd\_vars}(P,n+1,\langle\langle x,m\rangle,\langle x,n+1\rangle\rangle \cdot \mathit{rewrites}))$. The rest of the descent behavior is straightforward, with variables rewritten according to the (perhaps updated) correspondence *rewrites*.

capture permitting substitution *OE*-valued textual substitution function $\mathit{capture\_ok\_subst}(E{:}\mathit{OE}, \mathit{env}{:}\mathit{env}(\mathit{OE}))$ performs a recursive descent on E, returning a copy of E in which each variable whose *IDENT* is on *env* is replaced by the corresponding *OE*, using *lookup_env*. It protects against rewriting bound variables by updating *env* during the descent: if it encounters $(\forall x{:}\mathit{IDENT} \mathrel{|}: P{:}\mathit{OE})$, *env* is updated by $\langle x, \mathit{Ovar}(x)\rangle$; thus, bound variables are left unchanged.

capture-avoiding substitution Using the functions described above, we implemented a general capture-avoiding substitution function on $E{:}\mathit{OE}$, $V{:}\mathit{IDENT}$ List, and $P{:}\mathit{OE}$ List:

$$E[V := P] = \mathit{capture\_ok\_subst}(\mathit{rename\_bd\_vars}(E, N, [\,]), \mathit{env}),$$

where $N = \mathit{big\_ident\_num}(E,V,P)$ and $\mathit{env} = \mathit{zip\_env}(V,P)$. We also implemented a commonly used, restricted variant on $V{:}\mathit{IDENT}$ and $P{:}\mathit{OE}$, using the general case and a singleton list constructor. We also display this variant as $E[V := P]$.

Typically, only one form of an operator is intended to be a user-level construct. This case is an exception: because substitution on lists and substitution on single variables are both quite common operations in the calculational logic of [GS93b], both the list-based and variable-based substitution operators are user-level. This caused extra complications for us as developers —all our user-level tactics pertaining to substitution had to accommodate both forms instead of just one normal form (c.f. type assignment updates)— but it was a convenient way to capture the actual practice of [GS93b].

5.5 Propositions on Object Expressions

From the perspective of modeling calculational logic inference, propositions on *OE*s are close to the heart of the data language. Some are simple and do not require further discussion, such as equality on *OE*s . In this section, we describe our implementation of the other important predicates on *OE*s (and *OE* lists), used to express properties such as whether an object variable occurs free in an object expression, whether an *OE* is an object-level theorem, or whether an object expression is assigned a particular type expression by a given type assignment. Before proceeding, we remind readers of the notion of *user level* introduced in subsection 5.1.1. Many of the definitions that follow are not intended for the user level; they are only auxiliaries in service of the primary definitions. In cases of complex definitions, we explicitly note which are intended for the user level.

5.5.1 List-based propositions

Lists are not quite as important in the Nuprl implementation of the data language as they are in the formalized language in chapter 3. Before, we avoided cluttering our simple type system with function types, so we used lists to represent type expression assignments. Now, we are no longer using a simple type system, we are using Nuprl, so we no longer substitute lists for functions that way.

Lists are still important in other areas: the dummy-variable argument to a quantifier is a list; the arguments to the textual substitution operator may be lists; arguments to the user-level free-variable-occurrence predicate are lists; assumptions (in the context of object-theoremhood judgments) are represented by lists. We use Nuprl's native functions for the cons operation, the empty list, etc. to create terms of list types. Not all the list functions we need, however, were pre-implemented in Nuprl. For instance, we implemented our own function $\mathit{zip}{:}(T1\ \mathrm{List}\times T2\ \mathrm{List}) \rightarrow (T1\times T2)\ \mathrm{List}$; it is similar to function *zip_env* described in the context of environments on page 63, only generalized for any two list arguments.

We also implemented our own list-membership function, *onlist*, parameterized by the list type: $A\ \mathit{onlist}(T)\ L$ holds when T is a Nuprl type, $L{:}T\ \mathrm{List}$, and A is on list L. We used the straightforward recursive definition, which effectively treats L as if it were a set, testing membership without regard for where in the list an element falls. This way, we can reason about collections of assumptions for theoremhood judgments (for example) as if they were sets. (See Figure 6.1 and lemmas (6.8) and (6.9) for examples of the use of *onlist*.) In addition, we defined other list predicates of lesser significance. Predicate *all_elts_diff* holds exactly when all elements on a given *OE* list are different; this is useful when reasoning about lists in the context of textual substitution, where we typically require that, say, all variables on a list for simultaneous replacement be distinct. Predicate *not_on_list* holds exactly when a given element does not appear on a given *OE* list; this is used both as an independent predicate and as a step in the definition of *all_elts_diff*. These are also defined in the straightforward recursive way.

5.5.2 Free variables on the object level

We implemented several different varieties of determiners of free occurrences of object variables. Recall that this is about free occurrences according to the binding structure of our object language, not that of Nuprl.

free variables in expressions Predicate $\mathit{Ofree}(V{:}OV, E{:}OE)$ is "V occurs free in E," defined in the expected way by recursive descent on E. Recall that $\forall_O$ is the only form of OE with any binding variables.

Predicate $\mathit{vsoccurs}(Vs{:}OV\ \mathrm{List}, E{:}OE)$ is "some variable on Vs occurs free in E," recursively applying $\mathit{Ofree}(V, E)$ to all V in Vs. It is used only to avoid a direct, doubly recursive definition for predicate *occurs*.

User-level predicate $\mathit{occurs}(Vs, Es)$ is "some variable on $Vs{:}OV\ \mathrm{List}$ occurs free in some expression on $Es{:}OE\ \mathrm{List}$," implemented by recursively applying $\mathit{vsoccurs}(Vs, E)$ to each E in Es. It is the commonly negated guard against one of a list of variables occurring free in any of a list of expressions. We display it as "(some Vs occur free in Es)."

We implemented several other related definitions that are not intended for users. For instance, predicate *occurs* is recursively defined on the OE List argument; when proving properties of *occurs*, it is sometimes convenient to compute on the OE List argument, and the definition facilitates this. However, it is sometimes more convenient to compute on the OV List argument —perhaps the property to be proved is best expressed recursively on the variables of *occurs*— so we also implemented another version of "(some Vs occur free in Es)" defined recursively on Vs. The details of this definition are irrelevant to users.

alpha-equality on *OE*s We implemented predicate $\mathit{Oalpha\_eq}(t1, t2, corr)$ to determine if OEs $t1$ and $t2$ are equal modulo renaming of bound variables. It is used to implement the change of bound variables clause in the definition of object-level theoremhood, and it is essential for reasoning about theoremhood involving capture-avoiding substitution and quantification (where we must respect equality modulo change of bound variables). In the initial call, $corr{:}(\mathit{IDENT}\times \mathit{IDENT})\ \mathrm{List}$ is $[\,]$; it is updated during the computation of the function. Essentially, $corr$ is maintained as a one-to-one correspondence between IDENT and IDENT; the empty list represents the trivial correspondence of every variable to itself.

We implemented this alpha-equality as a recursive descent through both $t1$ and $t2$, as long as their term-structure is equal; if it is unequal, the terms are not alpha-equal. When descending through quantifier subterms, say $(\forall x1 \mid{:}\ P1)$ in $t1$ and $(\forall x2 \mid{:}\ P2)$ in $t2$, $corr$ is updated by $\langle x1, x2\rangle$. When reaching variables in $t1$ and $t2$, say $v1$ and $v2$, it examines the current $corr$. If neither $v1$ nor $v2$ has been updated on $corr$ —i.e., there are no pairs of either the form $\langle v1, Y\rangle$ or $\langle Z, v2\rangle$ on $corr$— then *Oalpha_eq* requires that $v1 = v2$. If some pair $\langle v1, Y\rangle$ or $\langle Z, v2\rangle$ is on $corr$, then *Oalpha_eq* requires that the most recent update to $corr$ of either form must be $\langle v1, v2\rangle$.

5.5.3 Typing of object expressions

Predicate $Owfdelt(E{:}OE, s{:}OTS, \epsilon{:}OV \rightarrow OTS)$ determines if a single object expression E is "well-formed" on the object level, i.e., is assigned type expression s under type assignment ϵ; its implementation follows the description of type-expression assignment in chapter 3. Built upon this, user-level predicate $Owfd(L{:}OE\ \mathrm{List}, s{:}OTS, \epsilon{:}OV \rightarrow OTS)$ determines if all elements in L have type expression s under ϵ; a singleton list is used for a typing proposition about only one *OE*. In Nuprl, *Owfd* is displayed as $(\epsilon)\ L :: s$.

Note that actual typing is a relation: the empty list can have many types, the domain of individuals D could equal $\mathbb{B}$, and under anticipated extensions to our typing system, such as adding $\mathbb{N}$ and $\mathbb{Z}$, the phenomenon extends. By definition, however, type expression assignments are functions on object variables. Now, consider the object expression $x(x)$. It cannot be assigned a type. Thus we arrive at an interesting observation about our system of type expressions: some expressions permitted under data language syntax cannot be assigned type expressions. Though type expression assignments are total functions on variables, they are partial functions on object expressions.

5.5.4 Object theoremhood

We implemented $Othm(exp, \epsilon, A)$ to capture the determination of object theoremhood — $exp{:}OE$ is an object theorem under assumptions $A{:}OE\ \mathrm{List}$ and type expression assignment ϵ — according to the definition previously given. This is a complex recursive definition in many clauses; each individual clause refers to the theoremhood of expressions, which in turn depends on that clause. For this reason, we do not implement *Othm* directly. We first implement an auxiliary predicate *Othmaux*, using two devices to manage the subtleties: passing recursive calls as arguments to the implemented clauses and parameterizing theoremhood by a straightforward notion of proof depth.

$Othmaux(exp, \epsilon, A, n{:}\mathbb{N})$ is essentially a large disjunction with a guard on it; it holds exactly when the object theoremhood of *exp* under A and ϵ can be "proved" according to this recursive definition, with recursive depth at most

n. (For short, we say *exp* is a "depth n provable" object theorem under A and ϵ.) To illustrate, consider a part of the *Othmaux* definition:

$$
\begin{aligned}
&\mathit{Othmaux}(\mathit{exp}, \epsilon, A, n) == (0 \leq n \ \wedge \\
&\quad [\ldots \\
&\quad \vee (\exists e, f{:}\mathit{OE}.\ \mathit{modus\_ponens\_clause}(\mathit{exp}, f, \\
&\qquad \mathit{Othmaux}(e \Rightarrow f, \epsilon, A, n-1), \\
&\qquad \mathit{Othmaux}(e, \epsilon, A, n-1))) \\
&\quad \vee \ \ldots])
\end{aligned}
$$

(The elided disjuncts in *Othmaux*, above, correspond to the other clauses in the definition of object theoremhood given in chapter 3.) The clause in the theoremhood definition that is present, *modus_ponens_clause*, is separately defined to be:

$$
\begin{aligned}
&\mathit{modus\_ponens\_clause}(\mathit{exp}{:}\mathit{OE}, f{:}\mathit{OE}, \mathit{rec\_call1}{:}\mathbb{P}, \mathit{rec\_call2}{:}\mathbb{P}) == \\
&\quad \mathit{exp} = f \ \wedge \ \mathit{rec\_call1} \ \wedge \ \mathit{rec\_call2}
\end{aligned}
$$

Jointly, these fragments show that by modus ponens, *exp* is a "depth n provable" object theorem under ϵ and A if, for some $e, f{:}\mathit{OE}$, $\mathit{exp} = f$ and both e and $e \Rightarrow f$ are "depth $n-1$ provable" object theorems under ϵ and A. By extending this in the obvious way, implementing the other object theoremhood clauses, we complete *Othmaux*.

With this, $\mathit{Othm}(\mathit{exp}, \epsilon, A)$ is simply $\exists n{:}\mathbb{N}.\mathit{Othmaux}(\mathit{exp}, \epsilon, A, n)$.

Before leaving this discussion of object theoremhood, we briefly note our convention for representing assumptions in calculational logic proofs. Instead of implementing an actual definition of the "assuming$(\in \alpha) a, b$" construct given in the data language in chapter 3 —which basically was just a composition of list-membership and implication (see the data language semantics in section 3.6.3)— we represent assumptions explicitly using the "onlist" predicate and Nuprl implication. That is, to represent that some $P{:}\mathit{OE}$ is an assumption in a statement of object theoremhood $(\epsilon)\ \alpha \vdash Q$, we would say $(P \ \mathit{onlist}\ \alpha) \ \Rightarrow \ (\epsilon)\ \alpha \vdash Q$ (where $\Rightarrow$ is logical implication, not an *OE*-constructor). This is a comfortable practice; a statement that an expression is on an assumption list is treated like any other antecedent proposition, becoming a hypothesis when the proof is carried out. Because of this, and because such assumptions in proofs are used very infrequently in [GS93b], there was no need to implement an explicit, self-contained "assuming" construct.

5.6 Conclusions

5.6.1 Properties of the implemented data language

Our implemented data language generally follows from the content in chapter 3, in which we outlined the standards for our approach to a language for calculational logic. To help ensure that the Nuprl data language was correct —both internally correct and correctly reflecting those initial standards— we proved several properties of our defined expressions. For example, we proved:

- Every *OE* corresponds to the correct number of subterms and binding variables. (E.g., an object-language $\Rightarrow$ term has two subterms and no binding variables, a $\neg$ term has one subterm and no binding variables, etc.)
- The renaming function used in textual substitution preserves the opid of an *OE*, so $E[V := P]$ and E have the same opid.
- Functions *onlist* and *lookup_env* relate correctly: *lookup_env* returns a non-failure value only if the key element is actually on the environment list.
- List predicates such as *onlist* and *all_elts_diff* compute correctly.
- The free variable occurrence predicate behaves as expected on singleton lists of object variables: for $v, x{:}OV$, (some v occur free in x) iff $v = x \in OV$.

These are clearly necessary properties of a correct data language. We proved them all as part of an effort to make sure that our basic definitions were error-free and behaved the way we intended. For many higher-level properties, however, we took a faster approach to avoid direct computation with complicated definitions such as the *OE*-typing or object-theoremhood predicates. Instead of directly computing on the definition each time, we stated desired properties (e.g., the definition of *Owfd* unrolls the way we expect it to on an *OE* implication) as lemmas. Then we established the lemmas once, and we did not need to refer to the definitions directly again; many of the lemmas given in tables in chapter 6 were introduced for this purpose. Citing lemmas in proofs instead of actually unrolling complicated definitions is of great benefit to both the person doing the proof and anyone reading the proof afterward.

To further speed our development, we often used Nuprl's `Fiat` tactic to avoid formally proving these properties in Nuprl; this is essentially equivalent to stating and using a lemma without proof. We were comfortable with this practice when the risk of error in stating a property seemed small — although we employed `Fiat`, we still type-checked each lemma according to

Nuprl typing— and we often used the lemma promptly, where important errors would emerge from practice. This "state-and-use" practice seemed natural and was a source of significant savings to us as developers.

This practice also has other implications. For instance, we essentially did not formally employ the definitions of object-theoremhood and textual substitution in modeling calculational logic inference; we simply stated and used lemmas as we needed them. So, statements such as premises to Leibniz steps and properties of textual substitution on complex object expressions were not proved in Nuprl. (They were quickly analyzed by hand. Recall that our aim is to prototype the method rather than completely ground its application.) If errors exist in our Nuprl formalization of calculational logic, they may well have been introduced through this practice, but we do not believe there are any such errors.

5.6.2 Possible extensions

One of our important design goals was to establish a formalized framework for the calculational approach to discrete mathematics in [GS93b]. We focused our efforts on calculational predicate logic, but extensions to a wider range of topics affected several facets of our implementation. That is, many of our data language definitions may seem unnecessarily general for an account of predicate logic —indeed, they are— but they achieve our goal of ready extendibility: we can extend the mathematics we cover without re-defining many of our essential constructs. Here, we discuss a few example constructs of our data language that are intended to accommodate future extensions.

First, consider the Nuprl type *OE*, upon which so many of our other implemented definitions depend. Because we restricted ourselves to calculational predicate logic, our object-level syntax required only logical operators, and right now, we have essentially only six kinds of expressions in *OE*: object variables, object-level false, object-level implication, etc. If we were to extend our implementation to other areas of mathematics (e.g., arithmetic), we would need to admit other kinds of object expressions (e.g., addition, multiplication). By using a general syntactic structure in *OE*, we could accommodate these added operators without changing its definition (recall the definition of *OE* on page 60). In fact, *OE* is extremely general: it can accommodate object-language terms with any number of subterms/arguments, due to the dependent type construct that lets us define the number of subterms it has in terms of its opid; similarly, it can accommodate general binding structure because dependent types let us abstractly define whether any given subterm marks a variable to be considered bound in the term. Thus, no matter what sort of operator we wish to include in the future, our definition of *OE* can remain unchanged and many facts about it (e.g., the inclusion of *OV* in *OE*, the mutual inclusion of *OE* and the type $IDENT + (i : OPIDS \times (j : \mathbb{N}_{oesbtms(i)} \rightarrow OE \times PossBV(i; j))))$)

need not be restated. We would need only to change auxiliary definitions ($OPIDS$, $PossBV$, etc.) to accommodate the new syntactic elements.

Similarly, our general treatment of quantification could accommodate arithmetic accumulator operations such as summation ($\sum$) and product ($\prod$) without significant alteration. This represents one of the interesting, non-standard observations exploited in [GS93b], that with certain restrictions, accumulators and quantifiers can be treated uniformly. Our uniform treatment permits easy extendibility to common arithmetic accumulators without redefining the fundamental form $(\star_b\ X \mid R : B)$. We would, of course, need to extend the index type *Qind* to accommodate the new forms, and we would need to augment some existing cognate definitions, but the central definition and many facts about it could remain unaltered for the extended object language.

Our treatment of object-level typing also reflects our anticipation of possible extensions. Object expressions currently have only one type under a given type assignment (we stay faithful to the object language semantics given in chapter 3), and for the most part, we could have reasoned about typing using an *OE*-typing function. Instead, we implemented the more general object-typing relation $(\epsilon)\ L :: s$, which permits an expression to be associated with more than one object-level type. As we briefly mentioned before, this formalization would remain robust even if we extended our language to arithmetic and incurred the expected type polymorphisms that arise from having both $\mathbb{N}$ and $\mathbb{Z}$ in a type system. This general typing relation supports a general framework for the calculational approach to discrete mathematics in [GS93b] in a way that a more restrictive typing function would not.

This is not a complete list of ways in which we as developers anticipated extending our system, but it does illustrate the sorts of steps we took to accommodate future extensions. Other such "over-generalized" definitions in our implementation can be similarly justified as part of a more robust system, anticipating applications (e.g., to lists, function symbols, or an extended type system) that we do not describe.

5.6.3 Closing remarks on the data language

When we use the phrase "data language," we refer to the portion of Nuprl that we defined or otherwise shaped to be recognizable as support for the calculational predicate logic of [GS93b]. That is, we consider the data language to include both user-level operations (e.g., *OE*-based textual substitution) and constructs implemented purely to support those operations (e.g., capture-permitting textual substitution). The user level as presented in this chapter should feel familiar to readers; with few exceptions, it corresponds directly to the data language syntax presented in chapter 3. The major exceptions are the inclusion of two kinds of *OE*-based capture-avoiding textual substitution (OV-based and OV list-based) and using functions instead of lists for type assignments (we used lists in chapter 3 only to simplify its semantics). Other useful elements of the data language not mentioned in chapter 3 —such as the type assignment agreement proposition, list-related predicates other than membership, and alpha equality— might be considered part of the user level if users wanted to state and reason about properties not directly covered in [GS93b], but they are not truly on the user level for our current application.

Despite the fact that they are an essential part of our implementation, we tend to consider high-level elements such as Nuprl's definition mechanism as being outside the data language. We can address an illustrative segment of calculational logic to explain theoremhood and inference without involving definitions, and we designed our data language without complicating matters further. Thus, this conception of the data language is convenient for us, but it is not necessary. Indeed, the data language could potentially become all of Nuprl. If we wanted to model the way [GS93b] makes definitions, for instance, we would include a definition mechanism in our data language. Moreover, we would probably use Nuprl's definition mechanism, adopting it whole or with some modification but not building a new one ourselves.[1] In general, if we wished to extend our data language or otherwise innovate in some way, we could simply use Nuprl's established methods and conventions.

Consider one interesting possible extension: we could in principle develop our data language to the point where it could express arguments that establish assertions called "Metatheorems" in [GS93b]. (Recall that our data language is already adequate to express many "metatheorems" that are not labeled as such in [GS93b].) Such an extension is certainly possible. Nuprl can certainly handle such inferences, and our data language is potentially unlimited within the loose confines of Nuprl's capacity to formalize mathematics. From a practical standpoint, the simplicity and flexibility of such extensions is a significant benefit to developing our system in Nuprl and adopting its methods rather than truly starting from scratch.

In general, Nuprl has permitted us to implement a concrete data language that is very similar to the abstract data language blueprint of chapter 3. Of course, there are some differences. Some of them are due simply to the fact that we needed to make our definitions concrete; we can no longer simply say that something exists, we must now say what it actually is. Some are auxiliary constructs that are too unimportant to mention individually; this chapter is not a full account of everything we implemented to support the data language. Others are due to our commitment to Nuprl's semantics; we

[1] If there were some psychological theory about how people make definitions that we wanted to simulate or explore from the perspective of cognitive modeling, we might build a whole new definition mechanism. Other than the mere fact that this is possible in Nuprl, however, this is not a primary concern for us at this stage of development.

gave meanings to data language expressions by defining them in terms of previously defined Nuprl constructs rather than by implementing a direct semantic function (such as *Dsem* in chapter 3). For instance, the logical implication (not the *OE*-constructor) $A \Rightarrow B$ —notated as *if* A *then* B in giving its semantics on page 32— would not have a meaning if B had no meaning; in our implementation, if A were false, $A \Rightarrow B$ would have a meaning whether B had a meaning or not. Such discrepancies are of academic interest, but they do not directly affect users of the system. Indeed, except for some largely irrelevant display form issues, we have previously mentioned all the user-level differences. Our concrete data language is a good representation of the abstract one presented in chapter 3 and, as we mentioned then, that data language is a good representation of the language used to express calculational predicate logic in [GS93b].

Chapter 6

Implementing Calculational Logic Inference

6.1 Introduction

In formalizing and implementing a data language for calculational logic, we exposed many details that were not clear from the textbook [GS93b]. The same is true for calculational logic inference: in this chapter we describe our implementation of a formalized, tactic-based explanation of the fundamental inferences of calculational logic,[1] and once again, we expose many details that were not apparent in [GS93b].

Gries and Schneider intended [GS93b] to be a college-level textbook, so it is no surprise that they presented a simplified version of actual calculational logic inference. They omit significant material, correctly presuming that their readers will be able to fill in those gaps that arise in applications. Nuprl has no such gap-filling capacity, so we must make the essential propositions explicit, even ones that may seem too obvious to mention in a textbook. In addition, they provide few or no explicit instructions for reasoning about free variable occurrence, typing, and many other calculational logic inferences. They emphasize the process of Leibniz inference, making sure students are comfortable with the machinery of that most-important inference form, but in general they let people figure out on their own what inferences are necessary. Once again, Nuprl has no innate capacity to figure out such things, so we fill in the details in our tactic implementation of calculational inference.

Since Leibniz is the central inference form of calculational logic, a corresponding Leibniz tactic, `Leib_tac`, is the central tactic in our implementation; all our other tactics and auxiliary ML procedures are in some (perhaps indirect) way performing some component sub-task of Leibniz inference. Taken

[1] By "inferences," we do not mean only the patterns described by inference rules; instead, we refer to the general logical/mathematical inferences people make in carrying out calculational logic proofs.

together, our tactics and ML programs are our explanation of calculational logic inference.

Above, we mentioned that the explanation in [GS93b] was somehow simplified and incomplete. Our tactic implementation is also simplified and incomplete, but in a different way. It is a thorough, formalized account of the fundamental inferences with all the components explicitly included, but it is based on heuristics, and it does not work in all the cases that may come up in the book. In that way, it is like a possible description of the cognitive inferences of a newcomer to the calculational approach, one who knows only a few of the simplest methods and may well be stumped by some of the problems in [GS93b]. Just as a novice could be taught, our tactics could be extended, and we discuss some of those possible extensions in the course of this chapter. Even our simplified version of calculational inference, however, can handle many of the cases that arise in practice.

6.2 Representing Leibniz Inference in Nuprl

We once again motivate our technical development by considering the proof of Theorem Change of Dummy from [GS93b, page 151], a chain of seven equalities. (It is presented verbatim in Figure 3.1 of chapter 3.) Each equality is established by an instance of one of the three related inference rules called Leibniz; such Leibniz steps are the basis for all calculational logic proofs. The remainder of the proof is simply applications of inference rule Transitivity, inferences that are substantially more straightforward than typical Leibniz steps.

Our main goal in this chapter is to describe Leibniz inference, formalizing and implementing it to expose details not apparent from the presentation in [GS93b]. Consider the first Leibniz step in the proof of Theorem Change of Dummy, presented here exactly as in [GS93b]:

$$
\begin{array}{lll}
(6.1) & & (\star y \mid R[x := f.y] : P[x := f.y]) \\
& = & \langle \text{One-point rule (8.14)} \\
& & \text{—Quantification over } x \text{ has to be introduced. The One-} \\
& & \text{point rule is the } \textit{only} \text{ theorem that can be applied at first.}\rangle \\
& & (\star y \mid R[x := f.y] : (\star x \mid x = f.y : P))
\end{array}
$$

(We chose this as an example simply because it was the first step; we had no mathematically significant reason to choose it.) It encodes the inference that expression $(\star y \mid R[x := f.y] : P[x := f.y]) = (\star y \mid R[x := f.y] : (\star x \mid x = f.y : P))$ is an object-level theorem, the conclusion of one of the three "substitution of equals for equals" inference rules named "Leibniz" in [GS93b]. The name "One-point rule" in angle brackets refers to the *premise* *theorem*, an instance of which is the premise of that instance of Leibniz. In [GS93b], this premise theorem is given as:

(6.2) **One-point rule:** $(\star x \mid x = E : P) \ = \ P[x := E]$

This is the book-style presentation of One-point rule. Our formalization of it (see chapter 3, page 35) contains significant details that are not present in the book. In general, our explanation of Leibniz inference will be in terms of our detailed formalizations, not the versions in [GS93b].

In Nuprl, we formally implemented a Leibniz inference tactic `Leib_tac` such that a single call of `Leib_tac` corresponds to a single Leibniz step in [GS93b]. The user-supplied arguments of `Leib_tac` are essentially the same as the information explicitly given in a "hint" (in angle brackets) in a book-style Leibniz step. As seen in chapter 2, in general, a hint consists of some reference to the premise theorem —perhaps its name, perhaps a statement of the theorem expression itself— and information about how to instantiate the premise theorem for the present instance of Leibniz. Therefore, `Leib_tac` takes two arguments: a reference to the premise theorem and a list that provides information on how to instantiate it. Figure 6.1 shows our sequent-style representation of the inference in example (6.1).

In Figure 6.1, the Leibniz application tactic is `Leib_tac`, the indented expression beginning with `Thm*` refers to the premise theorem One-point rule, and the list of terms that follows (within the `Tms:[...]` construct) is the instantiation information. (The little `...w` that follows represents the application of Nuprl's `Auto` tactic to handle Nuprl `wf` subgoals; it is part of the standard Nuprl utilities for inference.) The instantiation list contains one expression for each variable of the One-point rule, given in the order in which the variables are quantified. Guided by it, the instantiated One-point rule is sufficient to prove:

(6.3)
```
∀x,y,f,f⁻:OV, α:OE List, R,P:OE, ∈:OV⟶OTS, b:Qind.
(∀ [x; y] |t_o:x = f(y) ≡ y = f⁻(x)) onlist(OE) α &
(∈[[x]⟵ ind]) [R; P] :: bool &
(∈) α :: bool &
¬(some [y] occur free in [R; P]) &
¬x = y &
¬(some [x; y] occur free in α) &
¬x = f &
(∈[[x; y]⟵ ind]) [f; f⁻] :: ind(1)
⇒
(∈[[y]⟵ ind]) α ⊢ (*_b [x] | x = f(y) : P)
                     =
                     (P)[x := f(y)]
```

This is the premise needed for this particular Leibniz inference.

```
1. x : OV
2. y : OV
3. f : OV
4. f^- : OV
5. α : OE List
6. R : OE
7. P : OE
8. ∈ : OV ⟶ OTS
9. b : Qind
10. (∀ [x; y] |t_o:x = f(y) ≡ y = f^-(x)) onlist(OE) α
11. (∈[[x] ⟵ ind]) [R; P] :: bool
12. (∈) α :: bool
13. ¬(some [y] occur free in [R; P])
14. ¬x = y
15. ¬(some [x; y] occur free in α)
16. ¬x = f
17. (∈[[x; y] ⟵ ind]) [f; f^-] :: ind(1)
⊢  (∈) α ⊢ (*_b [y] | (R)[x := f(y)] : (P)[x := f(y)])
            =
            (*_b [y] | (R)[x := f(y)] :
                 (*_b [x] | x = f(y) : P))
by  Leib_tac
    Thm* ∀x:OV, E:OE, α:OE List, ∈:OV ⟶ OTS, b:Qind, P:OE.
         ¬(some [x] occur free in [E]) &
         (∈) α :: bool & (∈) [E] :: ind &
         (∈[x ⟵ ind]) [P] :: bool
         ⇒
         (∈) α ⊢ (*_b [x] | x = E : P) = (P)[x := E]
    Tms:[x ; f(y) ; α ; ∈[[y] ⟵ ind] ; b ; P] ...w
```

Figure 6.1: Sequent-style inference using `Leib_tac`. The theorem argument to `Leib_tac` is given here in its long form, showing its full content. In practice, we can use a more concise display, just showing the theorem name.

This is a brief example of how Nuprl's representation of Leibniz inference relates to the presentation of Leibniz in [GS93b]. In general terms, it shows that our Leibniz tactic uses previously proved object-theoremhood judgments to establish unproved object-level theorems. As suggested by the hypotheses of the sequent in Figure 6.1 and the antecedent of the formalized One-point rule, however, Leibniz inference involves more than just object-theoremhood propositions. In the course of a single Leibniz step, several important kinds of propositions that occur in the data language expressions above —including OV-inequalities, statements about variables not occurring free in object expressions, and OE-typing— frequently affect the inference, often becoming intermediate proof goals that our tactic must automatically solve. In the following sections, we discuss our methods for manipulating and making inferences about these propositions before describing how we combined these methods in the overall Leibniz tactic.

6.3 Inferring That Two *OV*s Are Not Equal

Although non-negated OV-equality propositions are rare in our account of the calculational logic in [GS93b], OV-inequalities are extremely common. Recall that, because we interpret [GS93b] as using a formalized metalanguage instead of a typical object language, OV expressions stand for object variables but are not object variables themselves.

The conventions in [GS93b] involving OV-inequalities are somewhat inconsistent. People working with the book realize that different letters can refer to the same variable or expression when matching theorems; for instance, it is clear that the expression for Theorem Symmetry of Equivalence $p \equiv q \equiv q \equiv p$ can be used to justify the theoremhood of $A \equiv A \equiv A \equiv A$ for any $A{:}OE$. Typically, however, when not in the context of pattern-matching or instantiating expressions, people working with [GS93b] will also assume that different letters refer to different variables when it becomes relevant. For instance, after instantiating Theorem Symmetry of Equality to get $A \equiv B \equiv B \equiv A$ (for $A, B{:}OV$), someone working with calculational logic would conventionally presume that $\neg(A = B \in OV)$,[2] even though the fact that A and B are not identical letters is not logically sufficient for that conclusion.

Therefore, to reason about OV-inequalities, we cannot simply rely on convention. Further, since there is not typically an environment that gives explicit values to the Nuprl variables $P, Q{:}OV$, we cannot typically compute from a proposition $\neg(P = Q \in OV)$ whether that proposition is true. So, we must reason from hypotheses. In our implementation, we insist that users explicitly include the hypotheses that certain OVs are unequal, even though such

[2]The user probably "knows" $\neg(A = B \in OV)$ before instantiation, even if it is unproved, and carries that knowledge after the instantiation as well.

propositions are frequently omitted in [GS93b].[3]

6.3.1 Methods used to prove OV-inequality judgments

For this first, heuristic implementation of calculational logic inference, we implemented only two methods for solving OV-inequalities. We do not mean to suggest that other approaches would be uninteresting or ineffective, but the design issues in extending the current account quickly become complicated. (For example, we could try to reason from type expressions, but the simplest approach —that OVs having different type expressions stand for different object variables— would not work if we added $\mathbb{N}$ and $\mathbb{Z}$ to our type expressions.) In addition, it is not clear how much such extensions would add to the inferential power or descriptive value of the tactics we have already implemented. We discuss these issues further in section 6.3.2.

Our tactic for solving OV-inequalities is called `solve_noteqOV`. On a proof goal $\neg(P = Q \in OV)$, it does the following:

- If $\neg(P = Q \in OV)$, $\neg(Q = P \in OV)$, $\neg(\text{some } P \text{ occur free in } Q)$, or $\neg(\text{some } Q \text{ occur free in } P)$ is a hypothesis, it proves the goal from hypotheses. (Other tactics, which we describe later, analyze more complex hypotheses into the above forms before `solve_noteqOV` is invoked.)
- If not, it fails.

Note that for $P, Q{:}OV$, $\neg(\text{some } P \text{ occur free in } Q)$ implies $\neg(P = Q \in OV)$. Thus, the tactic also accounts for symmetry of OV-equality in its methods.

6.3.2 Discussion

OV-inequalities are pervasive in calculational logic; they are relevant to almost every important inference. Therefore, despite the simplicity of `solve_noteqOV`, it is not surprising that many design issues in OV-inequality reasoning apply to our implementation in general. For instance, our tactics must account for the fact that the variables on which they act are actually metavariables. As developers, we must also decide how much information we require a user to provide to the system —e.g., we require users to explicitly give OV-inequality hypotheses rather than trying to infer some of them with tactics— and how we explain our heuristics in light of the overall goal of implementing Leibniz inference. Many issues that complicate all areas of our tactic development are present to some extent in this seemingly simple inference.

[3] Even for us as developers, one of our most common errors in formalizing of the theorems of [GS93b] is the omission of such information. The automated exposure of such errors is one of the virtues of our system.

Our design for OV-inequality reasoning also reflects the heuristic nature of our overall Leibniz implementation; it is not intended to be exhaustive, and there are many possible extensions to nearly every aspect of it. For instance, one possible extension to `solve_noteqOV` would be to look for hypotheses of the form $\neg(X = Y \in OE)$ where X and Y are also declared to be of type OV in the hypotheses. With the subtyping in our calculational logic data language, we could conclude $\neg(\text{some } X \text{ occur free in } Y)$ from these hypotheses. OE-inequality hypotheses are not generally a part of calculational logic, however, and such an extension might never become useful.

The simple tactic `solve_noteqOV` is adequate for many examples, given certain conventions governing what propositions must be supplied in the antecedent of an object-theoremhood statement. (Propositions in the antecedent become hypotheses in the proof.) Thus, in the division of labor between tactics and users of the calculational inference system, our design places significant responsibility for expressing OV-inequalities on the user rather than employing more powerful tactics for inferring OV-inequalities.

Indeed, readers may notice that the form of the first step of the Change of Dummy proof given in this chapter in example (6.1) is not the same as in chapter 3 (page 17). In this chapter, we explicitly provide some hypotheses (i.e., propositions in the antecedent) stating that certain OVs are not equal. In chapter 3, we did not specify that such information was necessary —from that theoretical perspective, it seemed possible that more sophisticated heuristics than the ones we implemented in `solve_noteqOV` might infer those OV-inequalities. For our initial account of calculational logic inference, however, it is reasonable to require the user to explicitly supply such information in theorem statements, as we soon explain.

Some necessary OV-inequalities —e.g., hypothesis 14 in our example inference in Figure 6.1— could not be inferred from other given information, even with sophisticated inference methods. We therefore require users to supply all necessary OV-inequality hypotheses, even if some would be unnecessary with a more sophisticated tactic `solve_noteqOV`.[4] Without this requirement, users would need to know before entering their data —before receiving automated support for the problem at hand— which OV-inequalities were necessary to explicitly provide and which would be inferred by the system. Thus, the analysis required to support a stronger OV-inequality tactic could become difficult, and designing more sophisticated tactics would have required a significantly deeper cognitive study. Since users presumably know what OV-inequalities are intended, the requirement that users specify all necessary OV-inequalities adds little cost to the user and avoids added technical complexity for both user and developer. In light of our primary goal —implementing a prototype

[4]The simplicity of our tactic is not the only factor that requires user-supplied OV-inequality hypotheses. If that requirement could be completely automated away, we would see our imposing it as a significant design flaw, making too many demands on users.

cognitive model of calculational logic inference— it seems a sensible decision.

Other complications would also emerge if we had decided to strengthen `solve_noteqOV`. Since we have been mindful of possible extensions of our calculational logic language throughout this project, we would also wish to accommodate those in strengthening `solve_noteqOV`, and this could prove difficult with respect to an extended typing system. There are several interesting and reasonable ways in which one might extend the typing system in our calculational logic implementation, such as introducing new types and subtyping relationships ($\mathbb{N}$ and $\mathbb{Z}$, for instance) or introducing complex structures that would make type inference undecidable. Incorporating heuristics to deal with all the complications arising from such extensions is beyond the scope of this project.

Even within the current language, difficulties can emerge in unusual cases. For instance, it is tempting to use information about type signatures in our heuristics. We know that signature bool refers to type $\mathbb{B}$, but $\mathbb{B}$ is also a permissible domain of the predicate logic; we cannot conclude that $\neg(P = Q \in OV)$ from $(\epsilon)\ P :: \ \mathrm{bool}$ and $(\epsilon)\ Q :: \ \mathrm{ind}$, which is a significant restriction on how we might incorporate type signature information into OV-inequality reasoning. It gets worse. The signature bool refers to the expected two-element type $\mathbb{B}$. Does $\mathrm{bool} \times \mathrm{bool} = \mathrm{bool} \rightarrow \mathrm{bool}$? Might either of those equal a signature for the integer subtype $[0..3]$, if we extended our type system to account for $\mathbb{N}$? It was particularly tempting to include heuristics for inferring OV-inequality from type signature-inequalities in our implementation, because we believe that people use such techniques when working with [GS93b]. It was not necessary to do so to achieve our goals, however, and we decided to avoid such complications.

6.4 Inferring That *OV*s Do Not Occur Free in *OE*s

Propositions of the form $\neg(\text{some } P \text{ occur free in } Q)$ —we call them *not-occur* propositions for short— are fundamental to calculational logic, exactly the kind of syntactic property that it is designed to accommodate. Unlike the other propositions we discuss before describing our Leibniz tactic —OV-inequalities, OE-typing judgments, and object-theoremhood judgments— not-occur propositions are explicit in the statements of theoremhood in [GS93b].

There are strong connections between not-occur propositions and OV-inequalities. As with OV-inequalities, elements of not-occur propositions are typically metavariables, so we cannot directly compute the truth value of a proposition from its form. As with OV-inequalities, we must reason from hypotheses. Not-occur propositions, however, cause somewhat less confusion: it

Table 6.1: Lemmas for list-decomposition of not-occur propositions

(6.4)
```
∀X:OV, Y:OV List, alpha:OE List.
¬(some X.Y occur free in alpha)
⟺
¬(some [X] occur free in alpha) &
¬(some Y occur free in alpha)
```

(6.5)
```
∀X:OV, Y:OV List, E:OE.
¬(some X.Y occur free in [E])
⟺
¬(some [X] occur free in [E]) &
¬(some Y occur free in [E])
```

(6.6)
```
∀X:OV List, E:OE, Y:OE List.
¬(some X occur free in E.Y)
⟺
¬(some X occur free in [E]) & ¬(some X occur free in Y)
```

is tempting to interpret P and Q as object variable literals in the notation $\neg(P = Q \in OV)$ but perhaps less tempting to make that interpretation in the notation $\neg(\text{some } P \text{ occur free in } Q)$. We usually talk about variables occurring free in expressions, so it is easy to believe that Q is intended to range over expressions.

The arguments to not-occur propositions generally have more complex structure than OVs, so reasoning about not-occur propositions is more complex than reasoning about OV-inequalities. Bluntly comparing proof goals to hypotheses is not sufficient. By decomposing not-occur propositions on the list-structure and OE-structure of their arguments before comparing hypotheses and proof goals, our tactics are capable of performing all the inferences needed for a first implementation of Leibniz inference.

6.4.1 Decomposition of not-occur propositions

The decomposition of propositions on the structure of their arguments is essential to our calculational logic tactics. We use this approach on both hypotheses and conclusions, and it is as important to OE-typing judgments as to not-occur propositions. As an introduction, we discuss it here in the context of not-occur propositions; the main ideas also apply elsewhere.

The fundamentals of decomposing a not-occur proposition are the same for hypotheses and proof goals. If a complex not-occur proposition is a hypothesis, we analyze it into several separate hypotheses that state all the atomic propositions it entails by decomposition (using the lemmas in Tables 6.1 and 6.2).

Table 6.2: Lemmas for *OE*-structure-decomposition of not-occur propositions

(6.7) ∀X:OV List, P,Q:OE.
¬(some X occur free in [P ≡ Q]) ⟺
¬(some X occur free in [P]) & ¬(some X occur free in [Q])

(Similarly for *OE*-constructors =, ∧ , ∨ , ⇒ , ¬ and function application P(Q)).

Recall the unified treatment of quantification. Hence, lemmas (6.8) and (6.9) apply to both existential and universal quantifiers. Lemma (6.8) is for decomposing propositions in proof goals, while lemma (6.9) is used for decomposing hypotheses.

```
(6.8) ∀V:OV, b:Qind, Y:OV List, R,B:OE.
      (¬V onlist(OV) Y ⇒
       ¬(some [V] occur free in [R]) &
       ¬(some [V] occur free in [B]))
      ⟺
      ¬(some [V] occur free in [(*_b Y | R : B)])
```

```
(6.9) ∀V:OV, b:Qind, Y:OV List, R,B:OE.
      V onlist(OV) Y ∨ ¬V onlist(OV) Y &
      ¬(some [V] occur free in [R]) &
      ¬(some [V] occur free in [B])
      ⟺
      ¬(some [V] occur free in [(*_b Y | R : B)])
```

```
(6.10)∀X:OV List, b:Qind, P,Q:OE.
      ¬(some X occur free in [P *_b Q])
      ⟺
      ¬(some X occur free in [P]) & ¬(some X occur free in [Q])
```

```
(6.11)∀X:OV List, b:Qind. ¬(some X occur free in [u_b])
```

(Similarly for *OE* constants t_o, f_o.)

```
(6.12) 1. A : OV
       2. B : OV
       3. P : OE
       4. Q : OE
       5. R : OE
       ⊢  ¬(some [A; B] occur free in [P; Q ⇒ R])

        by  Not_OccCD ...w

        \
         ⊢  ¬(some [A] occur free in [P])  by  <TACTIC>
         ---
         ⊢  ¬(some [B] occur free in [P])  by  <TACTIC>
         ---
         ⊢  ¬(some [A] occur free in [Q])  by  <TACTIC>
         ---
         ⊢  ¬(some [A] occur free in [R])  by  <TACTIC>
         ---
         ⊢  ¬(some [B] occur free in [Q])  by  <TACTIC>
         ---
         ⊢  ¬(some [B] occur free in [R])  by  <TACTIC>
```

Figure 6.2: Example decomposition of not-occur propositions

Similarly, if a complex not-occur proposition is a conclusion, we can solve it by decomposing it into atomic propositions and solving each of those individually. (We discuss our tactics for solving not-occur proof goals in section 6.4.2.)

We wrote two tactics to decompose complex not-occur propositions into atomic component not-occur propositions: `Not_OccHD` analyzes hypotheses; `Not_OccCD` analyzes conclusions. For an example, see (6.12) in Figure 6.2 (page 88), which demonstrates `Not_OccCD` decomposing a not-occur proposition in a conclusion; as we briefly elaborate later, `Not_OccHD` would have acted analogously on a hypothesis by adding new hypotheses. By decomposing on both list structure (of both arguments) and *OE*-structure of complex expressions, it analyzed the conclusion into not-occur subgoals whose arguments had no structure supporting further decomposition. The lemmas used for such decompositions are given in Tables 6.1 and 6.2.

Some complexities arise when decomposing quantifiers. Lemma (6.8) in Table 6.2 is used to analyze both existential and universal quantifier forms in proof goals, resulting in a new *onlist* hypothesis. For an example, see (6.13) in Figure 6.3. Relatedly, analyzing hypotheses using lemma (6.9) results in a case split with a new *onlist* hypothesis on each branch. (The case split is on the disjunction in the antecedent of the lemma. The proposition `V onlist(OV) Y`

```
(6.13) 1. A : OV
       2. P : OE
       3. X : OV
       4. Y : OV
       5. Q : OE
       6. R : OE
       ⊢  ¬(some [A] occur free in [(∀ [X; Y] | P : Q ∧ R)])

         by  Not_OccCD ...w

         \
          7. ¬A onlist(OV) [X; Y]
          ⊢  ¬(some [A] occur free in [P])  by  <TACTIC>
          ---
          7. ¬A onlist(OV) [X; Y]
          ⊢  ¬(some [A] occur free in [Q])  by  <TACTIC>
          ---
          7. ¬A onlist(OV) [X; Y]
          ⊢  ¬(some [A] occur free in [R])  by  <TACTIC>
```

```
(6.14) 1. A : OV
       2. P : OE
       3. X : OV
       4. Y : OV
       5. Q : OE
       6. R : OE
       7. ¬(some [A] occur free in [(∀ [X; Y] | P : Q ∧ R)])
       ⊢  False  by  NotOccHD 1 ...w

         \
          8. A onlist(OV) [X; Y]
          ⊢   by  <TACTIC>
          ---
          8. ¬A onlist(OV) [X; Y]
          9. ¬(some [A] occur free in [P])
          10. ¬(some [A] occur free in [Q ∧ R])
          11. ¬(some [A] occur free in [Q])
          12. ¬(some [A] occur free in [R])
          ⊢   by  <TACTIC>
```

Figure 6.3: Decomposition of not-occur propositions on quantifiers. Example (6.13) illustrates `Not_OccCD` on proof goals; example (6.14) illustrates `Not_OccHD` on hypotheses. In both cases, new *onlist* hypotheses are added.

is one disjunct; the other disjunction, which is a conjunction, also includes an *onlist* proposition.) For an example, see (6.14) in Figure 6.3.

The new *onlist* hypothesis on the branch in which the not-occur proposition is not decomposed typically results in obvious contradictions with other hypotheses that arise in the proof. For example:

(6.15)
```
1. x : OV
2. y : OV
3. a : OE
4. b : OE
5. ¬(some [x] occur free in [(∀ [y] |:a ∧ b)])
⊢  ¬(some [x] occur free in [(∀ [y] |:b ∧ a)])

 by  NotOccHD 1 ...w

 \
  6. x onlist(OV) [y]
  ⊢   by  Not_OccCD ...w

   \
    7. ¬x onlist(OV) [y]
    ⊢  ¬(some [x] occur free in [b])  by  Trivial
    ---
    7. ¬x onlist(OV) [y]
    ⊢  ¬(some [x] occur free in [a])  by  Trivial
  ---
  6. ¬x onlist(OV) [y]
  7. ¬(some [x] occur free in [a ∧ b])
  8. ¬(some [x] occur free in [a])
  9. ¬(some [x] occur free in [b])
  ⊢   by  <TACTIC>
```

The calls to tactic `Trivial` prove the contradiction between hypotheses 6 and 7.

Because of pragmatic differences between analyzing hypotheses and proof-goals, we have both `Not_OccCD` and `Not_OccHD` as top-level decomposition tactics. The recursive `Not_OccHD` tactic works as follows:

- Each call of `Not_OccHD` is on some hypothesis i; the initial call is on $i = 1$. If i is greater than the number of hypotheses in the proof, terminate the recursion by calling Nuprl's identity tactic `Id`.
- If hypothesis i is a not-occur proposition with structure that can be decomposed, add the results of the decomposition step to the end of the hypothesis list. For instance, on hypothesis $\neg$(some $[x]$ occur free in $[(P \wedge$

$R) \;\Rightarrow\; Q]$, the propositions $\neg$(some $[x]$ occur free in $[P \wedge R]$) and $\neg$(some $[x]$ occur free in $[Q]$) would be added at the end of the hypothesis list.

- Recursively call `Not_OccHD` on hypothesis $i+1$. This way, propositions added to the end of the hypothesis list will eventually be analyzed by a later recursive call.

Note that `Not_OccHD` adds the results of decomposition steps to the end of the hypothesis list without overwriting or thinning intermediate hypotheses. This increases the possibility that we could solve a proof goal directly from hypotheses before decomposing it with `Not_OccCD`. (As described in section 6.5, we use a similar add-to-end process for hypothesis decomposition of *OE*-typing propositions, as well.)

The recursive `Not_OccCD` tactic works similarly:

- If the current proof goal is a not-occur proposition with structure that can be decomposed, perform a decomposition step. All list-structure decomposition must be completed before *OE*-structure decomposition begins.
- Each decomposition step results in some number of new not-occur proof subgoals. Recursively call `Not_OccCD` on each such subgoal until no further decomposition is possible.

Except for a few complexities that arise in list decomposition (which we discuss later in section 6.4.3), both `Not_OccHD` and `Not_OccCD` are simple, straightforward implementations of the decomposition procedure described above.

6.4.2 Methods for solving

As previously mentioned, we cannot generally solve not-occur proof goals by direct computation, we must reason from hypotheses. For this reason, decomposing expressions is the biggest part of solving not-occur proof goals: essentially, we simply decompose the proof goal and hypotheses into their atomic propositions and perform some simple inferences to finish the proof. We encoded some of those auxiliary inferences in tactic `solve_not_occ_via_noteqOV`: on a not-occur proof goal $\neg$(some V occur free in X), it checks if there is a hypothesis $\neg(V = X \in OV)$ or $\neg(X = V \in OV)$; if so, it proves the goal from hypotheses; if not, it fails. In some ways, it is the inverse of some of the reasoning encapsulated in tactic `solve_noteqOV` for solving OV-inequality subgoals.

Tactic `solve_not_occ_concl` for solving not-occur proof goals works as follows:

- Call `Not_OccHD` on hypothesis 1.
- Try to prove the goal from hypotheses; if that fails, call `Not_OccCD`.
- On each goal produced by `Not_OccCD`: Try to prove the goal from hypotheses; if that fails, call `solve_not_occ_via_noteqOV`.

To prevent our tactics from diverging, we wrote the tactic code to invoke `solve_not_occ_via_noteqOV` only on an OV-inequality hypothesis that will solve the proof goal. That way, there is no loop of trying to prove not-occur goals using OV-inequalities, which we might in turn try to prove using not-occur goals, etc.

Essentially, `solve_not_occ_concl` simply matches the proof goal against not-occur and OV-inequality proposition hypotheses. The thorough not-occur decomposition tactics enable such a straightforward approach.

6.4.3 Discussion

Lists

The major complication in tactics `Not_OccCD` and `Not_OccHD` comes in the list-structure decomposition steps. Because we wanted to reason directly about both the $OV\,\text{List}$ and $OE\,\text{List}$ arguments of a not-occur predicate, we implemented two different, equivalent recursive definitions of $\neg(\text{some } L1 \text{ occur free in } L2)$; one permitted direct manipulation of $L1$, the other was directly in terms of $L2$. We then proved lemmas asserting the equivalence of the two forms. All this is hidden from users of the system, but as developers we needed to take care with list decomposition.

We also needed to account for meta-levels and multiple representations of lists. Users think of $L1$ and $L2$ in $\neg(\text{some } L1 \text{ occur free in } L2)$ as lists, and they do represent lists on the object-language level. To Nuprl, they are ordinary terms (of types $OV\,\text{List}$ and $OE\,\text{List}$, respectively), not lists of terms. In fact, they may not have any explicit list structure to them at all. They may simply be variables declared to have a list type, not terms with opid `cons` (the expected list-forming operator for Nuprl terms). Therefore, we cannot simply stop list-decomposition when both list arguments have the structure of singleton lists (i.e., a single element cons'd with the nil list). If a list argument is a Nuprl variable, it cannot be further decomposed even though it is not a singleton list (and may not stand for one). In tactics `Not_OccCD` and `Not_OccHD`, we decompose only on the structure visible to Nuprl, not on object-level structure not explicit in our data-language expressions.

Similar issues involving lists also appear in other contexts, such as the list arguments to the object-expression typing predicate and the construct that represents updates to type expression assignment functions. We handled these cases analogously to this not-occur case, and we will not mention it in other contexts.

Possible extensions

The most natural extension we could make to tactic `solve_not_occ_concl` would be to extend the decomposition tactics to accommodate the textual substitution operators of calculational logic. It was not necessary to implement such decomposition steps for our test cases, so we avoided the more complicated tactics that this would require. We do not, however, foresee conceptual difficulties in making such an extension in the future.

Despite the simplicity of `solve_not_occ_concl`, it is not clear that there are other extensions to it worth making. The fact that the arguments of not-occur propositions typically contain only metavariables means that direct computation of the truth values of not-occur propositions is not a strategy worth pursuing. This, in turn, significantly limits our options for extensions. Because $\neg(\text{some } P \text{ occur free in } Q)$ reduces to $\neg(P = Q \in OV)$ in cases where $P, Q \in OV$, the adequacy of our not-occur inference methods follows to some extent from the adequacy of our OV-inequality inference methods. Most not-occur inference improvements would likely come solely from improvements to tactic `solve_noteqOV`, which we discussed in section 6.3.

6.5 Type Inference on *OE*s

Two kinds of typing affect our Nuprl implementation of calculational logic: Nuprl typing and object expression typing. Nuprl typing is the standard association of Nuprl types with Nuprl variables, such as a Nuprl variable P having type *OE*, the type of object expressions described in chapter 5. In addition, the object expression typing described in chapter 3 also applies; expressions with Nuprl types *OE* or OV have an object-level type (more precisely, a type expression) under a type assignment ϵ in a model.

For the most part, reasoning about Nuprl typing is handled automatically by Nuprl in well-formedness subgoals (recall the `...w` in the Nuprl inference shown in example (6.1) in section 6.2). It affects our calculational system primarily by permitting us to check the type-correctness of the lemmas and definitions we implement. Our tactics do not attempt to solve well-formedness subgoals. We presume that our calculational logic inference tactics will be invoked inside the Nuprl `...w` wrapper, which solves typical well-formedness subgoals. Unproven well-formedness subgoals are frequently evidence of user error, such as declaring a variable to be of type *OE* instead of type *OE* List; it

Table 6.3: Lemmas for decomposition of *OE*-typing hypotheses

(6.16) `∀expn:OE, ∈:OV→OTS, L:OE List.`
`(∈) L ⊢ expn ⇒ (∈) L :: bool`

(6.17) `∀L:OE List, A:OE, sig:OTS, ∈:OV→OTS.`
`(∈) [A] :: sig & (∈) L :: sig ⟺ (∈) A.L :: sig`

(6.18) `∀Es:OE List, sig:OTS, ∈:OV→OTS, E:OE.`
`(∈) Es :: sig & E onlist(OE) Es ⇒ (∈) [E] :: sig`

would be a disservice to users if our tactics obscured such errors. We do not discuss Nuprl-typing further in this chapter.

Object expression typing is not native to Nuprl, so we implemented tactics expressly to perform *OE* typing inference under type assignments. *OE* typing is pervasive and essential to calculational logic inference: e.g., an *OE* must be a boolean to be a theorem; both arguments to an equality must have the same type for the equality to have a value; and for the result of a textual substitution $E[V := P]$ to have a type, there are constraints on how the types of V and P may be related. For our first heuristic account of calculational logic, we implemented a few heuristics that effectively perform typical *OE*-typing inferences.

6.5.1 Decomposing *OE*-typing hypotheses

We implemented tactic `OwfdHD` to derive *OE*-typing information from hypotheses. Like `Not_OccHD` for not-occur propositions, it performs essential preprocessing to support *OE*-typing inferences. It is quite simple; see Table 6.3 for the lemmas that correspond to `OwfdHD` decomposition steps. `OwfdHD` works as follows:

- First, it uses lemma (6.16) to draw *OE*-typing information from *OE*-theoremhood hypotheses. After this step, all *OE*-typing information is represented by propositions of the form $(\epsilon)\ L :: \tau$, where L:*OE* List, although some of those propositions may be further decomposable.
- Using lemma (6.17), `OwfdHD` decomposes *OE*-typing hypotheses on the structure of the *OE* List argument.
- Using lemma (6.18), `OwfdHD` derives all possible *OE*-typing information from matching *OE*-typing hypotheses and hypotheses of the form E *onlist*(*OE*) Es.

We decided not to decompose hypotheses on the structure of their *OE* arguments. This puts only a slightly increased demand on system users — instead of giving the types of compound expressions, they must give the types of individual variables, as in example (6.1) in section 6.2— and the practice seems natural and unobtrusive.

6.5.2 Guessing *OE* types

Sometimes, solving an *OE*-typing proof goal $(\epsilon)\ E1 :: \text{sig}$ requires instantiating a lemma with the type of another $E2{:}OE$. Rather than exhaustively testing every possible type for $E2$, our tactics use a cheap heuristic method to *guess* a type for $E2$. ML procedure `OE_sig_guess` performs the *OE* type guessing: (`OE_sig_guess` $E\ \ \epsilon\ \ H$) guesses the type of expression E under local type expression assignment function ϵ and Nuprl proof hypothesis list H. Although the guess might be incorrect, possibly resulting in false *OE*-typing subgoals for $E1$ or $E2$, our tactics would not be able to prove those false subgoals; the correctness of our *OE*-typing tactics is independent of the type-guessing procedure.

OE-valued expressions built from $\wedge$, $\vee$, $\Rightarrow$, $\equiv$, etc. must have type signature bool if they have any type signature at all, independent of hypotheses or the type expression assignments. This significantly simplifies `OE_sig_guess`. The only other kinds of expressions are variables, *OE* function applications, and textual substitution.

For a variable E, (`OE_sig_guess` $E\ \ \epsilon\ \ H$) guesses a type from *OE*-typing propositions on H and the updates that may be explicitly present on type assignment ϵ. Except for minor details involving those type assignment updates, the process is very straightforward.

For function application, `OE_sig_guess` uses a significant simplification: it only returns the range type of the function, ignoring the number of arguments and their types. For instance, to guess the type of $F(A)(B)$, `OE_sig_guess` recursively guesses the type of F; if it guesses $(k)\tau$ for F, it returns $(0)\tau$ for $F(A)(B)$ without further considering A or B.

The only remaining cases to consider are the two substitution operators, which are handled in the expected recursive manner. For single-variable substitution, (`OE_sig_guess` $E[(V{:}OV) := (P{:}OE)]\ \ \epsilon\ \ H$) is (`OE_sig_guess` E $\epsilon'\ \ H$), where ϵ' is a copy of ϵ with V updated to (`OE_sig_guess` $P\ \ \epsilon$ H). For list-replacement substitution, `OE_sig_guess` works similarly: it returns a type expression list when given an *OE* List for its argument, using an update construct ϵ_P^V defined for lists V and P. Then, (`OE_sig_guess` $E[(V{:}OV\ \text{List}) := (P{:}OE\ \text{List})]\ \ \epsilon\ \ H$) is (`OE_sig_guess` $E\ \ \epsilon'\ \ H$), where ϵ' is a copy of ϵ with list V updated to the list (`OE_sig_guess` $P\ \ \epsilon\ \ H$). As mentioned above, `OE_sig_guess` handles the complexities that come from

Table 6.4: Lemmas for decomposition of *OE*-typing proof goals

```
(6.19) ∀P:OE, ∈:OV⟶OTS, Q:OE.
       (∈) [P] :: bool & (∈) [Q] :: bool ⇒
       (∈) [P ≡ Q] :: bool
```

(Similarly for *OE*-constructors $\wedge$, $\vee$, $\neg$, $\Rightarrow$.)

```
(6.20) ∀P:OE, r:{ind, bool}, ∈:OV⟶OTS, Q:OE.
       (∈) [P] :: r & (∈) [Q] :: r ⇒ (∈) [P = Q] :: bool

(6.21) ∀P,Q:OE, ∈:OV⟶OTS.
       (∃r:{ind, bool}. (∈) [P] :: r & (∈) [Q] :: r) ⇒
       (∈) [P = Q] :: bool

(6.22) ∀b:Qind, X:OV List, R,P:OE, ∈:OV⟶OTS.
       (∈[X⟵ ind]) [R] :: bool & (∈[X⟵ ind]) [P] :: bool ⇒
       (∈) [(*_b X | R : P)] :: bool

(6.23) ∀P,Q:OE, b:Qind, ∈:OV⟶OTS.
       (∈) [P] :: bool & (∈) [Q] :: bool ⇒
       (∈) [P *_b Q] :: bool

(6.24) ∀b:Qind, ∈:OV⟶OTS. (∈) [u_b] :: bool
```

(Similarly for *OE* constants `t_o, f_o`.)

dealing with lists in the updates. In addition, because the type assignment update on two lists is not part of the user-level data language, we are careful to rewrite to an equivalent chain of single-variable updates before returning the type assignment to a user.

6.5.3 Decomposing *OE*-typing proof goals

As with other propositions, part of solving a complex *OE*-typing proof goal is decomposing it into subgoals and solving each of them independently. We implemented tactic `OwfdCD` to do this decomposition. Unlike *OE*-typing hypotheses, *OE*-typing proof goals can be decomposed on *OE*-structure, so `OwfdCD` is significantly more complex than `OwfdHD`. See Tables 6.4 and 6.5 for a list of the key lemmas used in `OwfdCD`.

`OwfdCD` performs decomposition on list structure followed by decomposition on *OE*-structure. The list-decomposition and most of the *OE*-structure

Table 6.5: Lemmas for substitution-decomposition in *OE*-typing proof goals

Lemmas (6.25) and (6.26) are for single-variable *OE*-substitution.

```
(6.25) ∀P:OE, T:OTS, V:OV, tau:OTS, env:OV⟶OTS, E:OE.
       (env[V⟵ tau]) [P] :: T & (env) [E] :: tau ⇒
       (env) [(P)[V := E]] :: T

(6.26) ∀b:Qind, sig:OTS, V:OV, E:OE, ep:OV⟶OTS.
       (ep) [u_b] :: sig ⇒ (ep) [(u_b)[V := E]] :: sig
```

(Similarly for *OE* constants `t_o, f_o`.)

Lemmas (6.27) and (6.28) are for list-based *OE*-substitution.

```
(6.27) ∀Vs:OV List, Es:OE List, Sigs:OTS List, P:OE, T:OTS
       , env:OV⟶OTS.
       Distinct_elts(Vs,OV) & ||Vs|| = ||Es|| & ||Sigs|| = ||Es|| &
       (env[Vs⟵ Sigs]) [P] :: T &
       (∀ESigpair:OE×OTS.
        ESigpair onlist(OE×OTS) zip(Es,Sigs) ⇒
        (ESigpair/E,Sig.(env) [E] :: Sig))
       ⇒
       (env) [P[Vs := Es]] :: T

(6.28) ∀b:Qind, Vs:OV List, Es:OE List, sig:OTS, ep:OV⟶OTS.
       (ep) [u_b] :: sig ⇒ (ep) [u_b[Vs := Es]] :: sig
```

(Similarly for *OE* constants `t_o, f_o`.)

decompositions are straightforward rewrites. Here are details of the less-straightforward cases:

- In the case of a single-variable textual substitution $P[V := E]$, `OwfdCD` generally calls `OE_sig_guess` to guess the type of E, instantiating `tau` in lemma (6.25) with that guess to do the decomposition. The propositions in the antecedent of lemma (6.25) then become new proof goals. `OwfdCD` then recursively decomposes the two new proof goals.

- For the list-based substitution operation $P[V := E]$, `OwfdCD` first checks if V and E are singleton lists, attempting to reduce it to the simpler single-variable substitution case. Assuming it is not reducible, `OwfdCD` generally calls `OE_sig_guess` to guess the list R of types of expressions on E, that is, the list R such that element i is the guessed signature of element i of E. Then, `OwfdCD` instantiates lemma (6.27) to do the decomposition; the five conjuncts in the antecedent of the lemma become new proof goals. In the case of the complex conjunct

  ```
  ∀ESigpair:OE × OTS.
  ESigpair onlist(OE × OTS) zip(Es,Sigs)
  ⇒ (ESigpair/E,Sig. (env) [E] :: Sig)
  ```

  the system reduces it to the equivalent proof goal

  ```
  E:OE, Sig:OTS, ⟨E,Sig⟩ onlist(OE × OTS) zip(Es,Sigs)
  ⊢ (env) [P[Vs := Es]] :: T
  ```

  Tactic `OwfdCD` then recursively decomposes this new subgoal.

  The non-*OE*-typing antecedents of lemma (6.27) that become proof goals in this textual substitution decomposition contain linguistic elements that do not typically occur elsewhere in our calculational logic implementation, such as list length equality and the property that all elements on a list are distinct. We implemented auxiliary tactics to solve these proof goals in straightforward ways; `OwfdCD` simply calls these auxiliary tactics. Users do not encounter these constructs.

- Due to the polymorphism of object-language equality, we cannot decompose a proposition $(\epsilon)\ A = B ::\ \text{bool}$ without knowing the types of A and B. So, `OwfdCD` calls `OE_sig_guess` to guess their types and then backchains through lemma (6.20) from Table 6.4, instantiating the lemma's type argument with the guessed type. If that strategy does not lead to a complete proof —which would happen if the guesses of `OE_sig_guess` were wrong, for instance— `OwfdCD` backchains through lemma (6.21) without guessing any *OE*-types. This results in an existential quantification over the possible types of A and B; the user would need to instantiate that quantifier by hand to complete with the proof.

The use of two conceptually different approaches to decomposing object-language equalities, embodied by lemmas (6.20) and (6.21), illustrates both the heuristic nature of our implementation and the division of labor between the system and its users. We automated enough detail for a simple explanation of Leibniz inference, but more sophisticated guessing is beyond the level of expertise we intended to embody in our tactics.

6.5.4 Solving *OE*-typing proof goals

We implemented two general approaches to solving *OE*-typing proof goals. One is a simple simulation of direct computation, which works as follows:

- We use a boolean predicate to determine if the proof goal is a good candidate for solution by direct computation. It takes a type assignment function, a Nuprl variable, and a type from a proof goal; it checks type assignment updates (nothing else) to see if that Nuprl variable is assigned that type by that type assignment. If so, that proof goal can be solved by direct computation simulation.
- The solve-by-direct-computation tactic is called only on proof goals that satisfy the above predicate. It either solves the proof goal directly using lemma (6.29) (`(ep[x` $\leftarrow$ `sig]) [x] :: sig`) or it unrolls one variable from the type assignment updates using lemma (6.30) ($\neg$ `x = y` $\Rightarrow$ `(ep) [y] :: sig2` $\Rightarrow$ `(ep[x` $\leftarrow$ `sig]) [y] :: sig2`) and calls itself recursively on the new *OE*-typing subgoal. In the update-unrolling case, an *OV*-inequality subgoal also emerges; tactic `solve_noteqOV` is called to solve it. This solve-by-direct-computation tactic fails unless it completes the proof (except for Nuprl well-formedness goals).

The key lemmas used in this method are given in Table 6.6. All the type assignment updates are single-variable updates. In our tactics, we will have rewritten all updates to the single-variable kind before applying this method. This is normal practice for us: typically, we develop tactics to work on single-variable updates, and we are responsible for ensuring that the data is in the correct form.

Our second approach to solving these proof goals is based on the observation about type assignments represented by lemma (6.31) in Table 6.7: if we know $(\epsilon')\ E :: \tau$ and we know that ϵ and ϵ' assign the same types to all variables that occur free in E, then we can conclude $(\epsilon)\ E :: \tau$. Solving by type assignment agreement is significantly deeper than solving by direct computation; Table 6.7 contains the key lemmas used in this approach.

Table 6.6: Lemmas for solving *OE*-typing proof goals by direct computation

```
(6.29) ∀x:OV, sig:OTS, ep:OV⟶OTS. (ep[x⟵ sig]) [x] :: sig

(6.30) ∀x,y:OV, sig2:OTS, ep:OV⟶OTS, sig:OTS.
       ¬x = y ⇒ (ep) [y] :: sig2 ⇒ (ep[x⟵ sig]) [y] :: sig2
```

Table 6.7: Lemmas for solving *OE*-typing proof goals by type assignment agreement

```
(6.31) ∀E:OE List, ∈1,∈2:OV⟶OTS, sig:OTS.
       (∀x:OV. (some [x] occur free in E) ⇒ (∈1,∈2 agree on x))
       ⇒
       (∈1) E :: sig ⇒ (∈2) E :: sig

(6.32) ∀x:OV, L:OE List, ∈:OV⟶OTS, sig:OTS.
       ¬(some [x] occur free in L) ⇒
       (∀y:OV.
        (some [y] occur free in L) ⇒ (∈,∈[x⟵ sig] agree on y))

(6.33) ∀x:OV, L:OE List, ∈:OV⟶OTS, sig:OTS.
       ¬(some [x] occur free in L) ⇒
       (∀y:OV.
        (some [y] occur free in L) ⇒ (∈[x⟵ sig],∈ agree on y))

(6.34) ∀ep:OV⟶OTS, x:OV. (ep,ep agree on x)

(6.35) ∀L:OE List, ep1,ep2:OV⟶OTS, v:OV, sig:OTS.
       (∀x:OV. (some [x] occur free in L) ⇒ (ep1,ep2 agree on x))
       ⇒

       (∀x:OV.
        (some [x] occur free in L) ⇒
        (ep1[v⟵ sig],ep2[v⟵ sig] agree on x))

(6.36) ∀x,y:OV, L:OE List, t,sig2,sig1:OTS, ep:OV⟶OTS.
       ¬x = y ⇒
       (ep[x⟵ sig1][y⟵ sig2]) L :: t ⇒
       (ep[y⟵ sig2][x⟵ sig1]) L :: t
```

To solve a proof goal $(\epsilon)\ E :: \tau$, our type assignment agreement tactics need to identify a hypothesis of the form $(\epsilon')\ E :: \tau$ to match the pattern of lemma (6.31). Then, upon instantiation, $(\epsilon')\ E :: \tau$ and $(\epsilon')\ E :: \tau \Rightarrow (\epsilon)\ E :: \tau$ are both true by hypothesis, and the desired proof goal follows directly.

This instantiation also results in a new proof goal: $\forall x{:}OV.(\text{some } [x] \text{ occur free in } e) \Rightarrow (\epsilon, \epsilon' \text{ agree on } x)$. Thus, solving proof goals of this form becomes an essential task, and tactics based on lemmas (6.32)-(6.34) are devoted to this purpose. Instantiating these lemmas either solves the proof goal or reduces it to a not-occur proposition, which is solved by tactic `solve_not_occ_concl`.

We also implemented two tactics to expand the cases that can be handled with this method. On a type assignment agreement proof goal, rewriting by lemma (6.35) can reduce it to a previously described case. On an *OE*-typing proof goal, rewriting by lemma (6.36) can make a goal solvable using type assignment agreement methods; `solve_noteqOV` solves the OV-inequality proof goal that comes from such a rewrite. These are simple ways of making the basic *OE*-typing proof methods more intelligent. There are certainly more, but these often suffice in practice; for instance, there are rarely more than two updated variables in a type assignment in the cases from [GS93b].

Our tactic for solving *OE*-typing proof goals, `solve_Owfd`, uses the methods described above. After decomposition steps `OwfdCD` and `OwfdHD` (if necessary), it does the following:

- It checks if the proof goal is one of the hypotheses. Recall that `OwfdHD` does not overwrite or thin intermediate steps in the decomposition, which increases the likelihood of this simple solution.
- If it has not yet succeeded, it tries to solve the proof goal by direct computation.
- If it has not yet succeeded, it tries the more complicated approach of solving by type assignment agreement.

It is somewhat surprising that these simple methods account for so many cases. From a cognitive modeling perspective, they correspond to a novice user of calculational logic who knows only basic techniques. Note that this hypothetical mathematician (or math student, perhaps) gets quick results in his learning process, not needing complex techniques to work through the proofs in [GS93b]. Other, more expert mathematicians/students would be modeled with more deeply developed tactics.

6.5.5 *OE*-typing discussion

One deficiency of our current implementation is that we lack lemmas to work effectively with variables over type *OTS*. For instance, we could not infer that

$V =$ bool from a hypothesis (ϵ) True$:: V$. Reasoning with such variables could be an important part of calculational logic inference —for instance, reasoning about the typing of *OE*-equalities might result in a hypothesis that an expression is of some incompletely specified *OE*-type, which would be represented by a variable. We avoided such issues in this first-level implementation. Considerable complexities might arise in automating methods for this kind of inference, so our system currently requires assistance from users in such contexts where *OE*-type variables need to be further specified.

Our lack of knowledge about complications that might arise from eventual extensions to our system also manifests itself in other ways. For instance, our *OE*-typing relation (based on the clauses for *OE* type assignment in chapter 3) is not a true definition: the clauses are implications, not equivalences. We do not, however, use the definition of the *OE*-typing relation in our tactics. That is, we do not unfold the definition to get at its computational innards; instead, we use lemmas that state properties about that definition. This is sensible for our system, which anticipates as-yet-unspecified extensions. Closing a definition at this level, only to rework it to accommodate the extensions we troubled to allow, would take considerable effort while yielding little information about the inference process itself.

6.6 Inferring Othm Judgments

Calculational logic relies primarily on inference rule Leibniz to establish the theoremhood of object expressions. Still, some simple object theoremhood inferences are needed as components of Leibniz. For instance, one of the antecedents of each Leibniz rule is an object-theoremhood predicate, so by instantiating the Leibniz lemma as part of a calculational logic proof step, we generate an object-theoremhood proof goal. Typically, this is an easy goal to solve, following directly from the instantiated premise theorem, but we must still manage *Othm* judgments as part of our Leibniz tactic.

We need only two methods for solving the auxiliary *Othm* subgoals generated by `Leib_tac`. One method establishes the theoremhood of expressions of the form $A = B$ when A and B are of *OE* type `bool` and equal modulo associativity of $\equiv$. This was essential to accommodate associativity of $\equiv$ in our implementation. It was the minimal concession to automatic inference about associativity of binary *OE*-operators (as in the method in [GS93b]) that we could reasonably make.

The other method establishes object-theoremhood proof goals from hypotheses, modulo symmetry of *OE*-constructor =. This is a commonly needed auxiliary to Leibniz steps; in standard practice, users of calculational logic expect a premise theorem instantiated to $A = B$ to match both proof goals $A = B$ and $B = A$. Since instantiations of premise theorems become Nuprl

Table 6.8: The Gries/Schneider representations of Leibniz rules

(6.37) $$\frac{P = Q}{E[z := P] = E[z := Q]}$$

(6.38) $$\frac{P = Q}{(\star x \mid E[z := P] : S) = (\star x \mid E[z := Q] : S)}$$

(6.39) $$\frac{R \;\Rightarrow\; P = Q}{(\star x \mid R : E[z := P]) = (\star x \mid R : E[z := Q])}$$

hypotheses in our implementation, we needed to match against hypotheses modulo symmetry of $=$.

In contrast to our approaches to solving OV-inequality and OE-typing goals, we do not have a tactic `solve_Othm` or some similar wrapper to automatically choose which method to use in a given situation. The object-theoremhood strategies needed as preliminaries to a simple Leibniz tactic are easy to manage, and there is no situation when both might apply. Each of the two methods is designed for a specific purpose. There is no greater sophistication in our implementation. The above paragraphs describe the full extent of our development involving object-theoremhood judgments, not including the Leibniz tactic itself.

We have not yet implemented the definition of object-theoremhood presented in chapter 3. In general, we are concerned primarily with proving object theorems from other object theorems, not basing every single object-theoremhood judgment formally in the definition each time. Therefore, we used Nuprl's `Fiat` tactic to establish the object-theoremhood of premise theorems such as the One-point rule, which is used in the first Leibniz step of the proof of Theorem Change of Dummy. We are confident that we could prove it directly from an implementation of our definition, but doing so would not illuminate the primary cognitive inferences that we are trying to model.

6.7 The Leibniz Tactic

The three Leibniz rules that encode substitution of equals for equals are the most important inferences in calculational logic. (See Table 6.8 for the representations of these rules as given in [GS93b].) Typically, we consider a calculational proof to be a chain of some number of equalities; the Leibniz rules are the means by which users establish the component equalities. Calculational proof steps typically correspond to single applications of Leibniz; calculational logic is essentially structured around Leibniz rules.

On a given proof goal, our Leibniz tactic first decides which of the three Leibniz rules to apply. Each rule is embodied in a Nuprl lemma; see Table 6.9 for the three theorems. After identifying which lemma to use, it instantiates the lemma with arguments chosen specifically for the purpose of justifying that proof goal. Then, with this framework established, it performs the inferences needed to complete the proof step.

Our past development, however, is not sufficient for this task. Although we can choose a Leibniz lemma, and we can solve all the proof goals that come from applying it by calling previously described tactics, intelligent instantiation and application of the chosen Leibniz lemma can be quite complex. In particular, the two textual substitutions operations replacing Z in the conclusion of the Leibniz lemmas in Table 6.9 (we call these substitutions *Leibniz substitutions* to distinguish them from others) must be properly managed.

For an example, we return to the first step in the Change of Dummy proof (Figure 6.1). Presume that `Leib_tac` has correctly identified Leibniz lemma (6.42) as the proper one for the step. Consider what else is involved in arriving at the proper instantiation for that lemma.

The object-theoremhood proposition in the antecedent must directly follow from the premise to `Leib_tac`, and the conclusion of the Leibniz lemma must be the proof goal. Inspection reveals, however, that the textual substitution sign for Leibniz substitution in Leibniz lemmas is not intended to match a textual substitution sign in the proof goal. Indeed, it is intended in calculational logic that the expressions *resulting from* the Leibniz substitutions match the proof goal.

This has several ramifications. Our tactics must be able to carry out these Leibniz substitutions. For that, we will want to instantiate Z with a fresh variable, so that no capture-related complications arise. In addition, we will also need to instantiate other variables (such as P, R, A, B) using methods that are more complicated than simply pattern-matching against Nuprl terms in the proof goal: simple pattern-matching could not possibly work correctly until the tactics performed the Leibniz substitutions, and they cannot perform the Leibniz substitutions without instantiating the lemma. Because of this, our Leibniz tactics use heuristics to guess instantiation values and then separately verify their correctness (if possible). In this chapter, we describe those heuristics and other aspects of carrying out a Leibniz proof step using tactics.

It is useful to have names for the component expressions of a textual substitution, so when talking about a substitution expression $E[V := P]$ in which a single expression P is to be substituted for a single variable V in E, we call

Table 6.9: The three Leibniz lemmas in Nuprl

(6.40) The simple case.

```
∀E,A,B:OE, ∈:OV⟶OTS, α:OE List, r,r':{ind, bool}, Z:OV.
(∈) α ⊢ A = B & (∈) [A; B] :: r &
(∈[[Z]⟵ r]) E :: r' ⇒
(∈) α ⊢ (E)[Z := A] = (E)[Z := B]
```

(6.41) For substitution into quantifier ranges.

```
∀b:Qind, Xs:OV List, R:OE, Z:OV, A,P,B:OE, ∈:OV⟶OTS
, α:OE List, r:{ind, bool}.
¬(some Xs occur free in α) &
(∈[Xs⟵ ind]) α ⊢ A = B &
(∈[Xs⟵ ind]) [P] :: bool &
(∈[Xs⟵ ind]) [A; B] :: r &
(∈[Xs⟵ ind][[Z]⟵ r]) [R] :: bool
⇒

(∈) α ⊢ (*_b Xs | (R)[Z := A] : P)
         =
         (*_b Xs | (R)[Z := B] : P)
```

(6.42) For substitution into quantifier bodies.

```
∀Xs:OV List, α:OE List, R,A,B:OE, ∈:OV⟶OTS,
r:{ind, bool}, P:OE, Z:OV, b:Qind.
¬(some Xs occur free in α) &
(∈[Xs⟵ ind]) α ⊢ R ⇒ A = B &
(∈[Xs⟵ ind]) [A; B] :: r &
(∈[Xs⟵ ind][[Z]⟵ r]) [P] :: bool
⇒

(∈) α ⊢ (*_b Xs | R : (P)[Z := A])
         =
         (*_b Xs | R : (P)[Z := B])
```

E the *substitution body*, V the *substitution variable*, and P the *substitution formula*. When clear, we may omit the word "substitution" from all three terms.

6.7.1 A simple `Leib_tac`

We started by creating a simple tactic `Leib_tac` that we thought could perform most basic Leibniz steps. Although we expected that this implementation would not embody enough knowledge to handle all the steps in the Change of Dummy proof, we discovered that only two of the proof steps required more advanced inferences. In this section, we describe the simple `Leib_tac`. In the next section, we describe the few extensions needed to make it sufficient for the entire Change of Dummy proof.

Instantiating the Leibniz substitution

As previously mentioned, the substitution variable in a Leibniz substitution must be a fresh variable. Finding a fresh variable is a familiar procedure from contexts other than calculational logic, but our metalinguistic implementation case brings a depth to the task that may not be immediately apparent. We do *not* explicitly find a new object variable that does not occur in some explicitly given object expressions. Instead, we operate on metavariables —variables in our data language— that range over object expressions; we don't even explicitly know the object expressions in which we need our fresh object variable not to occur. So, instead of identifying a particular fresh object variable, we find a new Nuprl variable $V{:}OV$ and introduce an assumption to the effect that V refers to a new object variable. Thus, there are two senses in which a new variable is being "found": V is a new metavariable that hypothetically stands for a new object variable.

The key lemma that our tactics use to generate the fresh variable is [5]

(6.43) `∀L:OE List. ∃x:OV. ¬(some [x] occur free in L)`

We instantiate the OE List argument with a list (with duplicates removed) of all the data-language variables that stand for object expressions in which we need to guarantee that our fresh variable does not occur; this instantiation of L results in a hypothesis of the form $\exists x{:}OV.\neg(\text{some }[x]\text{occur free in }L')$. Then,

[5] Interestingly, this lemma is itself a proposition of the data language, not at some meta-level above it. This is consistent with the fluid integration of the items called "theorems" and "metatheorems" in [GS93b]. The data language in which the "theorems" are written is expressive enough for "metatheorems" as well; no linguistic barrier prevents their integration in a coherent account of calculational logic.

we let Nuprl instantiate the resulting existential quantifier in that hypothesis; we use the witness it generates as V, the substitution variable for our Leibniz substitutions. After a call to `Not_OccHD`, any fresh-variable property we need for our proof goal is expressed in the hypotheses as a not-occur proposition on V and some other data-language variable.

Having found a name for the new substitution variable, our tactics can now use it in an attempt to guess the expression for the substitution bodies in the Leibniz lemma. (In all three Leibniz lemmas, there are two substitutions, and the expression for the body is the same in both.) Typically, the body will include the new substitution variable; vacuous Leibniz substitutions are uncommon.

To arrive at an expression for the substitution body, we implemented a heuristic method that corresponds to a simple but frequently sufficient case for Leibniz steps: our tactic will not guess an expression in which the substitution variable appears more than once. Although this commonly occurring case suffices for all the proof steps in the Change Of Dummy proof, there are cases for which it is not adequate. For instance, to prove $(C \wedge A \equiv C \vee A) = (C \wedge B \equiv C \vee B)$ from premise theorem $A = B$ using Leibniz, one would choose $C \wedge V \equiv C \vee V$ for the substitution body, where V is the Leibniz substitution variable. Variable V occurs twice, however, so for this case our substitution body-guessing tactic would make an incorrect guess and our Leibniz tactic would be unable to prove this instance of Leibniz correct. Although calculational logic novices might make an erroneous guess in the above case, experts certainly would not, and we could certainly build tactics that would demonstrate more expert behavior. Indeed, the structure of our existing tactic would readily accommodate such an extension. We discuss the matter of simplifications in substitution-body guessing at greater length in subsection 6.7.3.

Our simple procedure to guess the components of Leibniz substitutions exploits the restriction of substitution body guesses to expressions without multiple occurrences of the substitution variable. To guess the substitution components needed for Leibniz to establish the theoremhood of an equality of the form $A = B$, our ML procedure `Leib_subst_guess` essentially performs a simultaneous recursive descent on the structures of A and B. Note that it guesses the substitution body and the two substitution formulas simultaneously; it is provided the name of the fresh variable for use in constructing the substitution body (we use the name V in the description below). In the general case, `Leib_subst_guess L R` V works as follows, where `L` and `R`, respectively, are the current components of the expressions on the left and right of the proof-goal *OE*-equality:

- If the outermost operators of `L` and `R` differ, or if `L` and `R` differ in more than one subterm, let the body guess be V, the left substitution formula

```
letrec mk_assoc_sbtm_list opid term≡
 if opid_of_term term=opid
 then let [subterm1;subterm2] ≡ subterms term in
      append (mk_assoc_sbtm_list opid subterm1)
      (mk_assoc_sbtm_list opid subterm2)
 else [term] ;;

letrec form_list_for_AC_guess_V1 L1 L2 L3≡
 if L1=[] or L2=[] then L1,L2,L3
 else if hd L1=hd L2
 then form_list_for_AC_guess_V1 (tl L1) (tl L2) (hd L1).L3
 else L1,L2,L3 ;;
```

Figure 6.4: ML functions used in Leibniz substitution-body guessing

guess be `L`, and the right substitution formula guess be `R`.

- If `L` and `R` are equal, return `L` as the body guess. (This guess may well be a recursively determined component of the guess returned by a call of `Leib_subst_guess` on expressions for which `L` and `R` are corresponding subterms.) Because `L` and `R` are the same, we do not expect the Leibniz substitution to alter this part of the term structure, so it is not surprising that V does not occur in the body guess. In this case, no guesses for substitution formulas are necessary.

- If `L` and `R` are identical except for exactly one immediate OE subterm —say `L1` from `L` differs from corresponding subterm `R1` from `R`, but all other corresponding subterms match— let the body guess be a copy of `L` with `L1` replaced by the body guess of `Leib_subst_guess L1 R1` V. For the left and right substitution formula guesses, return the corresponding guesses from `Leib_subst_guess L1 R1` V.

This describes `Leib_subst_guess` in the normal case. The exception is for operator $\equiv$, because we need to accommodate its associativity. (Accommodating commutativity of $\equiv$ would have resulted in a more complex strategy than we intended our prototype explanation of cognitive inference to model.) If `L` were $A \equiv (B \equiv C)$ and `R` were $(A \equiv B) \equiv D$, we would not want to follow the above procedure, which would return V as the body guess (because `L` and `R` differ in both corresponding subterms). Instead, the body guess should be $A \equiv B \equiv V$. As a more complex example, if `L` were $(A \equiv B) \equiv (C \equiv D)$ and `R` were $A \equiv (E \equiv D)$, the body guess should be $A \equiv V \equiv D$. Therefore, we have an auxiliary procedure `Leib_subst_guess_for_AC_ops` for this case. `Leib_subst_guess_for_AC_ops L R` V works as follows:

- First, using ML procedure `mk_assoc_sbtm_list` with `opid` representing $\equiv$ (see Figure 6.4 for its ML code), form $\equiv$-subterm lists of `L` and `R`. For example, `mk_assoc_sbtm_list` $A \equiv ((B \equiv (C \vee D)) \equiv E)$ is $[A; B; C \vee D; E]$. For purposes of this description, let `Ls` and `Rs` be `mk_assoc_sbtm_list L` and `mk_assoc_sbtm_list R` respectively.

- (In the following, note that `rev` is the ML list-reversing function.) Then, using ML procedure `form_list_for_AC_guess` (see Figure 6.4 for its ML code), form four lists from `Ls` and `Rs`: Let `Ls', Rs', Lts` be `form_list_for_AC_guess Ls Rs []`; let `revLg, revRg, Rts` be the return values of `form_list_for_AC_guess (rev Ls') (rev Rs') []`, and let `Lg` and `Rg` be `rev revLg` and `rev revRg`, respectively. Then,
  - `Lts` is a list of the subterms that match from the heads of `Ls` and `Rs`
  - `Rts` is a list of the subterms that match from the the terms that from the tails of `Ls` and `Rs`
  - `Lg` is the list `L` with the matches from its head and tail deleted.
  - `Rg` is the list `R` with the matches from its head and tail deleted.

- (In the following, ML procedure `mk_assoc_term_from_subterm_list` is the inverse of `mk_assoc_subterm_list`: given the opid of $\equiv$, it creates an $\equiv$-term from a term list.) Unless the four lists formed in the above step satisfy a boundary condition, we use `mk_assoc_term_from_subterm_list` and the four lists to create guesses. The body-guess is the $\equiv$-term created from the list `Lts @ (`V`.Rts)`, where `@` and `.` are the ML list-append and cons operators, respectively. The guesses for the left-side and right-side substitution formulas are the $\equiv$-terms created from `Lg` and `Rg`, respectively.

- One boundary case we catch is if either `Lg` or `Rg` are empty as the result of one of `L` or `R` completely matching the other from either the head or the tail. For instance, consider what would happen if we used this guessing procedure on proof goal $(A \equiv B) = (A \equiv B \equiv C)$, where V was the name for the new variable. (Such a case could occur if the position of C in the example above is occupied by expression `t`.) Our substitution-body guess would be $A \equiv B \equiv V$, and our right-formula guess would be C, but our left-formula guess would be `nil`, indicating a bad body guess. We handle such a boundary case by extending both formula guesses by the term immediately next to the variable. In this case, we would change our body guess to $A \equiv V$, with the right-formula guess becoming $B \equiv C$ and the left-formula guess becoming B.

- The other boundary case we currently handle is `Lg` and `Rg` both of size 1 when they are constructed. In this case, we have a single subterm from `L` and a corresponding subterm from `R`, and there may be more term structure to consider. For instance, consider the case where `L` is $A \equiv (P \wedge Q) \equiv B$ and `R` is $A \equiv (P \wedge R) \equiv B$. After their construction, `Lg` and `Rg` would be `[` $P \wedge Q$ `]` and `[` $P \wedge R$ `]`, and the body guess would be $A \equiv V \equiv B$. A smarter guess, however, results from continuing the body-guessing procedure through the structure of the terms in `Lg` and `Rg`. `Leib_subst_guess` recursively calls itself on those terms, arriving at a body guess of $A \equiv (P \wedge V) \equiv B$, with left-formula guess Q and right-formula guess R.

Our implementation of these procedures is somewhat more general than it needs to be. It is designed to accommodate a general theory of associative/commutative operators, as are procedures `mk_assoc_sbtm_list` and `mk_assoc_term_from_subterm_list` ; for instance, to account for other associative operators, we would simply need to alter a list in `Leib_subst_guess` that right now contains only $\equiv$. In section 6.7.3, we further discuss the role of associative/commutative operators in implementing calculational logic.

Putting it together and cleaning up

Given a procedure that can guess the elements of the Leibniz substitutions, the rest of the main Leibniz tactic `Leib_tac` is reasonably straightforward. As previously described, `Leib_tac` is designed to establish the object theoremhood of expressions of the form $A = B$. It takes two arguments, the name of a lemma to use as the premise (after instantiation) and a list of terms to use when instantiating the premise. By analyzing the structure of object expressions A and B, it chooses one of the three Leibniz lemmas to use. Then, it instantiates that lemma by matching against the proof goal and using guesses of the components of the Leibniz substitutions as provided by the procedures described in the previous section.

One of the results of instantiating a Leibniz lemma is that its conclusion becomes a hypothesis to use when proving the original proof goal. To put the hypothesis in the proper form, `Leib_tac` carries out the Leibniz substitutions by recursively rewriting the expressions using the lemmas in Table 6.10. If the substitution guesses are good enough, this instantiation will solve the desired proof goal. There are a few subtle points to this Leibniz rewriting, but they do not affect the overall flow of `Leib_tac`, so we postpone consideration of them until subsection 6.7.3.

The other proof goals resulting from the Leibniz lemma instantiation correspond to the propositions in its antecedent. One of those is an object-theoremhood goal, which should follow (using our simple `solve_Othm` tactic) from instantiating the user-supplied premise theorem with the user-supplied

Table 6.10: Lemmas used in carrying out Leibniz substitution

(6.44) ∀P:OE, V:OV, E,Q:OE.
(P)[V := E] ≡ (Q)[V := E] = (P ≡ Q)[V := E]

(Similarly for *OE*-constructors ∧ , ∨ , ¬, ⇒ , =.)

(6.45) ∀f,x:OE, V:OV, E:OE.
(f(x))[V := E] = (f)[V := E]((x)[V := E])

(6.46) ∀b:Qind, P:OE, V:OV, E,Q:OE.
(P)[V := E] *_b (Q)[V := E] = (P *_b Q)[V := E]

(6.47) ∀b:Qind, V:OV, E:OE. u_b = (u_b)[V := E]

(Similarly for *OE* constants `t_o, f_o`.)

(6.48) ∀x:OV, E:OE. (x)[x := E] = E

(6.49) ∀X:OV, P,E:OE.
¬(some [X] occur free in [P]) ⇒ (P)[X := E] = P

term list; in the case of Leibniz lemma (6.42), an additional backchain through the following lemma

(6.50) ∀P:OE, ep:OV⟶OTS, Q:OE, alpha:OE List.
(ep) [P] :: bool & (ep) alpha ⊢ Q ⇒ (ep) alpha ⊢ P ⇒ Q

may be necessary if the premise theorem is an equality instead of an implication. If the premise instantiation is somehow not correct, tactic `solve_Othm` will not be able to solve the Leibniz proof goal. We consider this case a user error, since the user specifies both the premise and the terms to use in instantiating it, and we make no effort to correct it.

`Leib_tac` must also solve the proof goals corresponding to the propositions in the antecedents of the Leibniz lemma and the premise theorem that were instantiated. We call `solve_not_occ_concl`, `solve_Owfd`, `solve_noteqOV` and preprocessing tactics (`Not_OccHD`, `OwfdHD`, etc.) when needed to handle them. We do not know in advance exactly what new proof goals might appear from the premise instantiation, so we must call all our possibly useful tactics to solve them. We use a more structured approach for proof goals that result from Leibniz lemma instantiation and other such cases where we can hardwire into the tactic knowledge of the lemma being instantiated.

6.7.2 Extending the simple `Leib_tac`

Recall that `Leib_tac` as described above can perform all but two of the Leibniz steps in the proof of Theorem Change of Dummy from [GS93b]. In this section, we describe the few, straightforward extensions needed to make `Leib_tac` adequate for that entire proof. These are extensions to the implementation already described; the notes, descriptions, etc. above continue to apply to the completed `Leib_tac`.

The two steps `Leib_tac` could not handle were:

(6.51)
$$\begin{array}{ll} & (\star x, y \mid R[x := f.y] \land x = f.y : P) \\ = & \langle \text{Substitution (3.84a) —} R[x := f.y] \text{ must be removed} \\ & \quad \text{at some point. This substitution makes it possible.} \rangle \\ & (\star x, y \mid R[x := x] \land x = f.y : P) \end{array}$$

and

(6.52)
$$\begin{array}{ll} & (\star x \mid R : (\star y \mid x = f.y : P)) \\ = & \langle x = f.y \;\equiv\; y = f^{-1}.x \text{ —This step prepares for the} \\ & \quad \text{elimination of } y \text{ using the One-point rule.} \rangle \\ & (\star x \mid R : (\star y \mid y = f^{-1}.x : P)) \end{array}$$

(See page 35 for "Substitution" and "One-point rule.") The steps in (6.51) required significant additions to `Leib_tac` in two major areas: instantiating the Leibniz substitution and solving *OE*-typing goals by type assignment agreement.

We enhanced our heuristic for finding the Leibniz substitution body by adding a check to the recursive procedure described above to keep it from making too deep a descent. Consider a case where, at some point in its recursive descent, the procedure identified potential substitution formulas `Lg` and `Rg` that were exactly the sides of the object-level equality established by the premise theorem. Then, it would not continue the search to instantiate the Leibniz lemma; the premise theorem proves `Lg` and `Rg` equal, and it would not prove the equality of any corresponding subterms. For example, for proof step (6.51), our original heuristic guessed $(\star_b\ x, y \mid \mathtt{V} \land x = f.y : P)$ as the Leibniz substitution body, but that was incompatible with the premise theorem:

(6.53) **Substitution (3.84a):** $E[z := f] \land (e = f) \;=\; E[z := e] \land (e = f)$

With this new check added, the procedure stopped when it had `Lg` and `Rg` as $R[x := f.y] \land x = f.y$ and $R[x := x] \land x = f.y$, which matched the instantiated premise.

Table 6.11: More lemmas for proving type assignment agreement

(6.54)
```
∀x:OV, sig:OTS, ep:(OV⟶OTS).
ep[x⟵ sig][x⟵ sig] = ep[x⟵ sig]
```

(6.55)
```
∀L:OE List, e1,e2:(OV⟶OTS), v:OV, sig:OTS.
(∀x:OV. (some [x] occur free in L) ⇒ (e1,e2 agree on x)) &
(∀x:OV.
 (some [x] occur free in L) ⇒ (e2,e2[v⟵ sig] agree on x))
⇒

(∀x:OV.
 (some [x] occur free in L) ⇒ (e1,e2[v⟵ sig] agree on x))
```

(6.56)
```
∀∈:(OV⟶OTS), x:OV, f:(OV⟶OTS).
(∈,f agree on x) ⟺ (f,∈ agree on x)
```

We implemented this extension in the straightforward way. It encodes an obvious cognitive strategy: keep the premise in mind, and stop looking when you find what you're looking for. It was not in our simpler `Leib_tac`, because it required keeping the premise in mind; the previous strategy required *no outside information*, not even the premise.

The other major changes needed for this proof step are improvements in proving type assignment agreement goals. Because this step requires reasoning inside both quantification (with two dummy variables) and textual substitution, the type assignment updates are more complex than for other inferences. To accommodate this, we implemented the following (see Table 6.11):

- A way to reduce complexity by eliminating duplicate updates, using theorem (6.54).
- A simple way of generating intermediate subgoals to prove a goal, using theorem (6.55).
- Symmetry regarding type assignments, using theorem (6.56). The simpler `Leib_tac` used so few theorems that we didn't need a general symmetry theorem. Now that we anticipate more complex inferences, we include this symmetry theorem.

The only other significant addition was to make `solve_not_occ_concl` smarter when solving not-occur proof goals, using the lemma

(6.57) `∀V:OV, P,E:OE.`
`¬(some [V] occur free in [P]) ⇒`
`¬(some [V] occur free in [(E)[V := P]])`

and a similar lemma for list-based substitution. Jointly, all these additions make `Leib_tac` sufficient for proof step (6.51).

The simple `Leib_tac` failed on proof step (6.52) for entirely different reasons. In that step, the Leibniz substitution is nested inside two levels of quantifications; we replace the range of a quantifier that is itself the body of another quantifier. The Leibniz rules only permit substitution through one level of quantification, however, and no rule directly justifies this proof step.[6]

In general, Leibniz substitution into quantifiers requires iterated application of a Leibniz rule, one iteration per level of nested quantification. The simple `Leib_tac` only handled the special case where only one Leibniz application was necessary. For this proof step, we extended `Leib_tac` by enabling iterated Leibniz inference for arbitrary nesting depth.

It is straightforward to determine the depth of quantifier nesting from the instantiation of the Leibniz substitution. In our extended `Leib_tac` implementation, if the quantifier depth is 0 or 1, we perform the inference exactly as described in section 6.7.1. If repeated Leibniz applications are needed, we instantiate modus ponens to set up the necessary iterations. For instance, to prove the object theoremhood of $(\star_{b1}\ x \mid R1 : (\star_{b2}\ y \mid R2 : P)) = (\star_{b1}\ x \mid R1 : (\star_{b2}\ y \mid R3 : P))$ from some premise theorem, we would instead do the following:

- Using Leibniz steps, prove the object theoremhood of $(\star_{b2}\ y \mid R2 : P) = (\star_{b2}\ y \mid R3 : p)$ from the given premise theorem. This may require more than one iteration of Leibniz rules, so we recursively use this iteration/modus ponens framework.

- Prove that the object theoremhood of $(\star_{b2}\ y \mid R2 : P) = (\star_{b2}\ y \mid R3 : p)$ implies the object theoremhood of $(\star_{b1}\ x \mid R1 : (\star_{b2}\ y \mid R2 : P)) = (\star_{b1}\ x \mid R1 : (\star_{b2}\ y \mid R3 : P))$. This follows from a single application of Leibniz; no further iteration is needed for this goal.

From modus ponens, the above two goals establish the original desired theorem. Of course, all of this is done with respect to the appropriate type assignments and assumptions for the object-theorem predicates.

[6] Although not mentioned in [GS93b], Gries [Gri] suggests using a different, capture-permitting substitution for Leibniz substitution. This would permit substitution through nested quantifiers. Our explanation avoids the logic difficulties with capture-permitting substitution and uses only the rules presented in [GS93b].

Implementing this iteration strategy in `Leib_tac` is straightforward. It does require a different programming structure from the simple `Leib_tac` — we pass different arguments between component subtactics and several of the component tactics that were previously disjoint must be made mutually recursive to support the iteration— but the only major conceptual addition is modus ponens.

6.7.3 Discussion

The `Leib_tac` augmented by the strategies in subsection 6.7.2 handles many common cases in calculational logic inference, including all those in the proof of Change of Dummy. Still, many important extensions have not yet been explored. For instance, we have not developed user/system interactivity. We anticipate that this calculational logic system could come to be used as an interactive assistant for users of the calculational method. As such, it might try to guess the correct instantiation for a Leibniz lemma and, if it failed, it might prompt the user for assistance as needed. Such an improved interface could add a new dimension to our tactics as a cognitive model —the interaction between novice and expert, perhaps— but they are not our main focus at this stage of development.

In the remainder of this section, we examine other issues in extending and improving our current system.

The heuristic nature of `Leib_tac`

Tactic `Leib_tac` is central to both our formalized account of calculational logic and our cognitive model of calculational inference. It encompasses a considerable amount of interesting human inference, such as the decision-making and logical deductions that occur in instantiating the necessary lemmas for the particular context at hand. If we expanded our conception of `Leib_tac` to what an extended version might accomplish —e.g., a future version might actually search a lemma library to find an appropriate premise lemma and its correct instantiation— it becomes still clearer that it is the most sophisticated part of our cognitive model. For this reason, it is quite sensible that components of `Leib_tac` be heuristic rather than necessarily correct. Modeling ideal inference is not our primary goal; we want instead to model human inference, and people are sometimes incorrect.

Because our `Leib_tac` tactic must be heuristic, there was no reason not to accept simple heuristics in this initial implementation of our tactic/model. For instance, we now require that users supply a full list of terms to instantiate the premise lemma given to `Leib_tac`. In a weak way, this is consistent with the practice in [GS93b]: in the textbook, typically only the instantiations that are somehow tricky to figure out would be supplied in a term list. Indeed, the list is often empty (and omitted). So, right now our tactic model corresponds to a user who finds *every* instantiation tricky, which is (we hope!) not a typical practitioner.

In addition, our heuristics for instantiating Leibniz lemmas correspond to a simple strategy —one that a novice student of calculational logic might use— of only finding a match in which the fresh Leibniz variable encompasses all the unequal subterms in the expressions we are trying to match. Consider a case where two unequal subterm positions are not contiguous, as with the terms $a \equiv b \equiv c \equiv d \equiv e$ and $a \equiv f \equiv c \equiv d \equiv g$, where $b \neq f$ and $e \neq g$. Our guess for a substitution body would be $a \equiv Z$, because we don't accommodate the symmetry of $\equiv$ reasoning that would permit the more intelligent guess $a \equiv c \equiv d \equiv Z$. This is consistent with our general approach to associative/commutative operators —our heuristics account only for associativity of equivalence— but there is clearly room for improvement.[7]

In general, we are aware that future variants of the heuristics are likely, but the underlying procedures should not require alteration. Therefore, we designed the tactics so that, as much as possible, they were "roughly modular" in terms of their heuristics. For instance, we would not have to rewrite the entire tactic to incorporate a new heuristic for implementing a Leibniz lemma or a new structure for what inferences involving associative/commutative operators (we abbreviate to *AC operators*) are to be handled automatically. Instead, we would replace one ML procedure by another; assuming the interface between procedures remained the same, there would be no need for further change.

Carrying out Leibniz substitutions

Our tactic for carrying out Leibniz substitutions —a part of the process of verifying the correctness of a guess for instantiating a Leibniz lemma— rewrites only substitutions that replace the particular fresh variable generated to be the Leibniz substitution variable. It cannot simply rewrite *all* textual substitutions in the Leibniz substitution according to these lemmas. If it did, it would try to rewrite away the textual substitution operations that we intend to keep in our example proof step (first step in Change of Dummy proof). We kept the Leibniz substitution tactic focused.

In Table 6.10, no lemma addresses the issue of rewriting a Leibniz substitution through another textual substitution operator. The textbook [GS93b]

[7]Our not including automatic reasoning about symmetry of equivalence does not restrict the class of theorems we can prove. We would, however, need to make such symmetry reasoning explicit, instead of making it implicit in the tactics. This is consistent with the behavior of a novice student; in teaching from [GS93b], we encourage some students to make even simple symmetry inferences explicit until they are comfortable leaving them implicit in other, bigger inferences.

does not directly address whether such a move is even permissible, and we did not attempt to accommodate it. Doing so would have involved a more sophisticated treatment of textual substitution and would not have added significantly to our understanding of calculational logic inference.

One of our Leibniz lemmas is technically redundant: lemma (6.41) can be derived from lemma (6.42). (Both lemmas are given on page 105.) We keep all three lemmas in our system to avoid doing Leibniz substitution through quantifications. Without lemma (6.41), we would need to substitute into a quantifier expression, which could result in bound variable changes. By passing substitution inside the quantifier in the inference rule instead of doing it as a Leibniz rewrite, we avoid the need to reason about changes of bound variables.

Obvious inference and AC-operators

Our tactics automatically work modulo symmetry of equality in the context of matching to solve object-theoremhood proof goals; that is, we automatically prove $(\epsilon)\ \alpha \vdash A = B$ from $(\epsilon)\ \alpha \vdash B = A$. As previously mentioned, the only associativity/commutativity property we accommodate is associativity of $\equiv$, but extensions of our system will also need to handle properties for other AC-operators.

Consider a proof goal $(\epsilon)\ \alpha \vdash P = Q$ where P and Q are equal terms modulo some notion of associativity and commutativity of operators. Assuming correct typing, we consider the theoremhood of $P = Q$ to be obvious; as described below, we automated that inference as part of `Leib_tac`. We do not explicitly try to prove it from the provided premise —in fact, any premise would suffice.[8]

We automated this obvious *Othm*-inference using an ML predicate to determine term equality modulo some notion of what properties of AC-operators are obvious. If that predicate determines that P and Q are term-equal, we backchain through the following lemma:

```
(6.58) ∀A,B:OE, ∈:(OV⟶OTS), α:OE List.
       equal_mod_AC_props(A;B) & (∈) [A = B] :: bool &
       (∈) α :: bool
       ⇒
       (∈) α ⊢ A = B
```

The equality determination of the ML predicate is represented by the Nuprl relation `equal_mod_AC_props`; if it holds for P and Q (and if necessary expressions are type correct), our tactic will prove the object theoremhood goal.

[8]To see why any premise would suffice, instantiate the Leibniz substitutions with a substitution body in which the fresh Leibniz variable does not occur, so components of the premise do not get substituted into the conclusion.

Just as our ML predicate is intended to generally represent the properties of AC-operators that are considered obvious in a given student model, our relation `equal_mod_AC_props` is intended to be general, to match that ML predicate. For example, as of this writing, both represent only the property of associativity of $\equiv$. To accommodate different notions of which AC-properties are obvious, we would have to alter the ML predicate and any lemmas or other properties relating it to `equal_mod_AC_props`. Lemma (6.58) above would remain unchanged, however, as would the tactic that backchains through it.

To simplify implementation, we declared `equal_mod_AC_props` as a primitive Nuprl operator instead of defining it to directly encode equality modulo some AC-properties. Correspondingly, there is no definition to use in proving the `equal_mod_AC_props` subgoal that results from the backchaining; we implemented a tactic that essentially invokes `Fiat` to handle that subgoal.[9] Had we wished to be more complete, we could have implemented a definition of `equal_mod_AC_props` and a tactic that used that definition for a full proof, but the additional labor would not have have significantly improved our cognitive model. Our ML predicate assures us that P and Q are term equal; we do not prove it. In this way, we represent the ML determination of term-equality in our tactics. By not applying the major machinery of `Leib_tac` to the original subgoal, we represent the equality of P and Q as obvious with respect to Leibniz reasoning.

Our handling of AC-operators also serves as an illustration of the extent to which we succeeded in maintaining modularity in our implementation of calculational logic inference. If we were to change our predicate of term-equality modulo associativity and commutativity, `Leib_tac` would expand its notion of obvious theoremhood accordingly without having to rewrite or alter any tactic; in this way, `Leib_tac` is modular. If we changed the predicate to account for symmetry of $\equiv$ and we wanted our procedure for instantiating Leibniz lemmas to change accordingly, however, we would need to re-code the instantiation/guessing procedure.

6.8 Formalizing a Full Calculational Proof

6.8.1 Inference rule Transitivity

To show how `Leib_tac` works in the context of a full calculational proof, we formalized the entire proof of Theorem Change of Dummy in Nuprl. This required formalizing not just Leibniz inference but Transitivity inference as well.

[9] We use `Fiat` frequently to admit lemmas without formal proof, but this is the only place in our tactics where we directly call `Fiat`.

Recall that the proof of Change of Dummy is a chain of equalities. Inference rule Transitivity allows the desired conclusion from those component equalities. Here is the Gries/Schneider representation of Transitivity:

$$(6.59) \quad \frac{X = Y, Y = Z}{X = Z}$$

As with the representations of Leibniz in [GS93b], the expression $X = Y$ is a statement of object-level theoremhood. Thus, the transitivity captured in this rule is the transitivity of object-level equality in object theorems. We capture this in the Nuprl theorem on which we base our Transitivity tactic:

(6.60)
```
∀p,p1:OE, ep:(OV⟶OTS), a:OE List, q:OE.
(ep) a ⊢ p = p1 & (ep) a ⊢ p1 = q ⇒ (ep) a ⊢ p = q
```

Inference rule Transitivity is typically used in a generalized form, justifying a conclusion based on an arbitrary number of equalities, not just two. In our calculational logic transitivity tactic `CL_trans_tac`, we capture that standard usage. `CL_trans_tac` takes a term list that specifies the intermediate steps desired for a calculational proof of an object-level equality, and it results in subgoals corresponding to the intermediate equalities. This small example illustrates its usage:

(6.61)
```
1. A : OE
2. B : OE
3. C : OE
4. D : OE
5. ∈ : OV⟶OTS
6. α : OE List
⊢  (∈) α ⊢ A = D  by  CL_trans_tac Tms:[B ; C] ...w

  \
  ⊢  (∈) α ⊢ A = B  by  <TACTIC>
  ---
  ⊢  (∈) α ⊢ B = C  by  <TACTIC>
  ---
  ⊢  (∈) α ⊢ C = D  by  <TACTIC>
```

The implementation of `CL_trans_tac` is straightforward, recursively instantiating theorem (6.60) until the term list argument has been exhausted.

6.8.2 The formalized Change of Dummy proof

Figure 6.5 contains our (several page long) Nuprl formalization of the Change of Dummy proof in [GS93b].

```
1. x : OV
2. y : OV
3. f : OV
4. f⁻ : OV
5. α : OE List
6. R : OE
7. P : OE
8. ∈ : OV ⟶ OTS
9. b : Qind
10. (∀ [x; y] |t_o:x = f(y) ≡ y = f⁻(x)) onlist(OE) α
11. (∈[[x] ⟵ ind]) [R; P] :: bool
12. (∈) α :: bool
13. ¬(some [y] occur free in [R; P])
14. ¬x = y
15. ¬y = f
16. ¬x = f
17. ¬y = f⁻
18. ¬(some [x; y] occur free in α)
19. (∈[[x; y] ⟵ ind]) [f; f⁻] :: ind(1)
⊢  (∈) α ⊢ (*_b [y] | (R)[x := f(y)] : (P)[x := f(y)])
           =
           (*_b [x] | R : P)

 by  CL_trans_tac
     Tms:[(*_b [y] | (R)[x := f(y)] :
               (*_b [x] | x = f(y) : P)) ;
          (*_b [x; y] | (R)[x := f(y)] ∧ x = f(y) : P) ;
          (*_b [x; y] | (R)[x := x] ∧ x = f(y) : P) ;
          (*_b [x; y] | R ∧ x = f(y) : P) ;
          (*_b [x] | R : (*_b [y] | x = f(y) : P)) ;
          (*_b [x] | R : (*_b [y] | y = f⁻(x) : P)) ;
          (*_b [x] | R : (P)[y := f⁻(x)])] ...w
```

Figure 6.5: Nuprl formalization of Change of Dummy proof. This first page shows the application of our calculational logic transitivity tactic. The figure is continued on the following pages, showing the resulting subgoals and the applications of `Leib_tac` that solve them.

Figure 6.5 (Continued)

```
\
⊢  (∈) α  ⊢ (*_b [y] | (R)[x := f(y)] : (P)[x := f(y)])
          =
          (*_b [y] | (R)[x := f(y)] :
               (*_b [x] | x = f(y) : P))

 by  Leib_tac
     Thm* ∀x:OV, E:OE, α:OE List, ∈:(OV ⟶ OTS), b:Qind
          , P:OE.
          ¬(some [x] occur free in [E]) &
          (∈) α :: bool & (∈) [E] :: ind &
          (∈[x ⟵ ind]) [P] :: bool
          ⇒
          (∈) α ⊢ (*_b [x] | x = E : P) = (P)[x := E]
     Tms:[x ; f(y) ; α ; ∈[[y] ⟵ ind] ; b ; P] ...w
---
⊢  (∈) α  ⊢ (*_b [y] | (R)[x := f(y)] :
               (*_b [x] | x = f(y) : P))
          =
          (*_b [x; y] | (R)[x := f(y)] ∧ x = f(y) : P)

 by  Leib_tac
     Thm* ∀y:OV, R:OE, α:OE List, ∈:(OV ⟶ OTS), P,Q:OE
          , x:OV, b:Qind.
          ¬(some [x] occur free in [R]) &
          (∈) α :: bool &
          (∈[[x; y] ⟵ ind]) [P; Q; R] :: bool
          ⇒
          (∈) α ⊢ (*_b [x; y] | R ∧ Q : P)
                =
                (*_b [y] | R : (*_b [x] | Q : P))
     Tms:[y ; (R)[x := f(y)] ; α ; ∈ ; P ;
          x = f(y) ; x ; b] ...w
---
```

Figure 6.5 (Continued)

```
⊢  (∈) α ⊢ (*_b [x; y] | (R)[x := f(y)] ∧ x = f(y) : P)
         =
         (*_b [x; y] | (R)[x := x] ∧ x = f(y) : P)
 by  Leib_tac
     Thm* ∀ α:OE List, ∈:(OV → OTS), e,f:OE, r:{ind, bool}
          , E:OE, z:OV.
          (∈) α :: bool &
          (∈) [e; f] :: r &
          (∈[[z] ← r]) [E] :: bool
          ⇒
          (∈) α ⊢ (E)[z := f] ∧ e = f
                 =
                 (E)[z := e] ∧ e = f
     Tms:[α ; ∈[[x; y] ← ind] ; x ; f(y) ;
          ind ; R ; x] ...w
---
⊢  (∈) α ⊢ (*_b [x; y] | (R)[x := x] ∧ x = f(y) : P)
         =
         (*_b [x; y] | R ∧ x = f(y) : P)
 by  Leib_tac
     Thm* ∀ ∈:(OV → OTS), α:OE List, E:OE, x:OV.
          (∈) α ⊢ (E)[x := x] = E
     Tms:[∈[[x; y] ← ind] ; α ; R ; x] ...w
---
⊢  (∈) α ⊢ (*_b [x; y] | R ∧ x = f(y) : P)
         =
         (*_b [x] | R : (*_b [y] | x = f(y) : P))
 by  Leib_tac
     Thm* ∀y:OV, R:OE, α:OE List, ∈:(OV → OTS), P,Q:OE
          , x:OV, b:Qind.
          ¬(some [y] occur free in [R]) &
          (∈) α :: bool &
          (∈[[x; y] ← ind]) [P; Q; R] :: bool
          ⇒
          (∈) α ⊢ (*_b [x; y] | R ∧ Q : P)
                 =
                 (*_b [x] | R : (*_b [y] | Q : P))
     Tms:[y ; R ; α ; ∈ ; P ; x = f(y) ; x ; b] ...w
---
```

Figure 6.5 (Continued)

```
⊢  (∈) α  ⊢ (*_b [x] | R : (*_b [y] | x = f(y) : P))
           =
           (*_b [x] | R : (*_b [y] | y = f⁻(x) : P))

 by  Leib_tac
     Thm* ∀x,y,f,f⁻:OV, α:OE List, ∈:(OV⟶OTS).
          (∀ [x
             ; y] |t_o:x = f(y) ≡ ...) onlist(OE) α &
          (∈) α :: bool &
          ¬(some [x; y] occur free in α)
          ⇒
          (∈[[x; y]⟵ ind]) α ⊢ x = f(y) = y = f⁻(x)
     Tms:[x ; y ; f ; f⁻ ; α ; ∈] ...w
---
⊢  (∈) α  ⊢ (*_b [x] | R : (*_b [y] | y = f⁻(x) : P))
           =
           (*_b [x] | R : (P)[y := f⁻(x)])

 by  Leib_tac
     Thm* ∀x:OV, E:OE, α:OE List, ∈:(OV⟶OTS), b:Qind
          , P:OE.
          ¬(some [x] occur free in [E]) &
          (∈) α :: bool & (∈) [E] :: ind &
          (∈[x⟵ ind]) [P] :: bool
          ⇒
          (∈) α ⊢ (*_b [x] | x = E : P) = (P)[x := E]
     Tms:[y ; f⁻(x) ; α ; ∈[[x]⟵ ind] ; b ; P] ...w
---
⊢  (∈) α  ⊢ (*_b [x] | R : (P)[y := f⁻(x)])
           =
           (*_b [x] | R : P)

 by  Leib_tac
     Thm* ∀ α:OE List, ∈:(OV⟶OTS), P:OE, r:{ind, bool}
          , x:OV, E:OE.
          (∈) α :: bool & (∈) [P] :: r &
          ¬(some [x] occur free in [P])
          ⇒
          (∈) α ⊢ (P)[x := E] = P
     Tms:[α ; ∈[x⟵ ind] ; P ; bool ; y ; f⁻(x)] ...w
```

Note that our formalization has eight component equalities, instead of the seven in the presentation in [GS93b]. This is because the fourth step in the Gries/Schneider presentation is actually a composition of two separate Leibniz steps.

(6.62)
$$\begin{array}{ll} & (\star x, y \mid R[x := x] \wedge x = f.y : P) \\ = & \langle R[x := x] \;\equiv\; R;\ \text{Nesting},\ \neg occurs(\text{“}y\text{”},\text{“}R\text{”}) \\ & \quad \text{—Now we can get a quantification in } x \text{ alone.}\rangle \\ & (\star x \mid R : (\star y \mid x = f.y : P)) \end{array}$$

Our formalization represents each of the Leibniz steps separately, making both subgoals explicit.

Each of the eight subgoals in Figure 6.5 is solved by a single application of `Leib_tac`, demonstrating its adequacy for the calculational inferences necessary for this proof.

6.9 Conclusion

This chapter describes our tactic-based explanation of calculational logic inference and, more generally, our approach to formalizing inference methods. We identified the essential kinds of propositions upon which calculational logic is built —OV-inequalities, not-occur propositions, OE-typing judgments, and object-theoremhood judgments— and created tactics to carry out the most common, important inferences in proving each kind of proposition.[10] In `Leib_tac`, we combined all these component tactics into a single, complex, and often modular explanation of the key inferences in calculational logic.

We have not attempted to list *all* the ways in which our implementation is incomplete or otherwise simplified. Our system is a work in progress, and an exhaustive list of incompletenesses and simplifications is beyond the scope of this project. Any omissions, however, neither inhibit the ability of our system to express and prove calculational logic assertions nor detract significantly from the value of our tactics as an explanation of calculational predicate logic.

It is not surprising that [GS93b] did not give a detailed description of all the simple manipulations that people might make. Its goal is to teach, not to overwhelm with detail, and leaving many details implicit makes it successful. It uses some simplifying conventions —e.g., the one whereby different letters of the alphabet stand for different variables— without explicitly mentioning

[10] On rare occasions, list membership propositions expressed using the data language `onlist` form also occurred in our calculational logic proofs. We simply used Nuprl tactic `Hypothesis` to prove them. We mention this to complete our account of all the data language propositions that occurred; it merits no further discussion.

them, but its audience generally understands the text anyway, without needing to be burdened by such details. This is an example of how our formalized implementation highlights points of pedagogy. Indeed, we believe that the ways in which our system is more explicitly detailed than [GS93b] and the ways in which we have accomplished so much with such incomplete heuristics may correspond to pedagogical insights. We discuss this further in chapter 7.

Students using [GS93b] may ask for help from a teacher or other expert in doing a calculational proof. In essence, they express that they are unable to successfully accomplish a certain task and request guidance from outside. This corresponds to an implementation functionality that we have not yet developed but which might be promising for our system: our tactics might output informative error messages to improve interactivity with users. We envision a system in which the failure of a tactic may be accompanied by a request to the user for assistance. For instance, if the tactic could not correctly instantiate a Leibniz lemma, it could ask the user to provide a full list of terms to use for the instantiation. Such extensions to our system would improve it in two ways, making it a better pedagogical partner as well as an interactive theorem prover. It would also add a new dimension to its cognitive modeling, treating the inference task as a process that could be influenced by interaction with outsiders.

We began this chapter by emphasizing the fact that our account of calculational logic inference differed from the one in the book that first described it [GS93b]. We have described many of those those differences, ranging from the high-level inferences that were left omitted from [GS93b] to the low-level details of how to carry out those inferences. The Gries/Schneider text presented a description from which people might learn inference techniques; we presented a possible model of the inference techniques people might learn. It is no wonder our two accounts differ in the details.

Chapter 7

Observations on Pedagogy

> This text attempts to change the way we teach logic to beginning students. Instead of teaching logic as a subject in isolation, we regard it as a basic tool and show how to use it. We strive to give students a skill in the propositional and predicate calculi and then to exercise that skill thoroughly in applications that arise in computer science and discrete mathematics.
>
> We are not logicians, but programming methodologists, and this text reflects that perspective. We are among the first generation of scientists who are more interested in using logic than studying it. With this text, we hope to empower further generations of computer scientists and mathematicians to become serious users of logic.
>
> David Gries & Fred Schneider [GS93b, preface]

7.1 Introduction

The primary goal of our research was to implement a tactic-based model of a method of inference that people actually used. Choosing calculational logic for that central inference method, we worked substantially from the undergraduate-level textbook that presents it, "A Logical Approach To Discrete Math" [GS93b]. Aside from the student modeling that necessarily resulted from our choice, it was not our intent to make pedagogical criticisms of that textbook or otherwise contribute significantly to pedagogy. Nonetheless, our work led to a few pedagogical observations about the text, and we came to believe that our overall approach to modeling could contribute similarly to pedagogical criticism about other methods and other textbooks. We comment on [GS93b] as an illustration of how our approach can help elucidate aspects of pedagogy.

As is common with textbooks, [GS93b] is not completely formalized. It draws from an approach that is actually used in programming methodology research, applying it for the first time to discrete math instruction; in doing so, it attempts to capture the current state of the still-evolving calculational approach to mathematics, which was not completely formalized for this new context. Furthermore, a full formalization might not have been consistent with the authors' stated goals: it might well have overwhelmed or intimidated beginning students.

The differences between our formalized account of calculational logic and the incomplete formalization in the textbook are the source of our pedagogical criticisms. By exposing, identifying, and understanding what is not explicit in the text, one can explicitly discuss aspects of pedagogy that might otherwise remain hidden: whether the authors understand the skill level of their readers; whether the demands of the textbook are appropriate for novice students; whether more detailed instruction should have been given on points that might be considered obvious; whether decisions to omit details from the text were good ones. In the sections that follow, we demonstrate how our formalized modeling approach can improve understanding of issues such as these.

7.2 Implicit Metalogic

The calculational approach to logic emerged over time from research that employs logical techniques to reason about program correctness; [GS93b] reflects this history, mentioning textual substitution and Hoare triples[1] before boolean expressions. The purpose of [GS93b] is to introduce the calculational approach and demonstrate that it is also applicable to discrete mathematics topics such as predicate logic, set theory, and the theory of integers. The authors partially formalized their method of reasoning to make it clearer, more concise, and more syntactic, in keeping with the principles of general calculational methods.

When considering its intended applications (e.g., reasoning about textual substitution and free variable occurrences), it seems sensible that calculational logic can best be understood as a formalized metalogic. Typically, such a metalogic would be used and taught informally. Indeed, many discrete mathematics or logic textbooks have sections containing propositions *about* logical expressions (e.g., typical presentations of theorems about duals of boolean expressions). An informal or vague use of metalogic in textbooks may not be pleasing to strict logicians, but it is an established part of pedagogy. The use of meta-level reasoning in teaching logic is not a novelty of [GS93b].

[1] Hoare triples are a formalism used to reason about program execution (see [GS93b, page 17]). They are not typically elements of a discrete mathematics course.

What *is* unusual about the "Logical Approach" text [GS93b] is that its authors embraced the idea of treating metalogic more formally and applying it to the topics of discrete mathematics. According to student comments reprinted in [GS93a], this metalogical approach to discrete math seemed to help the authors achieve their goal of teaching students to be comfortable with rigorous, syntactic, calculational reasoning, which they stated was one of their primary objectives. They did not, however, explicitly state in [GS93b] that they were teaching and utilizing a metalogic in their approach. (This was the novelty of chapter 3, our recognition and demonstration that the textbook could be explained without semantic oddities as a formalized metalogic.) Further, they did not completely formalize their metalogic in presenting their predicate logic. This was a sensible decision; a full formalization would surely have overwhelmed their novice audience, and the authors could always make their presentation more precise if it became necessary later in the text. Perhaps more importantly, such added detail would not have improved the way the book reflects the ideas stated in its preface (quoted at the beginning of this chapter). The primary goal of the authors was not to teach logic per se but to teach their method of reasoning, and it is much more difficult to explain metalogic than syntactic manipulations.

By formalizing meta-reasoning, the authors extended the scope of their calculational approach without resorting to higher-order logic, combining pedagogically valuable formality with the informal, de facto meta-level reasoning that is common in discrete mathematics texts. Their focus on syntactic, calculational inference and formalized-looking notation was fully in service of their primary goal. Precise formalization and fully detailed explanations were not the point, and in fact were frequently omitted. The significant differences between the account in chapter 3 and [GS93b] are clear indications of this.

Such informality and omissions may well have made the basic syntactic manipulations of the calculational approach more accessible to students. We doubt, however, that students fully grasp the meta-level meanings of the notations; they learn the moves of the calculational approach without fully understanding the subject to which they apply those moves. This can cause difficulties when the moves come under greater scrutiny. As seen in chapter 3, the "formulas" established as theorems by calculational logic cannot coherently be considered boolean formulas in the traditional sense; the metalinguistic intent upon which calculational logic is based prohibits it. Even for formulas that could be read as object-level formulas, object-level meanings are only *indirectly* relevant —through the claim that object theorems are valid. Students may fail to appreciate this.

7.3 Omissions from the Text

As part of formalizing [GS93b], we exposed omissions from the text, identifying and filling those gaps to arrive at a full formalization for our tactic model. We can think of those omitted elements as components that the authors felt readers would already have mastered to the degree required for the immediate applications. That is, the authors felt those components —be they propositions left implicit or inference methods left undescribed— were *obvious*.[2] Recall that [GS93b] is a textbook, not a journal article, and it is not a forum that demands expert inference on the part of its readers. Therefore, the omissions can be construed as part of the authors' theory of *obvious inference* —the kinds of inference that would be obvious to readers and hence not worth explicitly detailing— in the context of the textbook and the applications contained within.

This does not mean that every element of our data language that did not appear in [GS93b] is obviously constructed and used by readers. Their omission, however, indicates that the major propositions and inferences on them must be considered obvious to the authors (according to the understanding they had when they wrote the text). We consider two examples: OV-inequalities and OE-typing.

Proving the inequality of object variables is a particularly important, pervasive aspect of calculational logic inference. Since [GS93b] is not thoroughly explicit in acknowledging its metalogical foundations, it is correspondingly vague in acknowledging the sophistication necessary to prove OV-inequalities; because variables in the text are typically metavariables, the mere fact that variables of type OV are different letters does not mean they are unequal as elements of OV. In proofs presented in [GS93b], however, a simplifying convention is frequently used and essentially un-mentioned: different metavariables are tacitly assumed to stand for different elements of OV. (This is not always true —different metavariables can be instantiated with the same object variable— but it is common enough to be taken as convention.) Students pick up on this convention as obvious (things that look different are different, what could be simpler?) and are given enough examples to understand when and how to apply it. The authors of the text provide no further direction for proving OVs unequal.

Indeed, this seems like the right decision. Including needless detail about such obvious inference is unnecessary and might overwhelm students. More

[2] What if the authors (Gries & Schneider) did not omit such elements because they consciously decided they were obvious but instead omitted them simply because they failed to consider them? In that case, the authors must have understood the subject they were teaching without consciously considering the omitted elements. For those elements that are a part of calculational logic, they were not unused or ignored. They were simply too obvious for conscious consideration.

open to criticism is the convention itself, which reflects the fact that the book is an incomplete formalization. Students often do not realize they are working with a metalogic, much less a metalogic to which this convention applies. Still, the inferences they make are adequate for the tasks at hand.

This bears repeating: although students may not appreciate the logic involved, the inferences they make are sufficient for the immediate applications. If it were important to teach metalogic in a discrete mathematics course, the book could be refined to do so. In the view of the authors of [GS93b], however, no such refinement is necessary, and the obvious, informal, and somewhat oversimplified inference methods the students already make are sufficient.

The case regarding typing of expressions of type *OE* is similar. The authors of [GS93b] explicitly acknowledge types in the book but do not present a formal type-inference method. Clearly, they feel the type inferences necessary for the demands placed on their students are obvious. Throughout the book, types are either explicitly provided or simply inferred; no significant instruction on type inference is provided in the textbook, and for the most part, none is needed. Students understand and perform some kind of obvious type inference without such inference methods being explicitly taught in the textbook.

These areas (solving *OV*-inequalities, establishing *OE*-typing) were salient aspects of calculational inference for which our treatment is complementary to that of [GS93b]. From a cognitive perspective, our model is a significant advance over [GS93b] in previously unelaborated areas like these. Our model has significant modules for both of these kinds of inference, and the textbook provides little or no prescriptive guidance for how the inferences should be carried out. This merits emphasis: our tactics model people, not merely methods given in a book.

As such, some tactics perform inferences that are not explained in [GS93b] but on which people nonetheless succeed. In this way, such tactics can be construed as identifying and explaining obvious inferences (in the context of tasks presented to students using [GS93b]). The correctness of this notion of obvious inference and our model of it are certainly debatable. If we could not identify and model a notion of obvious inference, however, we could not even begin to debate it.

This reflects a significant virtue of our cognitive modeling method, which could potentially contribute in two areas. Clearly, an investigation of which inferences are obvious and which are not is of benefit to the psychology of inference. In addition, a computational model or theory of obvious inference can provide benefits in the computer science field of automated reasoning.[3] For

[3]We refer readers to McAllester's paper [McA91] for an interesting approach to a computational theory of obvious inference. We do not propose our own theory of obvious inferences; we suggest a methodological criterion for identifying inferences implicit in a pedagogical presentation, and we suggest that such inferences may accurately be considered

instance, automated mathematical assistants become significantly more useful when they prevent users from spending time on obvious inferences; indeed, this is the major use of the `Auto` tactic in Nuprl.

7.4 Simple Inference Methods

Any introductory-level textbook should strive for simplicity, taking care not to be too demanding on students. In the context of the calculational approach taught in [GS93b], our focus on developing a detailed cognitive model gives us an atypical perspective on the pedagogical issue of simplicity. In particular, we have a formal, computational perspective on how simple the methods used by students really are, and we can discuss simplicity in a formal, well-specified framework instead of as a vague notion.

As they state in [GS93a], Gries and Schneider want their students to become quickly familiar and comfortable with formal calculational methods. Indeed, from the very beginning of [GS93b], they provide examples and exercises on which students can quickly succeed without learning many complex procedures. The fact that simple inference methods are sufficient for the book's full treatment of calculational predicate logic, however, was not initially apparent. Our modeling approach permits us to now discuss in detail the ways in which the pedagogical approach of [GS93b] permits students to succeed after learning only a few simple inference methods.

As previously mentioned, [GS93b] does not fully describe or formalize all the inference methods actually used by students. In contrast, our tactic model exposes a collection of inference methods sufficient for understanding a significant body of calculational logic proofs in minute detail. This formalization led to an unexpected result: there are not many such methods in our tactic model. We had not anticipated that so many of the examples in [GS93b] —including the steps of the Change of Dummy proof— could be explained with only six basic kinds of inference: list membership judgments, solving OV-inequalities, free variable judgments, OE-typing, object theoremhood judgments, and Leibniz inference.[4] At the user level, we did not need to directly reason about constructs like object type expressions or type assignment updates in much detail, but there was no *a priori* reason to believe this would be unnecessary. The fact that so few user-level inference forms are sufficient is an indication of the simplicity and accessibility of the pedagogical approach of [GS93b], one that might not be clear without a formalized model on which to base such an observation.

"obvious".

[4] Leibniz inference does establish object theoremhood, but because it has so much unique detail, we consider it separately from the other methods we implemented for object theoremhood judgments.

Our tactics that model the major user-level inference forms embody only a few assumptions and simple machinations; the fact that such simple methods were sufficient was another unexpected result. For example, after general-purpose pre-processing, our tactics prove OV-inequalities by straightforward matching against either an OV-inequality hypothesis or a free variable not-occurrence proposition; as described in chapter 6, the strategies involved are direct and simple in both cases. The examples in the book might have required far more sophisticated techniques, perhaps involving reasoning about type expressions, but none were needed.

This observation that simple methods suffice also applies to the other major user-level inference forms. In our model, it even extends beyond tactics to the simplicity of the type guessing and Leibniz instantiation heuristics that we demonstrated were sufficient for the examples in the book. All of this further supports our observation that [GS93b] is designed to facilitate quick success from the students who use it. We found it genuinely interesting and somewhat surprising that tactics as simple as ours were so effective on the material in the book.

Our perspective as model developers also brought another specific instance of simplification to light. There are three inference rules called Leibniz in [GS93b]. (Their Gries/Schneider representations are in Table 6.8; for convenience, we repeat them here.)

$$\frac{P = Q}{E[z := P] = E[z := Q]} \tag{7.1}$$

$$\frac{P = Q}{(\star x \mid E[z := P] : S) = (\star x \mid E[z := Q] : S)} \tag{7.2}$$

$$\frac{R \Rightarrow P = Q}{(\star x \mid R : E[z := P]) = (\star x \mid R : E[z := Q])} \tag{7.3}$$

From a logic-based perspective, one of them is redundant: rule (7.2) follows from rule (7.3). Indeed, we were tempted to streamline our computational model by including only two Leibniz rules, omitting the redundant one. We did not, however, because we found a significant computational and cognitive reason to keep all three: rule (7.2) permits Leibniz instantiation into the range of a quantifier expression without requiring a change of bound variables (and the accompanying alpha-equality reasoning). Were it not for this rule, proofs would be significantly more complicated, with every Leibniz instantiation into a quantifier range requiring the use of some alpha-equality rule as well. This would be both a computational burden for our model and a cognitive burden for students who might not be comfortable with alpha equality yet. Thus, the reason we elected to include that “extra” Leibniz rule corresponds to yet another pedagogical simplification in [GS93b].

Overall, our tactic-based model was unexpectedly rewarding from the perspective of reflecting the pedagogically motivated simplicity in [GS93b]. One of the compelling features of tactics in any context, not just this one, is that they can capture high-level, even human-level inferences. We see the simplicity of our tactic model as reflecting the design of [GS93b] to teach without being too demanding on students; we modeled the cognitive inferences of a student who has not needed to learn much extraneous methodology to achieve success at calculational proof. This correspondence between the goal of the textbook [GS93b] and our tactic model is satisfying and encouraging.

Lastly, analogous to a virtue of our method in the context of obvious inference, our model provides a detailed basis for discussing the role of simplicity in the pedagogy of [GS93b] that might not otherwise exist. It is not necessary to discuss pedagogical simplifications abstractly, without concrete elements to which one can point for examples. With the framework that results from our modeling approach, we can present concrete areas for discussion; we can list particular inference methods before asking if they are the right ones to teach or if more sophisticated methods should be included. Once again, it is not our primary goal to opine on this topic of pedagogy (although it does seem that the goal of Gries and Schneider of allowing their students early, encouraging success is a good one). We instead wish to emphasize that the increased understanding of calculational logic that emerges from our modeling method can support contributions to the discussion of pedagogy.

7.5 Concluding/General Remarks

As a basis for understanding the philosophy of Gries and Schneider and for evaluating the effectiveness of their book, we present here some of their pedagogical goals, taken from points of emphasis in the teachers’ guide [GS93a].

- Demonstrate that the calculational approach to logic is useful.
- Instill confidence in syntactic manipulation.
- Dispel the fear of mathematics.
- Provide quick success with the new approach.

In some ways, our formalization and model suggest that the authors have indeed achieved their goals. They apply logical methods in an interesting way, presenting an incompletely formalized metalogic that provides a framework for studying a range of topics in discrete mathematics. Their approach is consistent with the ideas behind calculational methods in general; the major user-level inference forms would be considered syntax-based judgments (OV-inequality, free variable occurrence judgments, etc.) by the programming methodology research community from which calculational methods emerged, and they can all be performed by relying heavily on syntax-oriented problem solving methods. Perhaps most importantly from the perspective of pedagogy, the authors help ensure that students quickly gain comfort with this style of mathematics. They have presented an effectively calculational approach, and they seem to effectively teach that approach.[5]

As an incomplete formalization of a (meta-)logical method, however, the textbook has significant problems that became clear under the scrutiny required by our approach to modeling. The concise presentation in the book may not pay adequate attention to typing concerns. It may acclimate students to inaccuracies in logic, which may not be a pedagogical virtue. Further, it may prevent students from becoming comfortable working outside of a metalogical framework when they are called upon to grasp object logics. For example, students may not understand why they cannot use textual substitution as part of an object logic.

Indeed, the methods taught seem to suffice for the examples in the textbook, but effective texts teach skills useful beyond their immediate context. The hidden difficulties in [GS93b] make it less effective in this sense: it seems successful in teaching the calculational approach and comfort with syntactic manipulation but less successful in actually teaching logic. Certainly, a fully formalized metalogic would overwhelm novice students, and it is not required to make the book effective as a pedagogical tool; in its informality, the approach in [GS93b] is similar to that taken in established texts such as [HC68]. Nonetheless, in some ways, "A Logical Approach…" might be even more effective if it explicitly introduced the ideas of object logic and metalogic as part of its pedagogy.

We understand that the calculational approach to discrete mathematics as presented in [GS93b] was simply a snapshot of the authors' understanding of a still-developing method. It was a method that they had found successful in actual research, and they saw that it could be valuable to students learning discrete mathematics. We do not feel that our work is adequate grounds for a full critique of their approach. We simply feel that it may add something to the discussion.

As a final remark on our unusual convergence of education and computational formalized mathematics, we revisit the suggestion that our general modeling approach could provide analogous pedagogical insights if applied to a different textbook. By no means is one example a conclusive demonstration of broad applicability, but we believe that the particular insights presented

[5] See the student comments reprinted in [GS93a] for further support.

here can be interpreted in a larger context as support for the effectiveness of a general technique. Considering the details of this chapter, it does not appear that there is anything unique about calculational logic that permits formalized inference modeling to support pedagogical insights. Suppose, for instance, that the same overall approach had been applied to a logic with different inference rules: certainly, the specific pedagogical results might be different (e.g., there might not be a “redundant” Leibniz rule), but analyzing the differences between a formalized model and an informal textbook could still yield interesting insights (e.g., relative simplicity of inference methods, whether a textbook is sufficiently explicit in teaching its subject matter). We believe that detailed, algorithmic modeling of inference methods people actually use can provide benefits not only to psychology and computer science but to critical study of pedagogy as well.

Chapter 8

Insight via Eye Movements

8.1 Introduction

As part of developing our cognitive model of calculational logic,[1] we are interested in applications of psychology to theorem proving, an interface largely unexplored by past research. Frequently, the psychology of inference has focused on very specific tasks, such as categorical syllogisms or card selection tasks (e.g., [WJL72]), or on broad ranges of behavior that have led to correspondingly broad projects such as the SOAR cognitive architecture [New90]. Some important models of logical inference fall between these two extremes, such as PSYCOP [Rip94] and the logic-oriented portion of the Mental Models theory from Johnson-Laird [JL83], but they are not truly models of theorem proving.

Some research, however, has applied psychological methods to study theorem proving, such as the work of Melis [Mel94] on deriving useful information for automated theorem provers by analyzing mathematicians' written proofs and their verbal recall of the proving process. We take a different approach, exploring the moment-by-moment computations that subserve theorem proving. To truly reflect and simulate human cognitive behavior, we must consider not only *macrocognition* on the level of "What are people writing?" but also *microcognition*[2] such as "What are people thinking when (or before) they write?" or, more specifically, "Where is attention directed while people search to decide the next proof step?"

By recording and analyzing the eye movements of people while they constructed calculational proofs, we investigated some of our microcognitive ques-

[1] Readers of this chapter should be familiar with the introduction to calculational logic in chapter 2.

[2] We adapt the notion of microcognition —the microstructure of cognition— from Rumelhart, McClelland, et al.[RM + 86]. Roughly, in their context of distributed representations, microcognitive structures correspond to components of a larger cognitive structure.

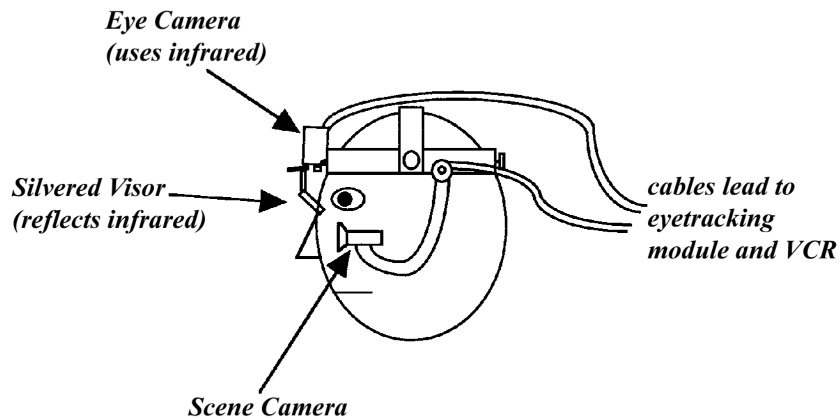


Figure 8.1: Diagram of the ISCAN headmounted eyetracker

tions. Although untested in the area of theorem proving, eyetracking methods have helped experimenters in many fields of study glean unique insight into information processing in general and into the real-time microcognition of inference in particular. (For recent demonstrations of the wide applicability of this methodology, see [BHP95, ES96, HMG92, TSKES95].)

We began our explorations with some primary questions about the process of theorem proving, particularly focusing on *attention during search*, i.e., the extent of attention paid to various features during visual search. Our experiments uncovered information of practical importance for a cognitive model, verifying expected answers —passing a baseline test, in a way, that eyetracking results are sensible in this context— as well as pointing to some more surprising behavior and further interesting questions for future study.

8.2 Method

In our experiment, participants performed calculational proof tasks, unaided, while we recorded their eye movements. A typical calculational proof task for a student involves a statement to be proved, a list of premises (i.e., axioms and previously proved theorems), and whatever procedural knowledge the student brings to the task. Students are generally given the standard premise list (a theorem list from the back of the text) and are not required to memorize it. The theorems are listed in the order of their presentation in the text and are thus roughly grouped by concept, with simpler theorems for a concept often presented first. By using heuristics and principles elucidated in [GS93b], including pattern matching with the premise list, students decide how to proceed in constructing their proofs.

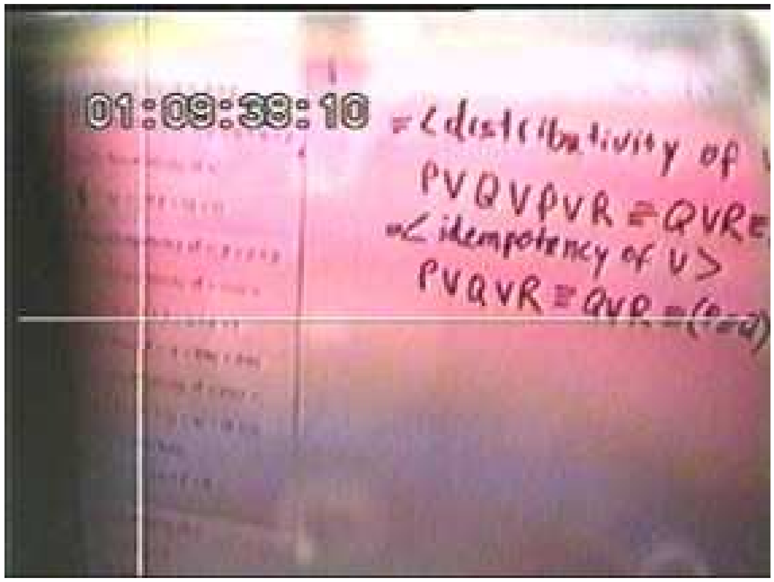
Eye movements were monitored by an ISCAN eyetracker mounted on top of a lightweight headband (Figure 8.1). The camera provided an infrared image of the left eye sampled at 60 Hz. The center of the pupil and the corneal reflection were tracked to determine the orbit of the eye relative to the head. A scene camera, yoked with the view of the tracked eye, provided an image of the subject's field of view. Gaze position (indicated by crosshairs) was superimposed over the scene camera image and recorded onto a Hi8 VCR with 30 Hz frame-by-frame playback. Accuracy of the gaze position record was about one degree of visual angle over a range of +/- 25 degrees. For purposes of determining *fixations* —instances where a participant's recorded glance on an object lasted long enough to indicate significant attention to that object and not an insignificant or random eye placement— we used a threshold of roughly 200ms, or six frames of video playback.

Participants were 15 Cornell students who had completed a course for which the Gries & Schneider book [GS93b] was the primary text. Group A (n=8) was given problem 1(a) as their first problem, and group B (n=7) was given 1(b). (For some results, this distinction was not relevant, and we considered the 15 subjects as a group, undivided by this condition.) The other four problems given to participants to prove are listed here as numbers 2-5.

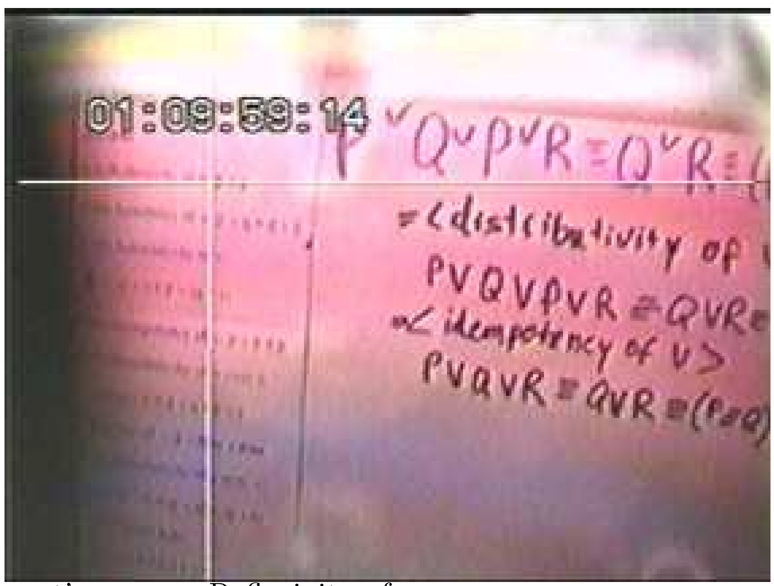
1(a). $P \equiv P \equiv Q \equiv Q \equiv \textbf{true}$
(b). $P \equiv P \equiv Q \equiv Q \equiv \textbf{true} \lor P \lor Q$
2. $P \lor Q \lor P \lor R \equiv Q \lor R \equiv (P \equiv Q) \lor (Q \lor R)$
3. $P \lor (P \equiv P \lor Q) \equiv P \lor Q \equiv P$
4. $P \lor (P \equiv Q \lor Q \equiv P \lor Q) \equiv P \lor P$
5. $\textbf{true} \land Q \equiv \textbf{true} \lor Q \equiv Q$

Before wearing the eyetracking equipment, subjects had the option of working through a warm-up exercise, a proof with no significant overlap with features central to the studies. If needed, reminders were given about points of general technique, such as the order of precedence of operators and definitions of terminology. When subjects were ready, they sat in front of a whiteboard with a premise list on its left (see Figure 8.2), and the eyetracking gear was calibrated for them. They were then read brief instructions: They would be given five statements (one at a time) to prove, all the necessary premises were on the list to their left, and they should let the experimenter know when they were done with a proof.

The experimenter presented the five statements to subjects by writing them on a whiteboard, erased the board when subjects indicated that they were ready to proceed, and did not answer any questions while the experiment was in progress. This is consistent with standard practice on exams about this material, in an attempt to recreate natural conditions as much as possible. The experiments were videotaped and later analyzed for data collection. No audio was considered.



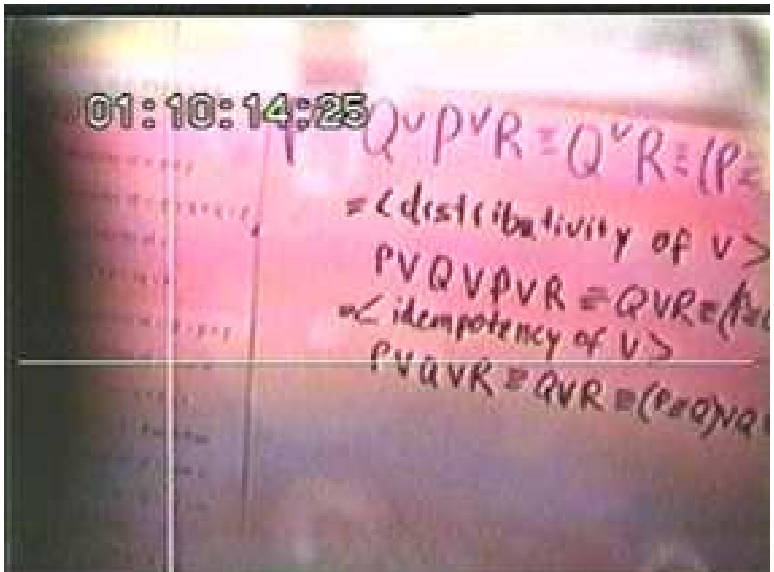
(a) Participant's gaze on Distributivity of $\vee$ over $\equiv$



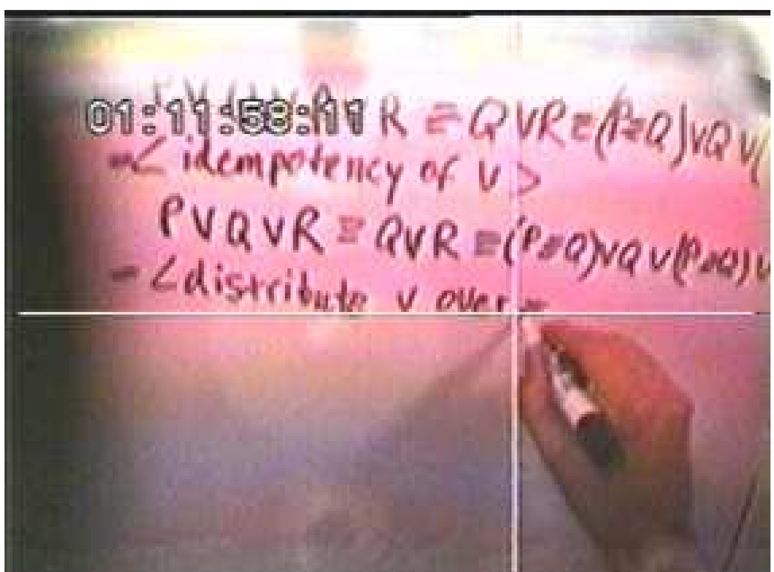
(b) Participant's gaze on Reflexivity of $\equiv$

Figure 8.2: Videotaped images from the headmounted eyetracker. The participant is working on a proof he has already started, deciding what premise to use next. He looks at the one he will eventually use (Distributivity of $\vee$ over $\equiv$), looks at another possibility, returns to Distributivity, and finally writes his choice. The white crosshairs indicate the particular place on the premise list (the list on the left, visible in images a, b, and c) or the workspace (the whiteboard on which he writes in image d) at which the participant is looking at the moment indicated by the timestamp.

Figure 8.2 (Continued)



(c) Participant's gaze returns to Distributivity of $\vee$ over $\equiv$



(d) Participant writes Distributivity of $\vee$ over $\equiv$ in the proof step

(3.2) Symmetry of $\equiv$: $p \equiv q \equiv q \equiv p$

(3.3) Identity of $\equiv$: $\textbf{true} \equiv q \equiv q$

(3.4) **true**

(3.5) Reflexivity of $\equiv$: $p \equiv p$

(3.24) Symmetry of $\vee$: $p \vee q \equiv q \vee p$

(3.25) Associativity of $\vee$:

$(p \vee q) \vee r \equiv p \vee (q \vee r)$

(3.26) Idempotency of $\vee$: $p \vee p \equiv p$

(3.27) Distributivity of $\vee$ over $\equiv$:

$p \vee (q \equiv r) \equiv p \vee q \equiv p \vee r$

(3.29) Zero of $\vee$: $p \vee \textbf{true} \equiv \textbf{true}$

(3.31) Distributivity of $\vee$ over $\vee$:

$p \vee (q \vee r) \equiv (p \vee q) \vee (p \vee r)$

(3.35) Golden Rule:

$p \wedge q \equiv p \equiv q \equiv p \vee q$

(3.36) Symmetry of $\wedge$: $p \wedge q \equiv q \wedge p$

(3.38) Idempotency of $\wedge$: $p \wedge p \equiv p$

(3.39) Identity of $\wedge$: $p \wedge \textbf{true} \equiv p$

(3.43a) Absorption: $p \wedge (p \vee q) \equiv p$

(3.43b) Absorption: $p \vee (p \wedge q) \equiv p$

(3.49) $p \wedge (q \equiv r) \equiv p \wedge q \equiv p \wedge r \equiv p$

(3.50) $p \wedge (q \equiv p) \equiv p \wedge q$

Figure 8.3: List of premises, as presented to participants

The premise list given to them (Figure 8.3) contained 18 theorems taken from the reference premise list from [GS93b], presented and labeled exactly as written in that textbook. The given list consisted of theorems about the three logical operators that appeared in the statements to be proved, presented in the order in which they appeared in the text, effectively but not explicitly grouped into theorems about $\equiv$ (equivalence, labeled by numbers (3.2-3.5)), $\vee$ (disjunction, labeled (3.24-3.31)), and $\wedge$ (conjunction, labeled (3.35-3.50)). Theorems in a group mentioned only the operator used to classify them and operators that appeared in preceding groups. There was nothing to lead subjects to divide the premises into groups; the spacing between theorems was uniform and no verbal cues were given. The list was adequate to prove the five statements that subjects were given. Although (as teachers know) it is possible to use almost any premise in an atypically creative proof, we expected that some of the theorems on the list would serve as distracters, not useful in any of the five proofs.

8.3 Experiment 1

Clearly, different problems result in different solutions on the level of macrorepresentations (e.g., full proofs). We are also interested in microrepresentations (e.g., syntactic features considered while constructing a proof), however, so we examined whether different initial problems resulted in different initial microcognitive behavior.

We compared groups A and B to determine whether the simple but nontrivial addition of an operator to the initial problem statement would affect the subjects' initial search of the premise list or their choice of initial step in that proof. We compared the number of fixations on Zero of $\vee$ (3.29), a theorem useful to group B but not group A, and on disjunction-related theorems in general (3.24-3.31) by members of each group.

8.3.1 Results

In group A, none of the eight subjects (0%) made recorded fixations on theorem (3.29) when considering their first step in the proof. In contrast, in group B, four of the seven subjects (57%) fixated theorem (3.29). We find similar results when considering all the disjunction theorems (3.24-3.31). In group A, three out of eight subjects (38%) fixated one or more of the disjunction theorems during their initial search. In group B, all seven subjects (100%) fixated one or more disjunction theorems during their initial search.

Notably, this did not result in a greatly increased *usage* of disjunction-oriented premises in the first step of the proofs of group B. Only one person in group B made an initial step that involved the added operator; the rest used an equivalence-oriented theorem, a step as viable for group A as for group B.

8.3.2 Discussion

The added operator did significantly alter subjects' attention during initial search, empirically answering a fundamental question: we cannot simply assume solvers will weigh all features the same independent of the particular problem; a cognitive model must take the features of the current problem into account each time. While this may not seem surprising, reflecting problem-solving skills we expect the students to have, it is important to understand the detail in which our observations support this answer. On different problems, students do not *pay attention* to the same features and then *use* them differently, resulting in a different proof. Instead, students actually attend to different features. This is an example of the sort of distinction that can be made only on the microcognitive level.

The observation that these differences did not lead to different choices for a first proof step also confirms an important assumption behind our research: this eyetracking paradigm can indeed yield results that are not achievable merely by examining written proofs. Written proofs alone would not give enough information to answer the question "Do subjects attend to the disjunction operator right away, or do they postpone it?"

8.4 Experiment 2

In this method of proof by rewriting, new symbols may emerge in the course of a proof, or symbols present at one time may be rewritten away and never reappear. This variation in the present symbols obviously leads to a variation in the premises that subjects actually *use* in their proofs. Does it also alter their behavior on the microcognitive level, changing attention during searches of the premise list?

In this experiment, we examined whether the presence or absence of an operator in the current step of a proof had an observable effect on search-space constraints when constructing the next step of the proof. We considered instances where a subject performed a rewrite step that resulted in the complete and final elimination of an operator from the proof, such as eliminating the last $\wedge$ in a statement, and this elimination step was not the conclusion of the proof. We then compared subjects' attention during the search immediately preceding the elimination step to their attention during the search immediately following it.

Table 8.1: Percentage of fixations on theorem groups before and after the elimination of an operator

Elimination of $\wedge$:	$\equiv$	$\vee$	$\wedge$
before	25%	32%	43%
after	35%	65%	0%

Elimination of $\vee$:	$\equiv$	$\vee$	$\wedge$
before	8%	84%	8%
after	79%	21%	0%

We analyzed all 15 subjects on all their work to see if this operator elimination affected the range of premises that subjects fixated when considering the proof step following the operator elimination. We considered only cases where we could make the necessary determinations with high confidence; due to poor eyetracking calibrations and similar obstacles, we excluded some proofs from consideration.

8.4.1 Results

In the proofs, the only operators eliminated by rewrites were $\vee$ and $\wedge$, and we present the results accordingly. We found 19 proof steps in which a subject eliminated some operator by a rewrite. In one such step, two operators were eliminated simultaneously, so we consider that operator $\wedge$ was eliminated 8 times and operator $\vee$ was eliminated 12 times.

In the 8 proofs in which $\wedge$ was eliminated, subjects fixated conjunction theorems 43% of the time immediately before its elimination and not at all after its elimination. In the 12 proofs in which $\vee$ was eliminated, subjects fixated disjunction theorems 84% of the time before its elimination and 21% of the time after its elimination (see Table 8.1).

8.4.2 Discussion

The elimination of an operator in a proof step had a notable effect on subjects' attention during search. This result combines with that of Experiment 1 to support an important answer to a primary question: natural changes in problem structure, both as different initial problems and as refinements during a proof in progress, are indeed accompanied by changes in attention during

Table 8.2: Percentages of proofs in which disjunction theorems were fixated upon and used

Theorem	Fixated	Used
3.24 (Symmetry of $\vee$)	71%	15%
3.25 (Associativity of $\vee$)	85%	4%
3.26 (Idempotency of $\vee$)	88%	58%
3.27 (Distrib. of $\vee$ over $\equiv$)	90%	41%
3.29 (Zero of $\vee$)	64%	32%
3.31 (Distrib. of $\vee$ over $\vee$)	62%	5%

search. (By a "natural change" in problem structure, we mean one that is not contrived, one that would occur, for instance, in the course of a student's doing a typical problem set.) People's attention during search is not static over the course of building a calculational proof, and the changes that occur are related in expected ways to the problems on which they are working.

8.5 Other Results

In addition to the interrelated results presented above, we observed other patterns in the search behavior. We present one such result here; Aaron & Spivey [AS98] contains an extended account of our observations.

8.5.1 Fixating Unused Premises

In their coursework, students become aware that some premises are cited frequently and others extremely rarely in proofs. One might expect them to pay less attention during visual search to the rarely cited premises. Across all 15 subjects, however, there were many fixations on disjunction premises (the range on which our eyetracking calibration was most reliable) that were infrequently cited in proofs (see Table 8.2).

We would not be surprised to see subjects fixating each premise for perhaps as many as 50% of the proofs. Even given their familiarity with the premise list before the experiment, we would expect subjects to look at every premise at least once and very likely twice in the span of the experiment. The fact that all the percentages of fixations here are higher than 60% suggests to us that participants paid more than minimal attention to all the disjunction premises, considering them all potentially useful.

As further background, it should be noted that Symmetry of $\vee$, Associativity of $\vee$, and Distributivity of $\vee$ over $\vee$ are cited infrequently during calculational logic coursework. Teachers encourage students to use Symmetry and Associativity implicitly without citing them, and our experiment did nothing to discourage that practice, encouraging natural behavior. Distributivity of $\vee$ over $\vee$ is also usually uncited: with expression $P \vee (Q \vee R)$, people generally (implicitly) use Associativity to get $P \vee Q \vee R$; with $(P \vee Q) \vee (P \vee R)$, people generally (implicitly) use Associativity and Symmetry and (explicitly use) Idempotency of $\vee$ to reduce the expression to $P \vee Q \vee R$, entirely bypassing Distributivity. So, we are not surprised to note that none of these three theorems were widely used.

We did find the amount of attention paid to these theorems noteworthy, given their general inutility. This supports the importance of pattern matching on the microcognition of calculational proving to an unexpected extent. The frequently observed but largely uncited premises in the range (3.24-3.29) have a strong feature-match with many of the expressions that resulted when constructing the proofs in this experiment. Premises (3.24) and (3.25) are at the top of the disjunction section, with (3.24) therefore being the first premise to feature-match with the disjunction operator in a serial, top-down search and (3.25) being the first premise to match both disjunction and parentheses, two common and frequently co-occurring features, in a serial search. Premise (3.31) may seem an exception to the influence of serial search, but it has a strong feature match with one of the most useful and frequently used premises, Distributivity of $\vee$ over $\equiv$ (3.27), a match extending even to the names of the theorems (which were present on the premise list). Hence, beyond even the expected emphasis that calculational methods place on it, pattern matching appears to play a surprisingly important role in attention during search for participants at this level of expertise.

8.6 Concluding Remarks

Some of our direct experimental results are satisfying, if not surprising, to people familiar with calculational logic, confirming for the first time their intuitions about students' microcognitive behavior. Other results are more unexpected, such as the extent of attention paid to premises that are not immediately used or not likely to ever be cited at all. The answers to these previously unexplored questions jointly serve as a preliminary demonstration that eyetracking studies can support attempts to incorporate empirically verified facts about cognitive processing in algorithmic models of logical inference. If we were to extend our tactic-based model to a more complete problem solving process (one that included searching the premise list), such a capability would become quite valuable. As it stands, our eyetracking results are examples of what could be done to further our understanding of the cognitive inferences underlying calculational logic.

In a larger sense, our results also demonstrate that eyetracking studies can indeed provide insight into the microcognition of theorem proving that experimenter intuition and studies of written output alone could not provide. In fact, it directly paves the way for new applications of this experimental procedure by providing yet another example that, far from being random, eye movements are closely related to moment-by-moment cognitive processes.

For instance, consider the result about attention to uncited premises. Our subjects, all of whom were relative novices to calculational methods, generally paid attention to unused premises. Would experts demonstrate the same behavior? A method such as ours, capable of microcognitive investigation, permits the natural continuation from the results in this paper into questions of expert/novice distinctions. This could prove useful for a cognitive model of calculational logic —in which we might hope to model different levels of expertise— but it is also of independent interest to the greater field of cognitive science. We see our application of eyetracking methods as both answering existing questions and opening up new questions in the psychology of proof.

Chapter 9

Concluding Remarks

By applying our general inference-modeling approach to calculational logic, we achieved results in several specific areas:

- We formalized calculational predicate logic (as taught to students) using a manageably sized language and conventional semantics.
- In formalizing calculational logic, we uncovered several interesting aspects of inference that had previously been hidden.
- We implemented a Nuprl tactic model of calculational logic inference. As components of that model, we implemented tactics corresponding to the major user-level inference forms of calculational logic.
- By analyzing the model and the formalization that preceded it, we gained greater insight into the pedagogy of calculational logic.
- We demonstrated that eyetracking methods can be used to gain greater insight into the cognition of theorem proving using calculational logic.

We believe that similar results would be achieved when applying the same general approach to an inference task other than calculational logic and a tactic development system other than Nuprl. As previously remarked, we chose calculational logic and Nuprl because they were good systems to use for establishing the feasibility of this general approach to inference modeling. Although eyetracking experiments do require a visual component to the high-level inference being modeled, nothing in our general approach depends upon particulars of either calculational logic or Nuprl. With this research establishing the feasibility of the approach, we believe it could fruitfully be continued in two ways: the particular calculational logic model described here could be extended and improved, and the general approach could be applied to other high-level inference tasks.

The approach results in cognitive models of high-level inference with several desirable qualities. For instance, the formalization and tactics jointly ensure that the models are deeply detailed and have a strong mathematical foundation —two sought-after properties for real-time information processing models. Eyetracking methods permit verification of descriptive accuracy on both microcognitive and macrocognitive levels. The models resulting from our approach are also compositional, so they can be understood component-wise. Cognitive theories represented in tactics can be tested component-wise: inadequate or inaccurate components could be improved or replaced without reconstructing the rest of the model. Models of experts could differ from models of novices merely by the content of selected tactics.

Models with these same desirable qualities could be constructed from tasks other than calculational logic, mathematical foundations other than Nuprl's intuitionistic type theory, and modeling tools other than tactics. This is thoroughly consistent with our research. We intended only to demonstrate the feasibility of a method for cognitive inference modeling that combines theoretical and applied elements of computer science and cognitive psychology to yield wide-ranging interdisciplinary benefits.

9.1 Tactics As Cognitive Models

As part of our concluding remarks, we interpret our tactics in the cognitive modeling context described in chapter 1. In so doing, we make explicit a fundamental premise that has been essentially implicit in this dissertation so far: a tactic that corresponds to a human-made inference embodies a cognitive model of that inference, no matter how obviously flawed that model might be.[1] The relative flaws or merits of tactics are not important to establish that, in a general way, tactics can be seen as a notational/computational framework for representing cognitive models.

In a trivial way, tactics that implement inferences that people actually make can be interpreted as cognitive models. For some purposes, such models may be inadequate in important ways —e.g., they might not be descriptively accurate in reflecting the complexity of an inference or its correct component steps in terms of actual human mental processing— but for other purposes, they may be adequate. For instance, consider the fundamental Nuprl tactic `Hypothesis`, which solves a proof goal if that goal matches a hypothesis.

[1] In fact, we believe Nuprl tactics are flawed *on some level* as cognitive models: we doubt that human inference is actually based in Nuprl type theory. Flawed models, however, are clearly not sufficient reason to discard a modeling framework. Aristotelean logic, Bayesian probability, and Johnson-Laird's mental models [JL83] all yield models seen as "obviously flawed" by some, but all those frameworks have some value for cognitive modeling of inference.

This inference is not actually atomic when people do it —for instance, the pattern-matching of a proof goal against hypotheses is a complex process— but for modeling high-level inferences, it may be appropriate to treat it as if it were. Thus, tactics may be more suitable for modeling high-level logical/mathematical inferences than low-level cognitive processes.

Another concern about tactics as cognitive models is a discrepancy that arises from a casual understanding of tactic inference and human inference: tactics are correct, whereas there are no such guarantees about human inferences. This deserves further consideration. Tactics are indeed correct *with respect to a set of primitive rules*; if the rules describe valid inferences, the tactics also do. In a flexible system such as Nuprl, the fundamental rule set can be changed, even to introduce invalid rules.[2] Therefore, the apparent restriction that tactics cannot model incorrect inferences is an illusion. If we want to model the cognitive inference of a person who consistently makes a mistaken inference,[3] we can contrive a way to do this. From a logician's perspective, this is an unusual and dangerous thing to do —soundness is not a property logicians easily sacrifice— but the ability of tactics to model false inference is significant for their viability for cognitive modeling.

9.1.1 The cognitive model embodied by our tactics

Up to this point, when considering tactics, this dissertation has been primarily concerned with implementation-level details: details of their logical foundation and how they are used in practice. As our work illustrates, it requires significant effort on this level to develop tactics representing complex, high-level inferences.[4] High-level investigations with tactic-based models require managing many low-level implementation details.

Historically, however, tactics are also strongly rooted in cognitive science, derived from a theory of problem solving as part of an approach to cognition generally. We exploited this theory and its embodiment in Nuprl to develop our cognitive inference model. As noted in [CKB84], the ML notion of tactics (in LCF [GMW79] and descendent systems such as Nuprl) is a formalization of natural top-down heuristic problem solving.[5] This goal-oriented problem solv-

[2] We employed such an invalid rule in Nuprl: the rule justifying Nuprl's `Fiat` tactic.

[3] We include common, well-studied fallacies like "affirming the consequent" (X implies Y and Y is true; therefore, X is true) and "denying the antecedent" (X implies Y and X is false; therefore, Y is false) alongside less easily explained logical failures in our notion of the mistaken inferences we might want to model.

[4] It may be tempting to think of writing tactics for this purpose as conceptually no harder than writing standard algorithms/programs in a standard programming language, but that perspective overlooks significant details. Creating tactics requires the development of rigorous, fully detailed (and mechanically checked) proofs about abstract formulas, not merely formalized methods for manipulating already-instantiated expressions.

[5] In Nuprl, the tactic mechanism was also extended to allow the study of analogy, using

ing approach (and the theory of mind it reflects) is a key element in heuristic-oriented systems like Logic Theorist (LT) [NSS57] and GPS [NS61]. Quoting Minsky [Min61]:

> The LT (Logic Theory) program is centered around the idea of 'working backward' to find a proof. ... The heuristic technique of working backwards yields something of a teleological process, and LT is a forerunner of more complex systems which construct hierarchies of goals and subgoals.

Tactic-based automated reasoning systems such as LCF, HOL [GM93], and Nuprl are among those that construct hierarchies of goals and subgoals; they adapted the cognition-oriented ideas of LT and GPS to represent high-level human inference. As the designers of LCF stated in [GMW79]:

> To make sense of the notion of tactic, we further postulate a binary relation of achievement between events and goals. Many problem solving situations can be understood as instance of these three notions: goal, event, and achievement.

Research involving these automated reasoning systems has not directly explored the connection to the cognitive modeling of their ancestors. This dissertation is a direct (if preliminary) exploration of that connection; it can be seen as an attempt to validate the ideas of LT and GPS by investigating how current systems reflect those original ideas. By reaching back in this way, we open the possibility for significant reward in two distinct but tightly related ways. Further re-consideration of LT and GPS could, of course, yield fresh insights on the old models. Novel insights into those models could also directly apply to their descendants (LCF, Nuprl, HOL), spelling out processes for enriching the cognitive models of all these systems. This describes an attractive interdependency: new insights into LT/GPS yielding cognitive improvements in modern systems, which in turn may yield new insights into LT/GPS, with the loop potentially continuing. We do not yet have the fully developed connections needed to establish such a loop, but by emphasizing the cognitive lineage behind our tactic-based approach to inference modeling, we perhaps take a first step in that direction.

With that introduction, we consider our tactics as a cognitive model of student behavior when verifying a calculational logic inference. We do not yet have an established way to describe precisely which implementation properties are part of the cognitive model (e.g., the division of Leibniz inference into sub-inferences) and which are not (e.g., the representation of type *OE* in Nuprl

---

transformation tactics (that we did not discuss here) as an attempt to capture other natural modes of cognition in the tactic framework.

type theory). Nonetheless, we are confident that, after reading the examples that follow, readers will be able to productively analyze our calculational logic tactics as a cognitive model. For full details of the tactic implementation, spelling out all the behavior of the cognitive model, see [Aar].

One aspect of our implementation with cognitive significance is the computational failure of tactics and ML functions. Major syntax errors often result in failure, as does running `Leib_tac` on a proof step that it cannot verify. This computational failure corresponds to a student giving up on the problem at hand, simply not knowing how to proceed any further. It is an integral part of the cognitive model, embodying the fact that students sometimes get stumped. It also opens a way to improve our implementation: augment the tactics with a system for user interaction. That way, when a student is stumped (a tactic fails), the teacher (user) can be asked for assistance. From many different perspectives (e.g., cognitive modeling, automated reasoning, educational software), it would be beneficial to model this kind of student/teacher interaction and its results.

To discuss the typical tactic calls that do not result in computational failure, we begin by considering the highest-level inferences directly embodied by `Leib_tac`. On a high level, according to our model, when students are presented with an inference to verify (including a premise and a way to instantiate that premise), they follow the method given in chapter 6: they decide which Leibniz lemma to apply, instantiate it, carry out the Leibniz substitution, match the premise needed for the Leibniz lemma against the premise presented to them, etc. On this level of detail, the model in chapter 6 seems plausible, but this shallow analysis fails to exploit the detail of the model. By considering slightly lower-level behavior, other interesting factors arise, and we are better able to evaluate the adequacy and cognitive descriptive accuracy of the model.

For instance, consider the behavior that our model suggests for instantiating a Leibniz lemma. It seems right in suggesting that the search for a way to instantiate the textual substitution in the Leibniz lemma is a significant component. Using some sort of recursive descent on formula structure may even be the way people actually arrive at that instantiation. Other aspects of the model seem descriptively accurate for only the most novice students, and even that assessment may be charitable. The model suggests that, when looking for a Leibniz substitution body, students consider only possibilities where the Leibniz variable appears once. In addition, it suggests that the only kind of associativity/commutativity reasoning students can do automatically (i.e., without requiring a separate proof step) is with equivalence operator $\equiv$. These make the model seem descriptively inaccurate.

Further, consider the model on sub-inferences that are components of Leibniz inference. For *OE*-typing inference, it suggests that people actually keep track of types using structures like type assignment updates, and people actually rely heavily on proving propositions by type-assignment agreement. The model may well be accurate on this point, but we have not tested it. Other modeled behavior seems less cognitively plausible. On *OV*-inequality reasoning, our tactics essentially only prove propositions by referring to hypotheses. As previously mentioned, those hypotheses are generally absent from the textbook [GS93b], but people still perform well on these inferences. People use methods and presumptions that we did not encode in our tactics, such as "different letters stand for different variables, except when obviously otherwise".

Indeed, a detailed analysis will expose many inadequacies of our tactics as a descriptively accurate model of cognitive inference. The fact that such a detailed analysis is possible, however, is a significant virtue. For instance, our model of *OV*-inequality reasoning may be inaccurate, but that inference has been absent from other models,[6] and our model could feasibly be improved to remedy its inaccuracy.

Some readers will surely find other, yet-unmentioned aspects of the proposed cognitive model implausible. They may have ideas on how to improve it by adding new constructs or changing existing ones.[7] Perhaps, for instance, we failed to encode an important strategy for proving *OE*-typing propositions. Perhaps our data language lacks some construct that is necessary for the model to be cognitively accurate. Such concerns can be readily addressed in our framework for cognitive modeling; the compositional/modular structure of the model makes it easy to alter/improve it or simply test it with different tactics. This combination —the essentials of a detailed cognitive model and the feasibility of extensions and improvements— is precisely the goal of our research. Perfection is not yet the point. We are more concerned with the approach to modeling than with one particular model.

9.2 Related Research and Future Directions

As we have mentioned, neither calculational logic nor Nuprl are essential for our approach to cognitive modeling. Continuing our research could therefore take several directions, including further feasibility demonstrations regarding the general approach, applications to calculational logic, and applications to other inference tasks.

[6]For instance, the book [GS93b] from which students were taught did not mention *OV*-inequality reasoning.

[7]The author is one such reader.

9.2.1 Extending the model

Throughout this dissertation, we have mentioned the possibility of extending our tactics to cover more of textbook [GS93b]. There are several other possible extensions of computational and cognitive interest, such as integrating eyetracking results into the model. Our feasibility demonstration for the actual computational model centered on the major user-level inference forms; our eyetracking results involved higher-level tasks (e.g., searching for premises to use in Leibniz inferences), so we did not incorporate them into the model. It would certainly be possible to run further eyetracking experiments to get results that could be used in the model or extend the current model to the point where our current eyetracking results could apply.

In addition, the model as described does not account for many factors that are important from a cognitive science perspective. For instance, we have not yet made any effort to model human memory recall and recognition in our model. We could make such a model, creating the necessary routines and data structures in the programming language ML. (Recall that tactics are part of ML in Nuprl.) Other than the expressiveness constraints on all programming languages, we would be unbounded in what kinds of memory models we could implement, from simple, brute force methods to complex spreading activation models (e.g., [Kli94]). Part of the strength of our approach is that it can be so powerfully extended.

Implementing a complex memory model in ML might be quite difficult. Indeed, implementing our complex tactic model took considerable effort, mostly in establishing its foundation. Once the fundamental modules were written, things moved much more quickly; we had essentially created a small library of tactics for the component inferences of calculational logic, so all we needed to do for further development was apply them, not re-create them. For large cognitive inference modeling tasks, this process of developing a library or toolkit of modules is an extremely useful preliminary to creating the actual model, and should perhaps be seen as a separate task altogether. Indeed, one can view this dissertation as describing a library of modular tactics for use in modeling calculational logic, along with the theory behind the modules and a demonstration that they work in the case of a simple model. A similar library could be assembled for adding memory-modeling features, and once it was implemented, creating the actual model might not be so demanding.

Another advantage of such a library would be its support of experimentation. For instance, several important issues could be explored by investigating the results of replacing one module with another from the library: expert/novice distinctions, the results of different inference strategies, the results of emphasizing certain concepts (as if they had been freshly taught) in problem solving, etc.

9.2.2 Applications to other tasks

Although our general approach could be applied to many high-level inference tasks, the model we presented in this dissertation was intended only for theorem proving. Like any computational/algorithmic model of high-level inference, it has some similarities to PSYCOP [Rip94] and other models of human inference, but those other projects are not geared to theorem proving. Our cognitive model is more similar to structures in automated mathematical proof assistants like Omega [SKM99].[8] We were not creating a comprehensive cognitive architecture (e.g., SOAR [New90]) or a model of general human problem-solving behavior; we were creating a model of formal, mathematical inference.

Nonetheless, there are aspects of our research that could be adapted for non-mathematical high-level inference tasks. For instance, the incorporation of visual attention in a deeply detailed model of inference lends itself to important questions in decision making, like what information a person *actually* attends to and uses in making a decision. The issue of what strategies are used in focusing and refocusing attention during decision making is also interesting. By adapting our method, we can study issues like how previously observed information affects where a person next focuses visual attention.

Such adaptations may be applicable to several disparate communities. Web-based technologies focus users' visual attention on presentations and products, and the focusing and refocusing process determines whether a point of pedagogy is communicated, whether a viewer is persuaded, whether a sale is made. The military must provide screens of information to strategic and tactical decision makers without flooding them with unnecessary or incomprehensible detail.[9] For life-critical occupations such as airline pilot, train engineer, and traffic controller, it is important to determine what level of granularity of information is sufficient to ensure safety without overwhelming or disrupting their abilities to focus attention. These need not be related to tactics in any way, but we believe that the general framework —formalize an inference method, create compositional models about which one can prove properties, and apply eyetracking methods to gain insight into the inference method— can be fruitfully adapted to such new contexts. Further, if someone applying this general framework wanted their model to be compositional, detailed enough to expose hidden inferences, and formalized in such a way that it could be proved correct with respect to a set of underlying rules, tactics might well be good tools to use.

[8]Omega is well suited for cognitive modeling. It has wide-ranging, well-developed capacities for representing inference methods and permitting interaction between such representations.

[9]In this military context, this suggestion is similar to work done by Marshall [M$^+$].

Computers are the tools of choice for constructing and applying practical models of high-level cognitive inference. Their successful widespread application for this purpose demonstrates that this confluence of applied logic, artificial intelligence, and cognitive science results in both practical and theoretical advancements. Our approach to cognitive modeling complements other approaches in many significant ways. It provides greater understanding of the inference method being modeled. It results in a substantial library of tactics that not only constitute a cognitive model but also serve as building blocks for future models. It draws interesting insights about computation and psychology alike from its eyetracking component. Perhaps most importantly, it opens the inference method being modeled (and its related psychology) so that new questions about it can be fruitfully pursued. Our work demonstrates the feasibility of our approach, suggesting that future research could be applied in various ways: as a part of further investigations into calculational logic, as part of existing projects pertaining to high-level cognitive inference (e.g., [Mel94, M$^+$]), or even to open up a previously unstudied inference method to exploration by both cognitive and computational sciences.

Appendix A

Highlights of the Nuprl Implementation

This appendix contains a listing of selected material from our Nuprl implementation of calculational logic inference. It presents typical code; it does not contain code for every procedure mentioned in the body of the dissertation. It has been automatically generated (except for the choice of included objects) by a Nuprl facility for printing hard copies of Nuprl objects. The kinds of objects listed here (and how they are indicated in the listing) are:

`ABS` *abstractions*, which are definitions of terms

`ML` *ML programs* and *tactics*

`THM` *theorems*

The objects are listed immediately below. Following it in section A.1 is an alphabetical index for this listing. The number next to each object name in the index refers to the position of that object in the listing. Abstraction objects are alphabetically sorted by display form; display forms are presented alongside the abstraction names.

```
---

Not_onlist(elt,l:T)                ABS notonlist_gen {1}
== Case of l
     null → True
     h.t → ¬h = elt ∈T & Not_onlist(elt,t:T)
{recursive}

---

                               THM notonlist_gen_wf {2}
∀elt:T, l:T List. Not_onlist(elt,l:T) ∈ Prop

---

Distinct_elts(l,T)              ABS alleltsdiff_gen {3}
== Case of l
     null → True
     h.t → Not_onlist(h,t:T) & Distinct_elts(t,T)
{recursive}

---

                             THM alleltsdiff_gen_wf {4}
∀l:T List. Distinct_elts(l,T) ∈ Prop

allonlist(A; L1; L2)                  ABS allonlist {5}
== ∀x:A. x onlist(A) L1 ⇒ x onlist(A) L2

                                   THM allonlist_wf {6}
∀L1,L2:A List. allonlist(A; L1; L2) ∈ Prop

---

zip(L1,L2)                                  ABS zip {7}
== Case of L1
     null → nil
     h.t → Case of L2
             null → nil
             x.y → <h,x>.zip(t,y)
{recursive}

---

                                         THM zip_wf {8}
∀L1:T List, L2:T2 List. zip(L1,L2) ∈ (T×T2) List

OGT == ℕ2                                    ABS OGT {9}
```

```
IND_O == 0                                  ABS oindtype {10}

IND_O ∈ OGT                                 THM oindtype_wf {11}

𝔹_O == 1                                    ABS obooltype {12}

𝔹_O ∈ OGT                                   THM obooltype_wf {13}

Qind == ℕ2                                  ABS quantind {14}

Qind∃ == 1                                  ABS qind_exists {15}

Qind∃ ∈ Qind                                THM qind_exists_wf {16}

Qind∀ == 0                                  ABS qind_all {17}

Qind∀ ∈ Qind                                THM qind_all_wf {18}

OTS == OGT × ℕ                              ABS obtypsigs {19}

                                            ABS ots_form {20}
ots(gdtype,degree) == <gdtype,degree>

                                            THM ots_form_wf {21}
∀gdtype:OGT, degree:ℕ. ots(gdtype,degree) ∈ OTS

ind(n) == ots(IND_O,n)                      ABS indsig {22}

∀n:ℕ. ind(n) ∈ OTS                          THM indsig_wf {23}

ind ∈ {ind, bool}                           THM indsig_wf2 {24}

bool(n) == ots(𝔹_O,n)                       ABS boolsig {25}

∀n:ℕ. bool(n) ∈ OTS                         THM boolsig_wf {26}

bool ∈ {ind, bool}                          THM boolsig_wf2 {27}
```

ABS IndBool {28}
{ind, bool} == {x:OTS| x = bool $\vee$ x = ind}

OPIDS == $\mathbb{N}5$ ABS OPIDS {29}

ABS oebv {30}
oebv(opid; posn) == if opid=$_2$4 $\longrightarrow$ 1 else 0 fi

THM oebv_wf {31}
$\forall$opid:OPIDS, posn:$\mathbb{N}$. oebv(opid; posn) $\in$ $\mathbb{N}2$

opid_f == 0 ABS opid_false {32}

opid_f $\in$ OPIDS THM opid_false_wf {33}

opid_all == 4 ABS opid_all {34}

opid_all $\in$ OPIDS THM opid_all_wf {35}

opid_eq == 3 ABS opid_eq {36}

opid_eq $\in$ OPIDS THM opid_eq_wf {37}

opid_imp == 2 ABS opid_imp {38}

opid_imp $\in$ OPIDS THM opid_imp_wf {39}

opid_ap == 1 ABS opid_ap {40}

opid_ap $\in$ OPIDS THM opid_ap_wf {41}

ABS oesbtms {42}
oesbtms(opid) == if opid=$_2$0 $\longrightarrow$ 0 ;opid=$_2$4 $\longrightarrow$ 1 else 2 fi

$\forall$opid:OPIDS. oesbtms(opid) $\in$ $\mathbb{N}$ THM oesbtms_wf {43}

IDENT == Atom $\times$ $\mathbb{N}$ ABS ident {44}

```
no_2ofs_equal(L)                    ABS no_2ofs_equal {45}
== ∀x1,x2:(IDENT × IDENT).
   ¬x1 = x2 &
   x1 onlist(IDENT × IDENT) L &
   x2 onlist(IDENT × IDENT) L
   ⇒
   ¬2of(x1) = 2of(x2)
---

                                 THM no_2ofs_equal_wf {46}
∀L:(IDENT × IDENT) List. no_2ofs_equal(L) ∈ Prop
---

all_2ofs_diff(L)                    ABS all_2ofs_diff {47}
== Case of L
     null ⟶ True
     h.t ⟶ (∀k:(IDENT × IDENT).
             k onlist(IDENT × IDENT) t ⇒
             ¬2of(h) = 2of(k) ∈IDENT) &
           all_2ofs_diff(t)
{recursive}
---

                                 THM all_2ofs_diff_wf {48}
∀L:(IDENT × IDENT) List. all_2ofs_diff(L) ∈ Prop

PossBV(opid;posn)                          ABS PossBV {49}
== if oebv(opid; posn)=₂1 ⟶ IDENT else Unit fi

                                        THM PossBV_wf {50}
∀opid:OPIDS, posn:ℕ. PossBV(opid;posn) ∈ Type

                                     ABS renamed_var {51}
ren'dvar(x,num) == <1of(x),num+1>

                                  THM renamed_var_wf {52}
∀x:IDENT, num:ℕ. ren'dvar(x,num) ∈ IDENT

                                          ABS ident2 {53}
ident2(identifier) == 2of(identifier)
```

```
                                   THM ident2_wf {54}
∀identifier:IDENT. ident2(identifier) ∈ ℕ
---

all_2ofs_gtr_n(L,n)             ABS all_2ofs_gtr_n {55}
== Case of L
     null → True
     h.t → ident2(2of(h)) > n & all_2ofs_gtr_n(t,n)
{recursive}
---

                              THM all_2ofs_gtr_n_wf {56}
∀L:(IDENT × IDENT) List, n:ℕ. all_2ofs_gtr_n(L,n) ∈ Prop
---

max_ident_num(L)                 ABS max_ident_num {57}
== Case of L
     null → 0
     h.t → imax(ident2(h);max_ident_num(t))
{recursive}
---

                               THM max_ident_num_wf {58}
∀L:IDENT List. max_ident_num(L) ∈ ℕ

id1=₂id2                              ABS eq_ident {59}
== ident1(id1)=₂ident1(id2)∈Atom∧₂
   (ident2(id1)=₂ident2(id2))

∀id1,id2:IDENT. id1=₂id2 ∈ 𝔹          THM eq_ident_wf {60}

$t|$n == <"$t",$n>          ABS ident_literal_form {61}

OE                                      ABS obexpn {62}
== rec(OE.IDENT+i:OPIDS × j:ℕ oesbtms(i) → OE × PossBV(i;j))

                                 ABS assoc_mod_iff {63}
equal_mod_AC_props(A;B) == PRIMITIVE

                               THM assoc_mod_iff_wf {64}
∀A,B:OE. equal_mod_AC_props(A;B) ∈ Prop
---
```

```
alleltsdiff(l)                      ABS alleltsdiff {65}
== Case of l
     null → True
     h.t → notonlist(h; l) & alleltsdiff(t)
{recursive}
---

                                 THM alleltsdiff_wf {66}
∀l:OE List. alleltsdiff(l) ∈ Prop

                                   ABS thmclause_mp {67}
thm_mp(exp,f,reccall1,reccall2)
== exp = f ∈OE & reccall1 & reccall2

                                THM thmclause_mp_wf {68}
∀exp,f:OE, reccall1,reccall2:Prop.
thm_mp(exp,f,reccall1,reccall2) ∈ Prop

is_a_ovar(ov) == isl(ov)              ABS is_a_ovar {69}

∀ov:OE. is_a_ovar(ov) ∈ 𝔹            THM is_a_ovar_wf {70}

(∀p|:q) == inr(<4,λposn.<q,p>>)        ABS oforall {71}

∀p:IDENT, q:OE. (∀p|:q) ∈ OE        THM oforall_wf {72}

                                            ABS oeq {73}
p = q == inr(<3,λposn.if posn=₂0→ <p,·> else <q,·> fi>)

∀p,q:OE. p = q ∈ OE                      THM oeq_wf {74}

                         ABS thmclause_equiv_as_eq {75}
thm_equiv_as_eq(exp,a,b,reccall)
== exp = a = b & reccall

                      THM thmclause_equiv_as_eq_wf {76}
∀exp,a,b:OE, reccall:Prop.
thm_equiv_as_eq(exp,a,b,reccall) ∈ Prop
```

p $\Rightarrow$ q ABS oimp {77}
== inr(<2, λposn.if posn=$_2$0 $\longrightarrow$ <p,·> else <q,·> fi>)

$\forall$p,q:OE. (p $\Rightarrow$ q) $\in$ OE THM oimp_wf {78}

ABS thmclause_arrow_intro {79}
thm_arrow_intro(exp,e,f,reccall)
== exp = (e $\Rightarrow$ f) & reccall

THM thmclause_arrow_intro_wf {80}
$\forall$exp,e,f:OE, reccall:Prop.
thm_arrow_intro(exp,e,f,reccall) $\in$ Prop

f == inr(<0, λposn.0>) ABS ofalse {81}

f $\in$ OE THM ofalse_wf {82}

t == f $\Rightarrow$ f ABS otrue {83}

t $\in$ OE THM otrue_wf {84}

u_b == if b =_Qind 0 $\longrightarrow$ t else f fi ABS unit_qind {85}

$\forall$b:Qind. u_b $\in$ OE THM unit_qind_wf {86}

$\neg$p == p $\Rightarrow$ f ABS onot {87}

$\forall$p:OE. ($\neg$p) $\in$ OE THM onot_wf {88}

($\exists$x|:p) == $\neg$($\forall$x|:$\neg$p) ABS oexists {89}

$\forall$x:IDENT, p:OE. ($\exists$x|:p) $\in$ OE THM oexists_wf {90}

p $\vee$ q == $\neg$p $\Rightarrow$ q ABS oor {91}

$\forall$p,q:OE. p $\vee$ q $\in$ OE THM oor_wf {92}

p $\wedge$ q == $\neg$ $\neg$p $\vee$ $\neg$q ABS oand {93}

```
∀p,q:OE. p ∧ q ∈ OE                      THM oand_wf {94}

                                    ABS q_op_infix {95}
r *_b p == if b =_Qind 0 → r ∧ p else r ∨ p fi

∀b:Qind, r,p:OE. r *_b p ∈ OE    THM q_op_infix_wf {96}

                                  ABS suchthat_qind {97}
s.t.(b,r,p) == if b =_Qind 0 → r ⇒ p else r ∧ p fi

                               THM suchthat_qind_wf {98}
∀b:Qind, r,p:OE. s.t.(b,r,p) ∈ OE

p ≡ q == p ⇒ q ∧ q ⇒ p                   ABS oequiv {99}

∀p,q:OE. p ≡ q ∈ OE                THM oequiv_wf {100}

Ovar(X) == inl(X)                       ABS ovar {101}

∀X:IDENT. Ovar(X) ∈ OV               THM ovar_wf2 {102}

∀X:IDENT. Ovar(X) ∈ OE                THM ovar_wf {103}

$vname_ov == Ovar(<"$vname",0>)      ABS ovar_lit {104}

IsOvar(Y) == ∃X:IDENT. Ovar(X) = Y     ABS IsOvar {105}

∀Y:OE. IsOvar(Y) ∈ Prop             THM IsOvar_wf {106}

OV == {Y:OE| ∃X:IDENT. Ovar(X) = Y}     ABS obvar {107}

                                   ABS otse_agree {108}
(otse1,otse2 agree on x) == otse1(x) = otse2(x) ∈ OTS

                                THM otse_agree_wf {109}
∀otse1:(OV → OTS), x:OV, otse2:(OV → OTS).
(otse1,otse2 agree on x) ∈ Prop
```

```
identofovar(ov) == outl(ov)       ABS identofovar {110}

                               THM identofovar_wf {111}
∀ov:OV. identofovar(ov) ∈ IDENT

(*_b x|:p)                     ABS emptyrangequant {112}
== if b =_Qind 0 → (∀identofovar(x)|:p)
   else (∃identofovar(x)|:p) fi

                            THM emptyrangequant_wf {113}
∀b:Qind, x:OV, p:OE. (*_b x|:p) ∈ OE
---

(*_b xs | r : p)                          ABS qstar {114}
== Case of xs
     null → s.t.(b,r,p)
     h.t → (*_b h|:(*_b t | r : p))
{recursive}
---

                                      THM qstar_wf {115}
∀b:Qind, xs:OV List, r,p:OE. (*_b xs | r : p) ∈ OE

                                     ABS ol_exists {116}
ol_exists(xs; r; p) == (*_1 xs | r : p)

                                 THM ol_exists_wf {117}
∀xs:OV List, r,p:OE. ol_exists(xs; r; p) ∈ OE

                                        ABS ol_all {118}
ol_all(xs; r; p) == (*_0 xs | r : p)

                                     THM ol_all_wf {119}
∀xs:OV List, r,p:OE. ol_all(xs; r; p) ∈ OE

                                         ABS eq_ov {120}
x=₂y == identofovar(x)=₂identofovar(y)

∀x,y:OV. x=₂y ∈ 𝔹                       THM eq_ov_wf {121}
```

```
eq_OV(x,y) == x=₂y                    ABS eqovsep {122}

eq_OV ∈ OV → OV → 𝔹                   THM eqovsep_wf {123}

                                        ABS obap {124}
p(q) == inr(<1, λposn.if posn=₂0 → <p,·> else <q,·> fi>)

∀p,q:OE. p(q) ∈ OE                     THM obap_wf {125}

                                        ABS non_ov {126}
NonOV == {Y:OE| ¬(∃X:IDENT. Ovar(X) = Y)}

                                   ABS opid_of_OE {127}
opid_of_OE(obexpn) == 1of(outr(obexpn))

                               THM opid_of_OE_wf {128}
∀obexpn:NonOV. opid_of_OE(obexpn) ∈ OPIDS

                                     ABS fn_of_OE {129}
fn_of_OE(obexpn) == 2of(outr(obexpn))
---

∀obexpn:NonOV.                    THM fn_of_OE_wf {130}
fn_of_OE(obexpn)
∈ j:ℕoesbtms(opid_of_OE(obexpn)) → OE ×
  PossBV(opid_of_OE(obexpn);j)
---

                                      ABS nthsbtm {131}
nthsbtm(oe,posn) == 1of(fn_of_OE(oe)(posn))

                                   THM nthsbtm_wf {132}
∀oe:NonOV, posn:ℕoesbtms(opid_of_OE(oe)).
nthsbtm(oe,posn) ∈ OE
---
```

```
Case of term:OE                           ABS OEcase {133}
  Ovar(x)  ⟶  varfn(x)
  f     ⟶  falsecase
  u(v)  ⟶  apfn(u;v)
  u ⇒ v  ⟶  impfn(u;v)
  u=v  ⟶  eqlfn(u;v)
  (∀x|:v)  ⟶  allfn(x;u)
== Case of term
     inl(x)  ⟶  varfn(x)
     inr(y)  ⟶  if
                opid_of_OE(term)=₂0 ⟶  falsecase
               ;opid_of_OE(term)=₂1 ⟶
                apfn(nthsbtm(term,0);nthsbtm(term,1))
               ;opid_of_OE(term)=₂2 ⟶
                impfn(nthsbtm(term,0);nthsbtm(term,1))
               ;opid_of_OE(term)=₂3 ⟶
                eqlfn(nthsbtm(term,0);nthsbtm(term,1))
               else allfn(2of(fn_of_OE(term)(0))
                         ;nthsbtm(term,0)) fi
---

                                        ABS bv_of_OE {134}
bv_of_OE(oe,posn) == 2of(fn_of_OE(oe)(posn))
{returns IDENT or UNIT }


                                     THM bv_of_OE_wf {135}
∀oe:NonOV, posn:ℕoesbtms(opid_of_OE(oe)).
bv_of_OE(oe,posn)  ∈  PossBV(opid_of_OE(oe);posn)
---

is_bvar_in_OE(id,exp)           ABS is_bvar_in_OE {136}
== if
    is_a_ovar(exp) ⟶  False
   ;opid_of_OE(exp)=₂opid_f ⟶  False
   ;opid_of_OE(exp)=₂opid_all ⟶
    bv_of_OE(exp,0) = id ∈ IDENT  ∨  is_bvar_in_OE(
                                    id,nthsbtm(exp,0))
   else is_bvar_in_OE(id,nthsbtm(exp,0))
        ∨  is_bvar_in_OE(id,nthsbtm(exp,1)) fi
{recursive}
```

```
---

                              THM is_bvar_in_OE_wf {137}
∀exp:OE, id:IDENT. is_bvar_in_OE(id,exp) ∈ Prop
---

Osubstn_ready(E,n)              ABS osubstn_ready {138}
== ∀id:IDENT. is_bvar_in_OE(id,E) ⇒ ident2(id) > n


---

                              THM osubstn_ready_wf {139}
∀E:OE, n:ℤ. Osubstn_ready(E,n) ∈ Prop
---

hnum_aux(e)                          ABS hnum_aux {140}
== if
    is_a_ovar(e) → ident2(identofovar(e))
   ;opid_of_OE(e)=₂opid_f → 0
   ;opid_of_OE(e)=₂opid_all →
    imax(ident2(bv_of_OE(e,0));hnum_aux(nthsbtm(e,0)))
   else imax(hnum_aux(nthsbtm(e,0));
             hnum_aux(nthsbtm(e,1))) fi
{recursive}
---

∀x:OE. hnum_aux(x) ∈ ℕ              THM hnum_aux_wf {141}
---

hnum(es)                                 ABS hnum {142}
== Case of es
     null → 0
     x.y → imax(hnum_aux(x);hnum(y))
{recursive}

{Note it's for use in big_ident_num }
---

∀es:OE List. hnum(es) ∈ ℕ               THM hnum_wf {143}
---
```

```
big_ident_num(e,Vs,Ps)           ABS big_ident_num {144}
== imax(hnum_aux(e);imax(max_ident_num(Vs);hnum(Ps)))

{This is to get the number above which all bd vars need to
 be rewritten for substitution.}
---

                             THM big_ident_num_wf {145}
∀e:OE, Vs:IDENT List, Ps:OE List.
big_ident_num(e,Vs,Ps) ∈ ℕ
---

(v occurs free in e)                     ABS Ofree {146}
== if
    is_a_ovar(e) → e = v ∈OV
   ;opid_of_OE(e)=₂opid_f → False
   ;opid_of_OE(e)=₂opid_all →
    v≠Ovar(bv_of_OE(e,0)) ∈OV &
    (v occurs free in nthsbtm(e,0))
   else (v occurs free in nthsbtm(e,0)) ∨ ... fi
{recursive}

{this is for one OV and one expression }
---

                                     THM Ofree_wf {147}
∀e:OE, v:OV. (v occurs free in e) ∈ Prop

                          ABS thmclause_all_elim2 {148}
thm_all_elim2(exp,e,x,reccall)
== exp = e ∈OE & reccall & ¬(x occurs free in e)

                       THM thmclause_all_elim2_wf {149}
∀exp,e:OE, reccall:Prop, x:OV.
thm_all_elim2(exp,e,x,reccall) ∈ Prop
---
```

```
                                 ABS Es_notoccurs {150}
{OV v not occur free in any OE in Es }
Es_not_occurs(v,Es)
== Case of Es
     null → True
     h.t → ¬(v occurs free in h) & Es_not_occurs(v,t)
{recursive}
---

                              THM Es_notoccurs_wf {151}
∀Es:OE List, v:OV. Es_not_occurs(v,Es) ∈ Prop
---

(no vs⌈+⌉ occur free in Es)    ABS NotOcc_vsfirst {152}
== Case of vs
     null → True
     h.t → Es_not_occurs(h,Es) &
           (no t⌈+⌉ occur free in Es)
{recursive}
---

∀vs:OV List, Es:OE List.    THM NotOcc_vsfirst_wf {153}
(no vs⌈+⌉ occur free in Es) ∈ Prop
---

                                 ABS vs_notoccurs {154}
{no OV in vs occurs free in OE E }
vs_not_occurs(vs,E)
== Case of vs
     null → True
     h.t → ¬(h occurs free in E) & vs_not_occurs(t,E)
{recursive}
---

                              THM vs_notoccurs_wf {155}
∀vs:OV List, E:OE. vs_not_occurs(vs,E) ∈ Prop
---

(no vs occur free in Es⌈+⌉)    ABS NotOcc_Esfirst {156}
== Case of Es
     null → True
     h.t → vs_not_occurs(vs,h) &
           (no vs occur free in t⌈+⌉)
{recursive}
```

```
---
∀Es:OE List, vs:OV List.    THM NotOcc_Esfirst_wf {157}
(no vs occur free in Es⌈+⌉) ∈ Prop
---
                                        ABS Esoccurs {158}
{does OV v occur free in some OE in Es }
Esoccurs(v,Es)
== Case of Es
     null → False
     h.t → (v occurs free in h) ∨ Esoccurs(v,t)
{recursive}
---
                                     THM Esoccurs_wf {159}
∀Es:OE List, v:OV. Esoccurs(v,Es) ∈ Prop
---
                               ABS Occurs_vsfirst {160}
{some vs free in some OE on Es; unroll vs first }
(some vs⌈+⌉
 occur free in Es)
== Case of vs
     null → False
     h.t → Esoccurs(h,Es) ∨ (some t⌈+⌉
                             occur free in Es)
{recursive}
---
∀vs:OV List, Es:OE List.    THM Occurs_vsfirst_wf {161}
(some vs⌈+⌉ occur free in Es) ∈ Prop
---
vsoccurs(vs;E)                          ABS vsoccurs {162}
== Case of vs
     null → False
     h.t → (h occurs free in E) ∨ vsoccurs(t;E)
{recursive}
---
                                     THM vsoccurs_wf {163}
∀E:OE, vs:OV List. vsoccurs(vs;E) ∈ Prop
---
```

```
(some vs occur free in Es)            ABS Occurs {164}
== Case of Es
     null → False
     h.t → vsoccurs(vs;h) ∨ (some vs occur free in t)
{recursive}
---

∀vs:OV List, Es:OE List.            THM Occurs_wf {165}
(some vs occur free in Es) ∈ Prop
---

                          ABS thmclause_all_intro {166}
thm_all_intro(exp,x,e,assumps,reccall)
== exp = (∀x|:e) &
   reccall &
   ¬(some [x] occur free in assumps)
---

                       THM thmclause_all_intro_wf {167}
∀exp:OE, x:OV, e:OE, assumps:OE List, reccall:Prop.
thm_all_intro(exp,x,e,assumps,reccall) ∈ Prop

                                     ABS mk_nonOV {168}
mk_nonOV(opid,fn_of_oe) == inr(<opid,fn_of_oe>)
---

                                  THM mk_nonOV_wf {169}
∀opid:OPIDS, fn_of_oe:j:ℕoesbtms(opid)→OE ×
                      PossBV(opid;j).
mk_nonOV(opid,fn_of_oe) ∈ OE
---

                                 THM mk_nonOV_wf2 {170}
∀opid:OPIDS, fn_of_oe:j:ℕoesbtms(opid)→OE ×
                      PossBV(opid;j).
mk_nonOV(opid,fn_of_oe) ∈ NonOV
---

[x] == [x]                           ABS singlist {171}

∀x:T. [x] ∈ T List                 THM singlist_wf {172}

env(V) == (IDENT × V) List            ABS env_type {173}
---
```

```
lookup_env(ident,env)              ABS lookup_env {174}
== Case of env
     null  →  inr(·)
     h.t  →  if ident=₂1of(h) →  inl(2of(h))
           else lookup_env(ident,t) fi
{recursive}
---

∀ident:IDENT, env:env(T).       THM lookup_env_wf {175}
lookup_env(ident,env)  ∈  T+Unit
---

rename_lem_env(env,L)    ABS env_for_rename_lemma {176}
== Case of L
     null  →  nil
     h.t  →  if isl(lookup_env(1of(h),env)) →
              <2of(h),outl(lookup_env(1of(h),env))>
              .rename_lem_env(env,t)
           else rename_lem_env(env,t) fi
{recursive}

IDENTnums_gtr_n(L,n)          ABS IDENTnums_gtr_n {177}
== ∀id:IDENT.
   isl(lookup_env(id,L))  ⇒
   ident2(outl(lookup_env(id,L))) > n
---

                           THM IDENTnums_gtr_n_wf {178}
∀L:env(IDENT), n:ℤ. IDENTnums_gtr_n(L,n)  ∈  Prop
---
```

```
zip_env(idents,vals)                  ABS zip_env {179}
== Case of idents
     null ⟶ nil
     h.t ⟶ Case of vals
             null ⟶ nil
             x.y ⟶ <h,x>.zip_env(t,y)
{recursive}
---

                                   THM zip_env_wf {180}
∀idents:IDENT List, vals:T List.
zip_env(idents,vals) ∈ env(T)

l@env == l @ env                 ABS updates_env {181}

                               THM updates_env_wf {182}
∀l,env:env(T). l@env ∈ env(T)

                                   ABS update_env {183}
update_env([var,val],env) == [<var,val>]@env

                                THM update_env_wf {184}
∀var:IDENT, val:T, env:env(T).
update_env([var,val],env) ∈ env(T)
---
```

```
rename(e,num,rewrites)            ABS renamebdvars {185}
== Case of e
     inl(x)  →  Case of lookup_env(x,rewrites)
                inl(y)  →  Ovar(y)
                inr(z)  →  Ovar(x)
     inr(t)
      →  if
         opid_of_OE(e)=₂0 →  e
        ;opid_of_OE(e)=₂4 →
         (∀<1of(bv_of_OE(e,0)),num+1>
          |:rename(nthsbtm(e,0),num+1,
                  update_env([bv_of_OE(e,0),
                              <1of(bv_of_OE(e,0))
                              ,num+1>],
                             rewrites)))
        else mk_nonOV(opid_of_OE(e),λposn.
             if posn=₂0 →
                <rename(nthsbtm(e,0),num,rewrites),·>
             else <rename(nthsbtm(e,1),num,rewrites)
                  ,·> fi) fi
{recursive}

{rewrites is a env(IDENT) }
---

                              THM renamebdvars_wf {186}
∀e:OE, num:ℕ, rewrites:env(IDENT).
rename(e,num,rewrites)  ∈  OE
---
```

```
captureoksubst(e; vps)          ABS captureoksubst {187}
== Case of e
     inl(x) → Case of lookup_env(x,vps)
                inl(y) → y
                inr(z) → e
     inr(t) → if
               opid_of_OE(e)=_2 opid_f → f
              ;opid_of_OE(e)=_2 opid_all →
               (∀bv_of_OE(e,0)
                |:captureoksubst
                  (nthsbtm(e,0);
                   update_env([bv_of_OE(e,0),
                               Ovar(bv_of_OE(e,0))],
                              vps)))
              else mk_nonOV(opid_of_OE(e),λposn.
                   if posn=_2 0 →
                      <captureoksubst(nthsbtm(e,0); vps
                                      )
                      ,·>
                   else <captureoksubst(nthsbtm(e,1);
                                        vps)
                        ,·> fi) fi
{recursive}

{vps is a env(OE), named from v := p pairs }
---

                             THM captureoksubst_wf {188}
∀e:OE, vps:env(OE). captureoksubst(e; vps) ∈ OE
---

e[vs := ps]                             ABS osubstn {189}
== captureoksubst(rename(e,big_ident_num(e,map(λx.
                       identofovar(x);vs),ps),nil);
                  zip_env(map(λx.identofovar(x);vs),ps)
                  )
---

                                    THM osubstn_wf {190}
∀e:OE, vs:OV List, ps:OE List. e[vs := ps] ∈ OE
```

```
(e)[v := p] == e[[v] := [p]]      ABS osubst_1var {191}

                                 THM osubst_1var_wf {192}
∀e:OE, v:OV, p:OE. (e)[v := p] ∈ OE
---

findfirst(L; x.P(x))                ABS findfirst {193}
== Case of L
     null → inr(·)
     h.t → if P(h) → inl(h) else findfirst(t; x.P(x)) fi
{recursive}


---

                                   THM findfirst_wf {194}
∀L:T List, P:(T → 𝔹). findfirst(L; x.P(x)) ∈ T+Unit
---
```

```
t1 =_alph(corr) t2                  ABS Oalpha_eq {195}
== if
    is_a_ovar(t1) →
    if is_a_ovar(t2) →
       Case of findfirst(corr;
                         h.(1of(h)=₂ identofovar(t1)
                         ∨ ₂  2of(h)=₂ identofovar(t2)))
         inl(x)  →  x = <identofovar(t1),identofovar(t2)>
         inr(y)  →  t1 = t2  ∈ OV
    else False fi
   ;is_a_ovar(t2) →  False
   ;opid_of_OE(t1)=₂ opid_of_OE(t2) →
    ∀i:ℕ oesbtms(opid_of_OE(t1)).
    nthsbtm(t1,i) =_alph(if oebv(opid_of_OE(t1); 0)=₂ 1 →
                            <bv_of_OE(t1,0)
                            ,bv_of_OE(t2,0)>
                            .corr
                         else corr fi) nthsbtm(t2,i)
   else False fi
{recursive}

{corr is going to be an IDENTXIDENT list, not an env }
---

                                 THM Oalpha_eq_wf {196}
∀t1,t2:OE, corr:(IDENT × IDENT) List.
t1 =_alph(corr) t2  ∈  Prop


                                ABS thmclause_cbv {197}
thm_cbv(exp,f,rec_call) == exp =_alph(nil) f & rec_call


∀exp,f:OE, rec_call:Prop.    THM thmclause_cbv_wf {198}
thm_cbv(exp,f,rec_call)  ∈  Prop


                                     ABS fnupdate {199}
f[x{eq}:=v](y) == if eq(x,y) →  v else f(y) fi


                                  THM fnupdate_wf {200}
∀eq:(T → T → 𝔹), x:T, v:U, f:(T → U). f[x{eq}:=v]  ∈  T → U
```

```
otse[X ← v] == otse[X{eq_OV}:=v]    ABS otseupdate {201}

                                THM otseupdate_wf {202}
∀X:OV, v:OTS, otse:(OV → OTS). otse[X ← v] ∈ OV → OTS
---

otse[Vs ← Sigs]              ABS otseupdate_2lists {203}
== Case of Vs
     null → otse
     X.Ys → Case of Sigs
              null → otse
              S.Ts → otse[X ← S][Ys ← Ts]
{recursive}
---

                         THM otseupdate_2lists_wf {204}
∀Vs:OV List, Sigs:OTS List, otse:(OV → OTS).
otse[Vs ← Sigs] ∈ OV → OTS

otse[Xs ← v]               ABS otseupdate_var_list {205}
== Case of Xs :  null → otse ; X.Ys → otse[X ← v][Ys ← v]
{recursive}

                       THM otseupdate_var_list_wf {206}
∀Xs:OV List, otse:(OV → OTS), v:OTS.
otse[Xs ← v] ∈ OV → OTS
---

f[Xs{eq}:=v]                ABS fnupdate_var_list {207}
== Case of Xs
     null → f
     X.Ys → f[X{eq}:=v][Ys{eq}:=v]
{recursive}
---

                         THM fnupdate_var_list_wf {208}
∀Xs:T List, f:(T → U), eq:(T → T → 𝔹), v:U.
f[Xs{eq}:=v] ∈ T → U
---
```

```
(env) e :: sig                           ABS Owfdelt {209}
== Case of e:OE
     Ovar(x)  →  env(e) = sig  ∈ OTS
     f      →  sig = bool
     p(q)  →  (env) q :: ind &
            (env) p :: ots(1of(sig),2of(sig)+1)
     p ⇒ q  →  sig = bool &
            (env) p :: bool &
            (env) q :: bool
     p=q  →  sig = bool &
           ((env) p :: bool & (env) q :: bool  ∨  (env)
            p :: ind &
            (env) q :: ind)
     (∀x|:q)  →  sig = bool &
               (env[Ovar(x){eq_OV}:=ind]) p :: bool
---

                                       THM Owfdelt_wf {210}
∀e:OE, sig:OTS, env:(OV → OTS). (env) e :: sig  ∈  Prop

                           ABS thmclause_false_elim {211}
thm_false_elim(exp,e,env,reccall)
== exp = e  ∈ OE & reccall & (env) e :: bool

                        THM thmclause_false_elim_wf {212}
∀exp,e:OE, reccall:Prop, env:(OV → OTS).
thm_false_elim(exp,e,env,reccall)  ∈  Prop

Ogdtmelt(exp,env)                        ABS Ogdtmelt {213}
== (env) exp :: bool  ∨  (env) exp :: ind

                                      THM Ogdtmelt_wf {214}
∀exp:OE, env:(OV → OTS). Ogdtmelt(exp,env)  ∈  Prop
---

(env) Es :: sig                               ABS Owfd {215}
== Case of Es
     null  →  True
     e.es  →  (env) e :: sig & (env) es :: sig
{recursive}
```

```
---
                                          THM Owfd_wf {216}
∀Es:OE List, sig:OTS, env:(OV → OTS).
(env) Es :: sig ∈ Prop
---
                               ABS thmclause_eq_intro {217}
thm_eq_intro(exp,a,env,assumps)
== exp = a = a &
   (∃r:{ind, bool}.
    (env) a :: r & (env) assumps :: bool)
---
                            THM thmclause_eq_intro_wf {218}
∀exp,a:OE, env:(OV → OTS), assumps:OE List.
thm_eq_intro(exp,a,env,assumps) ∈ Prop
---
                                ABS thmclause_eq_elim {219}
thm_eq_elim(exp,e,x,a,b,env,reccall1,reccall2)
== exp = (e)[x := b] &
   reccall1 &
   reccall2 &
   (∃r:{ind, bool}.
    (env) [a; b] :: r & (env[x ← r]) e :: bool)
---
                             THM thmclause_eq_elim_wf {220}
∀exp,e:OE, x:OV, b:OE, reccall1,reccall2:Prop, a:OE
, env:(OV → OTS).
thm_eq_elim(exp,e,x,a,b,env,reccall1,reccall2) ∈ Prop
---
                                            ABS Ogdtm {221}
Ogdtm(Es; env) == (env) Es :: bool ∨ (env) Es :: ind

                                         THM Ogdtm_wf {222}
∀Es:OE List, env:(OV → OTS). Ogdtm(Es; env) ∈ Prop

                               ABS thmclause_all_elim {223}
thm_all_elim(exp,e,x,f,env,reccall)
== exp = (e)[x := f] & reccall & (env) f :: ind
```

```
                        THM thmclause_all_elim_wf {224}
∀exp,e:OE, x:OV, f:OE, env:(OV → OTS), reccall:Prop.
thm_all_elim(exp,e,x,f,env,reccall) ∈ Prop
---

                           ABS thmclause_thinning {225}
thm_thinning(sublist,env,assumps,reccall)
== allonlist(OE; sublist; assumps) & reccall &
   (env) assumps :: bool
---

                        THM thmclause_thinning_wf {226}
∀sublist,assumps:OE List, reccall:Prop, env:(OV → OTS).
thm_thinning(sublist,env,assumps,reccall) ∈ Prop

                         ABS thmclause_hypothesis {227}
thm_hyp(exp,env,assumps)
== exp onlist(OE) assumps & (env) assumps :: bool

                      THM thmclause_hypothesis_wf {228}
∀exp:OE, env:(OV → OTS), assumps:OE List.
thm_hyp(exp,env,assumps) ∈ Prop
---
```

```
Othmaux_2(exp,env,assumps,n)       ABS Othmaux_v2 {229}
== thm_hyp(exp,env,assumps)  ∨  0<n &
   ((∃sublist:OE List.
     thm_thinning(sublist,env,assumps,
                  Othmaux_2(exp,env,sublist,n-1)))
    ∨ (∃e,f:OE.
       thm_mp(exp,f,Othmaux_2(e  ⇒  f,env,assumps,n-1),
              Othmaux_2(e,env,assumps,n-1)))
    ∨ (∃e,f:OE.
       thm_arrow_intro(exp,e,f,Othmaux_2(f,env,
                                         e.assumps,
                                         n-1)))
    ∨ (∃f:OE.
       thm_cbv(exp,f,Othmaux_2(f,env,assumps,n-1)))
    ∨ (∃x:OV, e,f:OE.
       thm_all_elim(exp,e,x,f,env,
                    Othmaux_2((∀x|:e),env,assumps,
                              n-1)))
    ∨ (∃x:OV, e:OE.
       thm_all_elim2(exp,e,x,
                     Othmaux_2((∀x|:e),env,assumps,
                               n-1)))
    ∨ (∃x:OV, e:OE.
       thm_all_intro(exp,x,e,assumps,
                     Othmaux_2(e,env[x ⟵  ind],assumps,
                               n-1)))
    ∨ (∃e:OE.
       thm_false_elim(exp,e,env,
                      Othmaux_2(f,env,assumps,n-1)))
    ∨ (∃x:OV, e,a,b:OE.
       thm_eq_elim(exp,e,x,a,b,env,
                   Othmaux_2((e)[x := a],env,assumps,
                             n-1),
                   Othmaux_2(a = b,env,assumps,n-1)))
    ∨ (∃a:OE. thm_eq_intro(exp,a,env,assumps))
    ∨ (∃a,b:OE.
       thm_equiv_as_eq(exp,a,b,
                       Othmaux_2(a  ≡  b,env,assumps,
                                 n-1))))
{recursive}
```

THM Othmaux_v2_wf {230}

$\forall$n: $\mathbb{N}$, exp:OE, env:(OV $\longrightarrow$ OTS), assumps:OE List.
Othmaux_2(exp,env,assumps,n) $\in$ Prop

(env) assumps $\vdash$ exp   ABS Othm {231}
== $\exists$n: $\mathbb{N}$. Othmaux_2(exp,env,assumps,n)

THM Othm_wf {232}

$\forall$exp:OE, env:(OV $\longrightarrow$ OTS), assumps:OE List.
(env) assumps $\vdash$ exp $\in$ Prop

nil $\in$ env(t)   THM NilIsEnv {233}

$\forall$x:OPIDS. x $\in$ $\mathbb{Z}$ THM OPIDS_in_int {234}

THM unrolled_OE_in_OE {235}

(IDENT+i:OPIDS $\times$ j: $\mathbb{N}$ oesbtms(i) $\longrightarrow$ OE $\times$ PossBV(i;j)) $\subseteq$ OE

THM OE_in_unrolled_OE {236}

OE $\subseteq$ (IDENT+i:OPIDS $\times$ j: $\mathbb{N}$ oesbtms(i) $\longrightarrow$ OE $\times$ PossBV(i;j))

$\forall$x,y:OE.   THM OEexps_in_unrolled_OE {237}
x = y
$\Longleftarrow\!\Longrightarrow$
x = y $\in$ IDENT+i:OPIDS $\times$ j: $\mathbb{N}$ oesbtms(i) $\longrightarrow$ OE $\times$ PossBV(i;j)

THM unrolled_OE_exps_in_OE {238}

$\forall$x,y:IDENT+i:OPIDS $\times$ j: $\mathbb{N}$ oesbtms(i) $\longrightarrow$ OE $\times$ PossBV(i;j).
x = y $\in$ OE $\Longleftarrow\!\Longrightarrow$ x = y

OV $\subseteq$ OE   THM obvar_inc {239}

THM Ofree_on_OVs {240}

$\forall$v,x:OV. (v occurs free in x) $\Longleftarrow\!\Longrightarrow$ v = x

THM Occurs_on_OVs {241}

$\forall$v,x:OV. (some [v] occur free in [x]) $\Longleftarrow\!\Longrightarrow$ v = x

THM Occurs_on_OVs_for_backchn {242}

```
∀v,x:OV. ¬(some [v] occur free in [x]) ⇒ ¬v = x
```

THM Occurs_on_OVs_for_NotOcc_bckchn {243}

```
∀v,x:OV. ¬v = x ⇒ ¬(some [v] occur free in [x])
---
```

THM Occ_EsList_expand {244}

```
∀Vs:OV List, H:OE, T:OE List.
(some Vs occur free in H.T)
⇐⇒
(some Vs occur free in [H]) ∨ (some Vs occur free in T)
---
```

```
∀x,y:OV. ¬x = y ⇒ ¬y = x
```

THM NotEq_OV_sym {245}

THM Occurs_on_OVs_for_backchn_sym {246}

```
∀v,x:OV. ¬(some [v] occur free in [x]) ⇒ ¬x = v
```

THM Occurs_on_OVs_for_NotOcc_bckchn_sym {247}

```
∀v,x:OV. ¬x = v ⇒ ¬(some [v] occur free in [x])
---
```

THM OVs_diff_OTS_then_neq {248}

```
∀v:OV, sig1:OTS, ∈:(OV→OTS), x:OV, sig2:OTS.
(∈) [v] :: sig1 & (∈) [x] :: sig2 & ¬sig1 = sig2 ⇒
¬v = x
---
```

THM Not_Occ_on_varListvariable {249}

```
∀X:OV, Y:OV List, alpha:OE List.
¬(some X.Y occur free in alpha)
⇐⇒
¬(some [X] occur free in alpha) &
¬(some Y occur free in alpha)
---
```

THM NotOcc_Es_iff_NotOcc_vs {250}

```
∀Es:OE List, Vs:OV List.
(no Vs occur free in Es⌈+⌉)
⇐⇒
(no Vs⌈+⌉ occur free in Es)
---
```

```
                   THM NotOcc_Es_iff_Es_notoccurs {251}
∀Es:OE List, x:OV.
Es_not_occurs(x,Es) ⟸⟹ (no [x] occur free in Es⌈+⌉)
---

∀Vs:OV List, Es:OE List.    THM Occ_Es_iff_Occ_vs {252}
(some Vs occur free in Es)
⟸⟹
(some Vs⌈+⌉
 occur free in Es)
---

                        THM NotOcc_iff_not_Occ_Es {253}
∀Vs:OV List, Es:OE List.
¬(some Vs occur free in Es)
⟸⟹
(no Vs occur free in Es⌈+⌉)
---

                          THM Not_Occ_on_var_List {254}
∀X:OV, Y:OV List, E:OE.
¬(some X.Y occur free in [E])
⟸⟹
¬(some [X] occur free in [E]) &
¬(some Y occur free in [E])
---

                          THM Not_Occ_on_exp_List {255}
∀X:OV List, E:OE, Y:OE List.
¬(some X occur free in E.Y)
⟸⟹
¬(some X occur free in [E]) & ¬(some X occur free in Y)
---

                        THM NotOcc_iff_not_Occ_vs {256}
∀Vs:OV List, Es:OE List.
¬(some Vs⌈+⌉
  occur free in Es)
⟸⟹
(no Vs⌈+⌉ occur free in Es)
---
```

```
∀X:OV List, P,Q:OE.                THM Not_Occ_oeq {257}
¬(some X occur free in [P = Q])
⇐⇒
¬(some X occur free in [P]) &
¬(some X occur free in [Q])
---

∀X:OV List, P,Q:OE.              THM Not_Occ_OLand {258}
¬(some X occur free in [P ∧ Q])
⇐⇒
¬(some X occur free in [P]) &
¬(some X occur free in [Q])
---

∀X:OV List, P,Q:OE.               THM Not_Occ_OLor {259}
¬(some X occur free in [P ∨ Q])
⇐⇒
¬(some X occur free in [P]) &
¬(some X occur free in [Q])
---

                             THM Not_Occ_OLfalse {260}
∀X:OV List. ¬(some X occur free in [f])

                              THM Not_Occ_OLtrue {261}
∀X:OV List. ¬(some X occur free in [t])

                          THM Not_Occ_OLunitqind {262}
∀X:OV List, b:Qind. ¬(some X occur free in [u_b])
---

                          THM Not_Occ_OLqopinfix {263}
∀X:OV List, b:Qind, P,Q:OE.
¬(some X occur free in [P *_b Q])
⇐⇒
¬(some X occur free in [P]) &
¬(some X occur free in [Q])
---
```

```
∀X:OV List, P,Q:OE.              THM Not_Occ_obap {264}
¬(some X occur free in [P(Q)])
⇐⇒
¬(some X occur free in [P]) &
¬(some X occur free in [Q])
---

∀X:OV List, P:OE.                THM Not_Occ_OLnot {265}
¬(some X occur free in [¬P])
⇐⇒
¬(some X occur free in [P])
---

∀X:OV List, P,Q:OE.            THM Not_Occ_oequiv {266}
¬(some X occur free in [P ≡ Q])
⇐⇒
¬(some X occur free in [P]) &
¬(some X occur free in [Q])
---

∀X:OV List, P,Q:OE.              THM Not_Occ_oimp {267}
¬(some X occur free in [P ⇒ Q])
⇐⇒
¬(some X occur free in [P]) &
¬(some X occur free in [Q])
---

∀V:OV, P,E:OE.               THM Not_Occ_osubst1v {268}
¬(some [V] occur free in [P]) ⇒
¬(some [V] occur free in [(E)[V := P]])
---

                              THM Not_Occ_osubstn {269}
∀Vs:OV List, Ps:OE List, E:OE.
¬(some Vs occur free in Ps) ⇒
¬(some Vs occur free in [E[Vs := Ps]])
---
```

THM Not_Occ_qstar {270}

```
∀V:OV, b:Qind, Y:OV List, R,B:OE.
¬V onlist(OV) Y ⇒
(¬(some [V] occur free in [R]) &
 ¬(some [V] occur free in [B])
 ⇐⇒
 ¬(some [V] occur free in [(*_b Y | R : B)]))
---
```

THM Not_Occ_qstar_forC {271}

```
∀V:OV, b:Qind, Y:OV List, R,B:OE.
(¬V onlist(OV) Y ⇒
 ¬(some [V] occur free in [R]) &
 ¬(some [V] occur free in [B]))
⇐⇒
¬(some [V] occur free in [(*_b Y | R : B)])
---
```

THM Not_Occ_qstar_forH {272}

```
∀V:OV, b:Qind, Y:OV List, R,B:OE.
V onlist(OV) Y ∨ ¬V onlist(OV) Y &
¬(some [V] occur free in [R]) &
¬(some [V] occur free in [B])
⇐⇒
¬(some [V] occur free in [(*_b Y | R : B)])
---
```

THM decidable__isl_of_lookup {273}

```
∀x:IDENT, L:env(T). Dec(isl(lookup_env(x,L)))
---
```

THM LookupOnNonListHead {274}

```
∀v:env(t), u:(IDENT × t), k:IDENT.
k=_2 1of(u) = false_2 ⇒
lookup_env(k,u.v) = lookup_env(k,v)
---
```

THM NotIslLookupIsInr {275}

```
∀l:env(t), k:IDENT.
¬_2 isl(lookup_env(k,l)) ⇒ lookup_env(k,l) = inr(·)
---
```

THM LookupOnFrontOfList {276}

```
∀l:env(T), k:IDENT, env:env(T).
isl(lookup_env(k,l)) ⇒
lookup_env(k,l @ env) = lookup_env(k,l)
---
```

THM OutlLookupInType {277}

```
∀k:IDENT, v:env(T).
isl(lookup_env(k,v)) ⇒ outl(lookup_env(k,v)) ∈ T
```

THM identofovar_characterization {278}

```
∀ov:OV. Ovar(identofovar(ov)) = ov ∈OV
```

THM is_a_ovar_on_OV {279}

```
∀x:OV. is_a_ovar(x)
```

THM is_a_ovar_characterization {280}

```
∀x:OE. IsOvar(x) ⇐⇒ is_a_ovar(x)
```

THM inr_y_in_NonOV {281}

```
∀y:i:OPIDS × j:ℕoesbtms(i) ⟶ OE × PossBV(i;j).
inr(y) ∈ OE ⇒ inr(y) ∈ NonOV
```

THM imax_on_nat {282}

```
∀a,b:ℕ. imax(a;b) ∈ ℕ
```

THM imax_gtr_than {283}

```
∀a,b,c:ℤ. a > imax(b;c) ⇐⇒ a > b & a > c
```

THM make_equality_on_OPIDS {284}

```
∀x,y:OPIDS. x=₂y ⇐⇒ x = y ∈ℤ
```

THM OPIDS_bool_N5_prop {285}

```
∀x,y:OPIDS. x=₂y ⇐⇒ x = y ∈ℕ5
```

THM not_ofalse_then_subterms {286}

```
∀nonov:NonOV.
¬opid_of_OE(nonov)=₂opid_f ⇒
0 ∈ ℕoesbtms(opid_of_OE(nonov))
```

THM opid_not_ofalse_then_subterms {287}

```
∀op:OPIDS. ¬op=₂opid_f ⇒ 0 ∈ ℕoesbtms(op)
```

THM opid_not_false_or_all_then_2_subterms {288}
$\forall$op:OPIDS.
$\neg$op=$_2$opid_f & $\neg$op=$_2$opid_all $\Rightarrow$ 1 $\in$ $\mathbb{N}$oesbtms(op)

THM OEcase_else_opid_is_oforall {289}
$\forall$n:OPIDS. $\neg$n=$_2$0 & $\neg$n=$_2$1 & $\neg$n=$_2$2 & $\neg$n=$_2$3 $\Rightarrow$ n=$_2$opid_all

THM opid_Ap_or_Imp_or_Eq_then_2_subterms {290}
$\forall$op:OPIDS. op=$_2$1 $\vee$ op=$_2$2 $\vee$ op=$_2$3 $\Rightarrow$ 1 $\in$ $\mathbb{N}$oesbtms(op)

THM opid_Eq_then_2_sbtms {291}
$\forall$op:OPIDS. op=$_2$3 $\Rightarrow$ 1 $\in$ $\mathbb{N}$oesbtms(op)

THM opid_Imp_then_2_sbtms {292}
$\forall$op:OPIDS. op=$_2$2 $\Rightarrow$ 1 $\in$ $\mathbb{N}$oesbtms(op)

THM opid_Ap_then_2_sbtms {293}
$\forall$op:OPIDS. op=$_2$1 $\Rightarrow$ 1 $\in$ $\mathbb{N}$oesbtms(op)

THM if_oforall_then_bv {294}
$\forall$nonov:NonOV.
opid_of_OE(nonov)=$_2$opid_all $\Rightarrow$
bv_of_OE(nonov,0) $\in$ IDENT

THM if_not_oforall_then_no_bv {295}
$\forall$op:OPIDS, posn:$\mathbb{N}$. $\neg$op=$_2$opid_all $\Rightarrow$ $\cdot$ $\in$ PossBV(op;posn)

THM ContraryCases1 {296}
$\forall$n,opid1,opid2:OPIDS.
n=$_2$opid1 = true$_2$ & n=$_2$opid2 = true$_2$ & $\neg$opid1 = opid2 $\in\mathbb{Z}$
$\Rightarrow$
False

THM HnumauxUnrollOpidall {297}

$\forall$n:OPIDS, ff:j:$\mathbb{N}$oesbtms(n) $\longrightarrow$ OE $\times$ PossBV(n;j).
opid_of_OE(mk_nonOV(n,ff))=$_2$ opid_all = true$_2$ $\Rightarrow$
hnum_aux(mk_nonOV(n,ff))
=
imax(ident2(bv_of_OE(mk_nonOV(n,ff),0));
     hnum_aux(nthsbtm(mk_nonOV(n,ff),0)))
$\in \mathbb{N}$

THM HnumauxUnrollOpidelse {298}

$\forall$n:OPIDS, ff:j:$\mathbb{N}$oesbtms(n) $\longrightarrow$ OE $\times$ PossBV(n;j).
opid_of_OE(mk_nonOV(n,ff))=$_2$ opid_f = false$_2$ &
opid_of_OE(mk_nonOV(n,ff))=$_2$ opid_all = false$_2$
$\Rightarrow$

hnum_aux(mk_nonOV(n,ff))
=
imax(hnum_aux(nthsbtm(mk_nonOV(n,ff),0));
     hnum_aux(nthsbtm(mk_nonOV(n,ff),1)))
$\in \mathbb{N}$

THM RenamebdvarsToIfThenElse {299}

$\forall$k:IDENT, num:$\mathbb{Z}$, l:env(IDENT).
rename(Ovar(k),num,l)
=
if isl(lookup_env(k,l)) $\longrightarrow$ Ovar(outl(lookup_env(k,l)))
else Ovar(k) fi

THM Othm_oeq_refl {300}

$\forall$P:OE, r:{ind, bool}, ep:(OV $\longrightarrow$ OTS), alpha:OE List.
(ep) [P] :: r & (ep) alpha :: bool $\Rightarrow$
(ep) alpha $\vdash$ P = P

THM Othm_oeq_sym {301}

$\forall$P,Q:OE, env:(OV $\longrightarrow$ OTS), A:OE List.
(env) A $\vdash$ P = Q $\Rightarrow$ (env) A $\vdash$ Q = P

$\forall$x:NonOV. $\neg$is_a_ovar(x)   THM is_a_ovar_on_NonOV {302}

THM Othmaux_monotonicity {303}

```
∀n:ℕ, exp:OE, env:env(OTS), assumps:OE List.
Othm_aux(exp,env,assumps,n) ⇒
(∀k:ℕ. (k ≥ n ) ⇒ Othm_aux(exp,env,assumps,k))
```

THM OthmMP {304}

```
∀e,f:OE, A:OE List, env:env(OTS).
(env) A ⊢ e ⇒ f & (env) A ⊢ e ⇒ (env) A ⊢ f
```

THM OthmArrowIntro {305}

```
∀e,f:OE, A:OE List, env:env(OTS).
(env) e.A ⊢ f ⇒ (env) A ⊢ e ⇒ f
```

THM OthmThin {306}

```
∀e:OE, A,sublist:OE List, env:env(OTS).
allonlist(OE; sublist; A) &
(env) sublist ⊢ e &
(env) A :: bool
⇒
(env) A ⊢ e
```

THM OthmCBV {307}

```
∀e,f:OE, A:OE List, env:env(OTS).
e =_alph(nil) f & (env) A ⊢ f ⇒ (env) A ⊢ e
```

THM OthmAllIntro {308}

```
∀e:OE, x:IDENT, A:OE List, env:env(OTS).
(update_env([x,ind],env)) A ⊢ e &
¬(some [Ovar(x)] occur free in A)
⇒
(env) A ⊢ (∀x|:e)
```

THM OthmAllElim {309}

```
∀e,f:OE, x:IDENT, A:OE List, env:env(OTS).
(env) A ⊢ (∀x|:e) & (env) f :: ind ⇒
(env) A ⊢ (e)[x := f]
```

```
                             THM OthmEnvsAgree {310}
∀e:OE, A:OE List, env,env2:env(OTS).
(env2) A ⊢ e &
(∀id:IDENT.
 (Ovar(id) occurs free in e) ⇒
 lookup_env(id,env) = lookup_env(id,env2))
⇒
(env) A ⊢ e
---

∀e:OE, A:OE List, env:env(OTS).        THM OthmAx {311}
Axiom(e,env) & (env) A :: bool ⇒ (env) A ⊢ e

∀e:OE, A:OE List, env:env(OTS).       THM OthmHyp {312}
e onlist(OE) A & (env) A :: bool ⇒ (env) A ⊢ e

∀P:OE, env:(OV → OTS).             THM OnotWfdelt {313}
(env) P :: bool ⇒ (env) ¬P :: bool

∀P,Q:OE, env:(OV → OTS).            THM OorWfdelt {314}
(env) P :: bool & (env) Q :: bool ⇒
(env) P ∨ Q :: bool

∀P,Q:OE, env:(OV → OTS).           THM OandWfdelt {315}
(env) P :: bool & (env) Q :: bool ⇒
(env) P ∧ Q :: bool

∀P,Q:OE, env:(OV → OTS).         THM OequivWfdelt {316}
(env) P :: bool & (env) Q :: bool ⇒
(env) P ≡ Q :: bool
---

∀e:NonOV.                THM OesbtmsUnrollOn2sbtms {317}
opid_of_OE(e)=₂0 = false₂ & opid_of_OE(e)=₂4 = false₂
⇒
oesbtms(opid_of_OE(e)) = 2 ∈ ℕ
---

                        THM OsubstReadyOfSubterms {318}
∀e:NonOV, n:ℤ, i:ℕoesbtms(opid_of_OE(e)).
Osubstn_ready(e,n) ⇒ Osubstn_ready(nthsbtm(e,i),n)
```

$\forall$x,y:IDENT. x=$_2$y $\Longleftrightarrow$ x = y THM AssertOfEqIdent {319}

$\forall$L:env(T), x:IDENT. THM LookupFailsIffInr {320}
$\neg$isl(lookup_env(x,L)) $\Longleftrightarrow$ lookup_env(x,L) = inr($\cdot$)

THM LookupFailsOnFrontOfList {321}
$\forall$L,ep:env(T), x:IDENT.
$\neg$isl(lookup_env(x,L)) $\Rightarrow$
lookup_env(x,L @ ep) = lookup_env(x,ep)

THM LookupWithEqualListFronts {322}
$\forall$x:IDENT, L1,L2,ep:env(T).
lookup_env(x,L1) = lookup_env(x,L2) $\Rightarrow$
lookup_env(x,L1 @ ep) = lookup_env(x,L2 @ ep) $\in$T+Unit

THM RenameOvarIsOvar {323}
$\forall$id:IDENT, n:$\mathbb{N}$, L:env(IDENT).
$\exists$id2:IDENT. rename(Ovar(id),n,L) = Ovar(id2)

THM RenamePreservesOpid {324}
$\forall$e:OE, L:env(IDENT), n:$\mathbb{N}$.
$\neg$IsOvar(e) $\Rightarrow$ opid_of_OE(e) = opid_of_OE(rename(e,n,L))

THM NotIsOvarMknonOV {325}
$\forall$m:OPIDS, ff:j:$\mathbb{N}$oesbtms(m) $\longrightarrow$ OE $\times$ PossBV(m;j).
$\neg$IsOvar(mk_nonOV(m,ff))

```
                      THM RenameUnrolling2sbtmCase {326}
∀m:OPIDS, ff:j:ℕ oesbtms(m) ⟶ OE × PossBV(m;j), n:ℕ
, L:env(IDENT).
opid_of_OE(mk_nonOV(m,ff))=_2 opid_f = false_2 &
opid_of_OE(mk_nonOV(m,ff))=_2 opid_all = false_2
⇒

rename(mk_nonOV(m,ff),n,L)
=
mk_nonOV(opid_of_OE(mk_nonOV(m,ff)), λposn.
if posn=_2 0 ⟶ <rename(nthsbtm(mk_nonOV(m,ff),0),n,L), ·>
else <rename(nthsbtm(mk_nonOV(m,ff),1),n,L), ·> fi)
---

                           THM RenameUnrollOpidAll {327}
∀m:OPIDS, ff:j:ℕ oesbtms(m) ⟶ OE × PossBV(m;j), n:ℕ
, L:env(IDENT).
m=_2 opid_all ⇒
rename(mk_nonOV(m,ff),n,L)
=
(∀<1of(bv_of_OE(mk_nonOV(m,ff),0)),n+1>
 |:rename(nthsbtm(mk_nonOV(m,ff),0),n+1,
          update_env([bv_of_OE(mk_nonOV(m,ff),0),
                      <1of(bv_of_OE(mk_nonOV(m,ff),0))
                      ,n+1>],
                     L)))
---

                         THM OwfdeltUnrollForOpid1 {328}
∀m:OPIDS, ff:j:ℕ oesbtms(m) ⟶ OE × PossBV(m;j), sig:OTS
, env:(OV ⟶ OTS).
m = 1 ∈ OPIDS ⇒
(env) mk_nonOV(m,ff) :: sig
=
((env) nthsbtm(mk_nonOV(m,ff),1) :: ind &
 (env)
 nthsbtm(mk_nonOV(m,ff),0) :: ots(1of(sig),2of(sig)+1))
---
```

THM OwfdeltUnrollForOpidImp {329}

```
∀m:OPIDS, ff:j:ℕ oesbtms(m) ⟶ OE × PossBV(m;j), sig:OTS
, env:(OV ⟶ OTS).
m = 2 ∈ OPIDS ⇒
(env) mk_nonOV(m,ff) :: sig
=
(sig = bool &
 (env) nthsbtm(mk_nonOV(m,ff),0) :: bool &
 (env) nthsbtm(mk_nonOV(m,ff),1) :: bool)
---
```

THM OwfdeltUnrollForOpidEq {330}

```
∀m:OPIDS, ff:j:ℕ oesbtms(m) ⟶ OE × PossBV(m;j), sig:OTS
, env:(OV ⟶ OTS).
m = 3 ∈ OPIDS ⇒
(env) mk_nonOV(m,ff) :: sig
=
(sig = bool &
 ((env) nthsbtm(mk_nonOV(m,ff),0) :: bool &
  (env) nthsbtm(mk_nonOV(m,ff),1) :: bool
  ∨ (env) nthsbtm(mk_nonOV(m,ff),0) :: ind &
  (env) nthsbtm(mk_nonOV(m,ff),1) :: ind))
---
```

THM OwfdeltUnrollForOpidAll {331}

```
∀m:OPIDS, ff:j:ℕ oesbtms(m) ⟶ OE × PossBV(m;j), sig:OTS
, env:(OV ⟶ OTS).
m = 4 ∈ OPIDS ⇒
(env) mk_nonOV(m,ff) :: sig
=
(sig = bool &
 (env[Ovar(2of(fn_of_OE(mk_nonOV(m,ff))(0))){eq_OV}:=
      ind])
 nthsbtm(mk_nonOV(m,ff),0) :: bool)
---
```

THM InlNeqInrInOE {332}

```
∀x:IDENT, y:i:OPIDS × j:ℕ oesbtms(i) ⟶ OE × PossBV(i;j).
¬inl(x) = inr(y) ∈ OE
---
```

```
∀E,F:OE.                          THM OvarNotMknonOV {333}
(∃x:IDENT. E = Ovar(x)) &
(∃m:OPIDS, ff:j:ℕoesbtms(m) → OE × PossBV(m;j).
 F = mk_nonOV(m,ff))
⇒
¬E = F
---

                              THM OvarNotMknonOVPt2 {334}
∀E:OV, F:NonOV. ¬E = F ∈OE

                            THM RenameNonOVIsNonOV {335}
∀m:OPIDS, ff:j:ℕoesbtms(m) → OE × PossBV(m;j), n:ℕ
, L:env(IDENT). rename(mk_nonOV(m,ff),n,L) ∈ NonOV
---

       THM NotOpidOfRenameNonOVThenNotOpidOfNonOV {336}
∀m:OPIDS, ff:j:ℕoesbtms(m) → OE × PossBV(m;j), op:OPIDS
, L:env(IDENT), n:ℕ.
opid_of_OE(rename(mk_nonOV(m,ff),n,L))=₂op = false₂ ⇒
opid_of_OE(mk_nonOV(m,ff))=₂op = false₂
---

                            THM All2ofsGtrToOnlist {337}
∀L:(IDENT × IDENT) List, n:ℕ, x,y:IDENT.
all_2ofs_gtr_n(L,n) & <x,y> onlist(IDENT × IDENT) L ⇒
ident2(y) > n
---

∀x,u:T, v:T List.                 THM OnlistForTails {338}
¬x = u & x onlist(T) u.v ⇒ x onlist(T) v

                           THM IdentOnlistForTails {339}
∀x,u:IDENT, v:IDENT List.
¬x = u ⇒ x onlist(IDENT) u.v ⇒ x onlist(IDENT) v
---

                                   THM IslToOnlist {340}
∀L:(IDENT × IDENT) List, k:IDENT.
isl(lookup_env(k,L)) ⇒
<k,outl(lookup_env(k,L))> onlist(IDENT × IDENT) L
---
```

```
                               THM LookupToOnlist {341}
∀L:(IDENT × IDENT) List, x,y:IDENT.
lookup_env(x,L) = inl(y) ∈ IDENT+Unit ⇒
<x,y> onlist(IDENT × IDENT) L
---

∀x,y:IDENT. Dec(x = y)    THM decidable__eq_ident {342}

                 THM decidable__equalpairofidents {343}
∀x,u:(IDENT × IDENT). Dec(x = u)

                      THM No2ofsEqualForListTails {344}
∀u:(IDENT × IDENT), v:(IDENT × IDENT) List.
no_2ofs_equal(u.v) ⇒ no_2ofs_equal(v)

                   THM All2ofsDiffThenNo2ofsEqual {345}
∀L:(IDENT × IDENT) List.
all_2ofs_diff(L) ⇒ no_2ofs_equal(L)
---

∀L1,L2:env(T).         THM LookupsEqIffAlwaysSame {346}
(∀x:IDENT. lookup_env(x,L1) = lookup_env(x,L2))
⇐⇒
(∀x:IDENT, tau:T+Unit.
 lookup_env(x,L1) = tau ⇐⇒ lookup_env(x,L2) = tau)
---

∀L1,L2:env(T).     THM LookupsEqIffAlwaysSameInls {347}
(∀x:IDENT. lookup_env(x,L1) = lookup_env(x,L2))
⇐⇒
(∀x:IDENT, tau:T.
 lookup_env(x,L1) = inl(tau)
 ⇐⇒
 lookup_env(x,L2) = inl(tau))
---

                         THM NotEqOvIfDiffHnumaux {348}
∀z,v:OV. ¬hnum_aux(z) = hnum_aux(v) ⇒ ¬z = v

                                THM Otseup_reduce {349}
∀x,y:OV, sig:OTS, ∈:(OV → OTS).
¬x = y ⇒ ∈[x ← sig](y) = ∈(y)
```

```
                               THM Otseup_OVmatch {350}
∀x,y:OV, sig:OTS, ∈:(OV⟶OTS).
x = y ⇒ ∈[x⟵ sig](y) = sig

{ind, bool} ⊆ OTS                  THM IndBool_inc {351}
---

                        THM OtseAgree_on_freevars {352}
∀E:OE List, ∈1,∈2:(OV⟶OTS), sig:OTS.
(∀x:OV.
 (some [x] occur free in E) ⇒ (∈1,∈2 agree on x))
⇒
(∈1) E :: sig ⇒ (∈2) E :: sig
---

                         THM Not_Occ_to_OtseAgree {353}
∀x:OV, L:OE List, ∈:(OV⟶OTS), sig:OTS.
¬(some [x] occur free in L) ⇒
(∀y:OV.
 (some [y] occur free in L) ⇒
 (∈,∈[x⟵ sig] agree on y))
---

                                THM OtseAgree_sym {354}
∀ ∈:(OV⟶OTS), x:OV, f:(OV⟶OTS).
(∈,f agree on x) ⇐⇒ (f,∈ agree on x)
---

            THM Not_Occ_to_OtseAgree_on_firstOtse {355}
∀x:OV, L:OE List, ∈:(OV⟶OTS), sig:OTS.
¬(some [x] occur free in L) ⇒
(∀y:OV.
 (some [y] occur free in L) ⇒
 (∈[x⟵ sig],∈ agree on y))
---

                               THM Otseup1var_sym {356}
∀x,y:OV, L:OE List, t,sig2,sig1:OTS, ∈:(OV⟶OTS).
¬x = y ⇒
((∈[x⟵ sig1][y⟵ sig2]) L :: t
 ⇐⇒
 (∈[y⟵ sig2][x⟵ sig1]) L :: t)
---
```

```
                    THM Otses_eq_dup_updates {357}
∀x:OV, sig:OTS, ep:(OV ⟶ OTS).
ep[x ⟵ sig][x ⟵ sig] = ep[x ⟵ sig]
---

                    THM Otseup1var_sym_for_bckchn {358}
∀x,y:OV, L:OE List, t,sig2,sig1:OTS, ep:(OV ⟶ OTS).
¬x = y ⇒
(ep[x ⟵ sig1][y ⟵ sig2]) L :: t ⇒
(ep[y ⟵ sig2][x ⟵ sig1]) L :: t
---

                    THM Otseup1var_sym_in_OtseAgree {359}
∀x,y:OV, sig2,sig1:OTS, ∈:(OV ⟶ OTS), v:OV, f:(OV ⟶ OTS).
¬x = y ⇒
((∈[x ⟵ sig1][y ⟵ sig2],f agree on v)
 ⇐⇒
 (∈[y ⟵ sig2][x ⟵ sig1],f agree on v))
---

                    THM OtseAgree_non_updated_vars {360}
∀x:OV, A:OV List, ∈:(OV ⟶ OTS), sig:OTS.
¬x onlist(OV) A ⇒ (∈,∈[A ⟵ sig] agree on x)
---

                    THM Leib_quant_body {361}
∀Xs:OV List, α:OE List, R,A,B:OE, ∈:(OV ⟶ OTS)
, r:{ind, bool}, P:OE, Z:OV, b:Qind.
¬(some Xs occur free in α) &
(∈[Xs ⟵ ind]) α ⊢ R ⇒ A = B &
(∈[Xs ⟵ ind]) [A; B] :: r &
(∈[Xs ⟵ ind][[Z] ⟵ r]) [P] :: bool
⇒

(∈) α ⊢ (*_b Xs | R : (P)[Z := A])
        =
        (*_b Xs | R : (P)[Z := B])
---
```

```
                                THM Leib_quant_range {362}
∀b:Qind, Xs:OV List, R:OE, Z:OV, A,P,B:OE, ∈:(OV ⟶ OTS)
, α:OE List, r:{ind, bool}.
¬(some Xs occur free in α) &
(∈[Xs ⟵ ind]) α ⊢ A = B &
(∈[Xs ⟵ ind]) [P] :: bool &
(∈[Xs ⟵ ind]) [A; B] :: r &
(∈[Xs ⟵ ind][[Z] ⟵ r]) [R] :: bool
⇒

(∈) α ⊢ (*_b Xs | (R)[Z := A] : P)
         =
         (*_b Xs | (R)[Z := B] : P)
---

                                     THM Leib_simple {363}
∀E,A,B:OE, ∈:(OV ⟶ OTS), α:OE List, r,r':{ind, bool}
, Z:OV.
(∈) α ⊢ A = B & (∈) [A; B] :: r & (∈[[Z] ⟵ r]) [E] :: r'
⇒
(∈) α ⊢ (E)[Z := A] = (E)[Z := B]
---

                                  THM One_point_rule {364}
∀x:OV, E:OE, α:OE List, ∈:(OV ⟶ OTS), b:Qind, P:OE.
¬(some [x] occur free in [E]) &
(∈) α :: bool & (∈) [E] :: ind &
(∈[x ⟵ ind]) [P] :: bool
⇒
(∈) α ⊢ (*_b [x] | x = E : P) = (P)[x := E]
---
```

THM Nesting {365}

```
∀y:OV, R:OE, α:OE List, ∈:(OV → OTS), P,Q:OE, x:OV
, b:Qind.
¬(some [y] occur free in [R]) &
(∈) α :: bool &
(∈[[x; y] ← ind]) [P; Q; R] :: bool
⇒

(∈) α ⊢ (*_b [x; y] | R ∧ Q : P)
         =
         (*_b [x] | R : (*_b [y] | Q : P))
---
```

THM Nesting_varsym {366}

```
∀y:OV, R:OE, α:OE List, ∈:(OV → OTS), P,Q:OE, x:OV
, b:Qind.
¬(some [x] occur free in [R]) &
(∈) α :: bool &
(∈[[x; y] ← ind]) [P; Q; R] :: bool
⇒

(∈) α ⊢ (*_b [x; y] | R ∧ Q : P)
         =
         (*_b [y] | R : (*_b [x] | Q : P))
---
```

THM Substitution_3_84a {367}

```
∀ α:OE List, ∈:(OV → OTS), e,f:OE, r:{ind, bool}, E:OE
, z:OV.
(∈) α :: bool &
(∈) [e; f] :: r &
(∈[[z] ← r]) [E] :: bool
⇒
(∈) α ⊢ (E)[z := f] ∧ e = f = (E)[z := e] ∧ e = f
---
```

THM osubst1_var_same {368}

```
∀x:OV, E:OE. (x)[x := E] = E
---
```

THM Othm_Osubst1_unroll_notocc {369}

$\forall\, \boldsymbol{\alpha}$:OE List, $\in$:(OV $\longrightarrow$ OTS), P:OE, r:{ind, bool}, x:OV
, E:OE.
($\in$) $\boldsymbol{\alpha}$ :: bool & ($\in$) [P] :: r &
$\neg$(some [x] occur free in [P])
$\Rightarrow$
($\in$) $\boldsymbol{\alpha}$ $\vdash$ (P)[x := E] = P

THM Osubst1_rep_var_with_self {370}

$\forall \in$:(OV $\longrightarrow$ OTS), $\boldsymbol{\alpha}$:OE List, E:OE, x:OV.
($\in$) $\boldsymbol{\alpha}$ $\vdash$ (E)[x := x] = E

$\forall$X:OV, P,E:OE. THM Osubst1_unroll_notocc {371}
$\neg$(some [X] occur free in [P]) $\Rightarrow$ (P)[X := E] = P

THM Arrowintro_for_Leib {372}

$\forall$R:OE, env:(OV $\longrightarrow$ OTS), T:OE, A:OE List.
(env) [R] :: bool & (env) A $\vdash$ T $\Rightarrow$ (env) A $\vdash$ R $\Rightarrow$ T

$\forall$P,Q:OE, env:(OV $\longrightarrow$ OTS), A:OE List. THM oeq_sym {373}
(env) A $\vdash$ P = Q $\Rightarrow$ (env) A $\vdash$ Q = P

$\forall$x,h:T, t:T List. THM onlist_step_v2 {374}
x onlist(T) h.t $\Rightarrow$ x = h $\vee$ x onlist(T) t

$\forall$X,H:T, L:T List. THM notonlist_step_v2 {375}
$\neg$X onlist(T) H.L $\Rightarrow$ $\neg$X = H & $\neg$X onlist(T) L

$\forall$X:T. $\neg$X onlist(T) nil THM notonlist_on_nil {376}

THM onlist_on_nil_imp_false {377}

$\forall$x:T. x onlist(T) nil $\Rightarrow$ False

$\forall$X,H:T, L:T List. THM notonlist_step {378}
$\neg$H = X & $\neg$X onlist(T) L $\Rightarrow$ $\neg$X onlist(T) H.L

$\forall$x,h:T, t:T List. THM onlist_unroll {379}
x onlist(T) h.t $\Longleftrightarrow$ x = h $\vee$ x onlist(T) t

THM onlist_on_zip_nil {380}
$\forall$X:T, L:T List. X onlist(T) zip(nil,L) $\Rightarrow$ False

$\forall$h:T, t:T List, x:T. THM notonlist_gen_unroll {381}
Not_onlist(x,h.t:T) $\Longleftrightarrow$ $\neg$x = h & Not_onlist(x,t:T)

THM notonlist_gen_on_nil {382}
$\forall$x:T. Not_onlist(x,nil:T)

THM distinct_elts_gen_unroll {383}
$\forall$h1:T, t1:T List, h:T.
Distinct_elts([h; h1/ t1],T)
$\Longleftrightarrow$
Not_onlist(h,h1.t1:T) & Distinct_elts(h1.t1,T)

THM distinct_elts_gen_on_singletonlist {384}
$\forall$X:T. Distinct_elts([X],T)

THM otseup_var_list_unroll {385}
$\forall$H:OV, T:OV List, ep:(OV$\longrightarrow$OTS), sig:OTS.
ep[H.T$\longleftarrow$ sig] = ep[H$\longleftarrow$ sig][T$\longleftarrow$ sig]

THM otseup_var_list_on_nil {386}
$\forall$ep:(OV$\longrightarrow$OTS), sig:OTS. ep[nil$\longleftarrow$ sig] = ep

THM otseupdate_2lists_unroll {387}
$\forall$h1:OV, t1:OV List, ep:(OV$\longrightarrow$OTS), h2:OTS, t2:OTS List.
ep[h1.t1$\longleftarrow$ h2.t2] = ep[h1$\longleftarrow$ h2][t1$\longleftarrow$ t2]

THM otseupdate_2lists_on_nil_vars {388}
$\forall$ep:(OV$\longrightarrow$OTS), Ss:OTS List. ep[nil$\longleftarrow$ Ss] = ep

THM Owfd_otrue_on_osubst1 {389}
$\forall$sig:OTS, V:OV, E:OE, ep:(OV$\longrightarrow$OTS).
(ep) [t] :: sig $\Rightarrow$ (ep) [(t)[V := E]] :: sig

```
                        THM Owfd_ofalse_on_osubst1 {390}
∀sig:OTS, V:OV, E:OE, ep:(OV⟶OTS).
(ep) [f] :: sig ⇒ (ep) [(f)[V := E]] :: sig
```

```
                      THM Owfd_unitqind_on_osubst1 {391}
∀b:Qind, sig:OTS, V:OV, E:OE, ep:(OV⟶OTS).
(ep) [u_b] :: sig ⇒ (ep) [(u_b)[V := E]] :: sig
```

```
                         THM Owfd_otrue_on_osubstn {392}
∀Vs:OV List, Es:OE List, sig:OTS, ep:(OV⟶OTS).
(ep) [t] :: sig ⇒ (ep) [t[Vs := Es]] :: sig
```

```
                        THM Owfd_ofalse_on_osubstn {393}
∀Vs:OV List, Es:OE List, sig:OTS, ep:(OV⟶OTS).
(ep) [f] :: sig ⇒ (ep) [f[Vs := Es]] :: sig
```

```
                      THM Owfd_unitqind_on_osubstn {394}
∀b:Qind, Vs:OV List, Es:OE List, sig:OTS, ep:(OV⟶OTS).
(ep) [u_b] :: sig ⇒ (ep) [u_b[Vs := Es]] :: sig
```

```
                          THM zip_rewrite_nillist1 {395}
∀L:T List. zip(nil,L) = nil ∈T List
```

```
                     THM zip_rewrite_conslists_gen {396}
∀h1:T, t1:T List, h2:T2, t2:T2 List.
zip(h1.t1,h2.t2) = <h1,h2>.zip(t1,t2)
---
```

```
                     THM Othm_refl_for_AC_use_ONLY {397}
∀A,B:OE, ∈:(OV⟶OTS), α:OE List.
A = B & (∈) [A = B] :: bool & (∈) α :: bool ⇒
(∈) α ⊢ A = B
---
```

```
                     THM Same_envs_agree_in_conseq {398}
∀L:OE List, ep:(OV⟶OTS), x:OV.
(some [x] occur free in L) ⇒ (ep,ep agree on x)
```

```
                               THM Same_envs_agree {399}
∀ep:(OV⟶OTS), x:OV. (ep,ep agree on x)
```

```
---

                  THM Envagree_unroll_outmost_var {400}
∀ep1:(OV → OTS), v:OV, ep2:(OV → OTS), x:OV, sig:OTS.
(ep1,ep2 agree on v) ⇒
(ep1[x ← sig],ep2[x ← sig] agree on v)
---

               THM Envagree_unroll_outmost_var_V2 {401}
∀L:OE List, ep1,ep2:(OV → OTS), v:OV, sig:OTS.
(∀x:OV.
 (some [x] occur free in L) ⇒ (ep1,ep2 agree on x))
⇒

(∀x:OV.
 (some [x] occur free in L) ⇒
 (ep1[v ← sig],ep2[v ← sig] agree on x))
---

          THM Envagree_lemma_for_ltd_transitivity {402}
∀L:OE List, e1,e2:(OV → OTS), v:OV, sig:OTS.
(∀x:OV.
 (some [x] occur free in L) ⇒ (e1,e2 agree on x)) &
(∀x:OV.
 (some [x] occur free in L) ⇒
 (e2,e2[v ← sig] agree on x))
⇒

(∀x:OV.
 (some [x] occur free in L) ⇒
 (e1,e2[v ← sig] agree on x))
---

                          THM assoc_mod_iff_lemma {403}
∀A,B:OE, ∈:(OV → OTS), α:OE List.
equal_mod_AC_props(A;B) &
(∈) [A = B] :: bool &
(∈) α :: bool
⇒
(∈) α ⊢ A = B
---
```

```
                    THM Othm_oimp_if_othm_conseq {404}
∀P:OE, ep:(OV ⟶ OTS), Q:OE, alpha:OE List.
(ep) [P] :: bool & (ep) alpha ⊢ Q ⇒
(ep) alpha ⊢ P ⇒ Q
---

∀P:OE, V:OV, E,Q:OE.      THM Osubst1_unroll_oimp {405}
((P)[V := E] ⇒ (Q)[V := E]) = (P ⇒ Q)[V := E]
---

      THM Osubst1_unroll_qstar_capture_permitting {406}
∀b:Qind, X:OV List, R:OE, V:OV, E,B:OE.
(*_b X | (R)[V := E] : (B)[V := E])
=
((*_b X | R : B))[V := E]
---

∀f,x:OE, V:OV, E:OE.      THM Osubst1_unroll_obap {407}
(f(x))[V := E] = (f)[V := E]((x)[V := E])

∀P:OE, V:OV, E,Q:OE.       THM Osubst1_unroll_oeq {408}
(P)[V := E] = (Q)[V := E] = (P = Q)[V := E]

                           THM Osubst1_unroll_onot {409}
∀P:OE, V:OV, E:OE. (¬(P)[V := E]) = (¬P)[V := E]

∀P:OE, V:OV, E,Q:OE.    THM Osubst1_unroll_oequiv {410}
(P)[V := E] ≡ (Q)[V := E] = (P ≡ Q)[V := E]

∀P:OE, V:OV, E,Q:OE.       THM Osubst1_unroll_oor {411}
(P)[V := E] ∨ (Q)[V := E] = (P ∨ Q)[V := E]

∀P:OE, V:OV, E,Q:OE.      THM Osubst1_unroll_oand {412}
(P)[V := E] ∧ (Q)[V := E] = (P ∧ Q)[V := E]

                           THM Osubst1_unroll_qop {413}
∀b:Qind, P:OE, V:OV, E,Q:OE.
(P)[V := E] *_b (Q)[V := E] = (P *_b Q)[V := E]

                      THM Osubst1_unroll_unitqind {414}
∀b:Qind, V:OV, E:OE. u_b = (u_b)[V := E]
```

THM Osubst1_unroll_otrue {415}

$\forall$V:OV, E:OE. t = (t)[V := E]

THM Osubst1_unroll_ofalse {416}

$\forall$V:OV, E:OE. f = (f)[V := E]

THM Exist_new_OVs {417}

$\forall$L:OE List. $\exists$x:OV. $\neg$(some [x] occur free in L)

THM Othm_Oeq_trans {418}

$\forall$p,p1:OE, ep:(OV $\longrightarrow$ OTS), a:OE List, q:OE.
(ep) a $\vdash$ p = p1 & (ep) a $\vdash$ p1 = q $\Rightarrow$ (ep) a $\vdash$ p = q

THM Owfd_over_Onlist {419}

$\forall$Es:OE List, sig:OTS, $\in$:(OV $\longrightarrow$ OTS), E:OE.
($\in$) Es :: sig & E onlist(OE) Es $\Rightarrow$ ($\in$) [E] :: sig
---

$\forall$P,Q:OE, $\in$:(OV $\longrightarrow$ OTS). THM Owfd_oeq {420}
($\in$) [P = Q] :: bool
$\Leftarrow\Rightarrow$
($\exists$r:{ind, bool}. ($\in$) [P] :: r & ($\in$) [Q] :: r)

THM Owfd_oeq2 {421}

$\forall$P:OE, r:{ind, bool}, $\in$:(OV $\longrightarrow$ OTS), Q:OE.
($\in$) [P] :: r & ($\in$) [Q] :: r $\Rightarrow$ ($\in$) [P = Q] :: bool
---

$\forall$P:OE, $\in$:(OV $\longrightarrow$ OTS), Q:OE. THM Owfd_oeq3 {422}
($\in$) [P] :: ind & ($\in$) [Q] :: ind $\vee$ ($\in$) [P] :: bool &
($\in$) [Q] :: bool
$\Leftarrow\Rightarrow$
($\in$) [P = Q] :: bool

$\forall$sig:OTS, P,Q:OE, $\in$:(OV $\longrightarrow$ OTS). THM Owfd_and {423}
($\in$) [P $\wedge$ Q] :: sig
$\Leftarrow\Rightarrow$
($\in$) [P] :: bool & ($\in$) [Q] :: bool & sig = bool

```
∀sig:OTS, P,Q:OE, ∈:(OV ⟶ OTS).         THM Owfd_or {424}
(∈) [P ∨ Q] :: sig
⇐⇒
(∈) [P] :: bool & (∈) [Q] :: bool & sig = bool
---

                                       THM Owfd_false {425}
∀sig:OTS, ∈:(OV ⟶ OTS). (∈) [f] :: sig ⇐⇒ sig = bool
---

∀F,x:OE, sig:OTS, ∈:(OV ⟶ OTS).       THM Owfd_obap {426}
(∈) [F(x)] :: sig
⇐⇒
(∈) [F] :: ots(1of(sig),2of(sig)+1) & (∈) [x] :: ind
---

∀P:OE, sig:OTS, ∈:(OV ⟶ OTS).          THM Owfd_not {427}
(∈) [¬P] :: sig ⇐⇒ (∈) [P] :: bool & sig = bool
---

∀sig:OTS, P,Q:OE, ∈:(OV ⟶ OTS).     THM Owfd_oequiv {428}
(∈) [P ≡ Q] :: sig
⇐⇒
(∈) [P] :: bool & (∈) [Q] :: bool & sig = bool
---

∀sig:OTS, P,Q:OE, ∈:(OV ⟶ OTS).       THM Owfd_oimp {429}
(∈) [P ⇒ Q] :: sig
⇐⇒
(∈) [P] :: bool & (∈) [Q] :: bool & sig = bool
---

                                       THM Owfd_qstar {430}
∀b:Qind, X:OV List, R,P:OE, sig:OTS, ∈:(OV ⟶ OTS).
(∈) [(*_b X | R : P)] :: sig
⇐⇒
(∈[X ⟵ ind]) [R] :: bool & (∈[X ⟵ ind]) [P] :: bool &
sig = bool
---

∀V:OV, sig:OTS, ∈:(OV ⟶ OTS).        THM Owfd_on_OV {431}
(∈) [V] :: sig ⇐⇒ ∈(V) = sig
```

```
                        THM Owfd_otseup1_on_OV {432}
∀x:OV, sig:OTS, ep:(OV⟶OTS). (ep[x⟵ sig]) [x] :: sig

              THM Owfd_otseup1_unroll_diffvars {433}
∀x,y:OV, sig2:OTS, ep:(OV⟶OTS), sig:OTS.
¬x = y ⇒ (ep) [y] :: sig2 ⇒ (ep[x⟵ sig]) [y] :: sig2
---

                             THM Owfd_osubst1 {434}
∀E:OE, v:OV, p:OE, sig:OTS, ∈:(OV⟶OTS), sig2:OTS.
(∈) [p] :: sig2 & (∈[v⟵ sig2]) [E] :: sig ⇒
(∈) [(E)[v := p]] :: sig
---

                          THM Owfdelt_osubst1 {435}
∀E:OE, v:OV, p:OE, sig:OTS, ∈:(OV⟶OTS), sig2:OTS.
(∈) p :: sig2 & (∈[v⟵ sig2]) E :: sig ⇒
(∈) (E)[v := p] :: sig
---

                          THM Owfd_to_Owfdelt {436}
∀A:OE, sig:OTS, ∈:(OV⟶OTS).
(∈) A :: sig ⟸⟹ (∈) [A] :: sig

                         THM Owfd_list_unroll {437}
∀L:OE List, A:OE, sig:OTS, ∈:(OV⟶OTS).
(∈) [A] :: sig & (∈) L :: sig ⟸⟹ (∈) A.L :: sig

∀P:OE, ∈:(OV⟶OTS), Q:OE.     THM Owfd_oequiv_Vif {438}
(∈) [P] :: bool & (∈) [Q] :: bool ⇒
(∈) [P ≡ Q] :: bool

∀P:OE, ∈:(OV⟶OTS), Q:OE.       THM Owfd_oand_Vif {439}
(∈) [P] :: bool & (∈) [Q] :: bool ⇒
(∈) [P ∧ Q] :: bool

∀P:OE, ∈:(OV⟶OTS), Q:OE.       THM Owfd_orCD_Vif {440}
(∈) [P] :: bool & (∈) [Q] :: bool ⇒
(∈) [P ∨ Q] :: bool

                           THM Owfd_notCD_Vif {441}
∀P:OE, ∈:(OV⟶OTS). (∈) [P] :: bool ⇒ (∈) [¬P] :: bool
```

```
∀P:OE, ∈:(OV ⟶ OTS), Q:OE.      THM Owfd_impCD_Vif {442}
(∈) [P] :: bool & (∈) [Q] :: bool ⇒
(∈) [P ⇒ Q] :: bool

                                THM Owfd_true_Vif {443}
∀ ∈:(OV ⟶ OTS). (∈) [t] :: bool

                            THM Owfd_unitqind_Vif {444}
∀b:Qind, ∈:(OV ⟶ OTS). (∈) [u_b] :: bool

∀P,Q:OE, b:Qind, ∈:(OV ⟶ OTS).    THM Owfd_qop_Vif {445}
(∈) [P] :: bool & (∈) [Q] :: bool ⇒
(∈) [P *_b Q] :: bool

                               THM Owfd_false_Vif {446}
∀ ∈:(OV ⟶ OTS). (∈) [f] :: bool
---

                                THM Owfd_obap_Vif {447}
∀F,x:OE, sig:OTS, ∈:(OV ⟶ OTS).
(∈) [F] :: ots(1of(sig),2of(sig)+1) & (∈) [x] :: ind
⇒
(∈) [F(x)] :: sig
---

∀P,Q:OE, ∈:(OV ⟶ OTS).            THM Owfd_oeq_Vif {448}
(∃r:{ind, bool}. (∈) [P] :: r & (∈) [Q] :: r) ⇒
(∈) [P = Q] :: bool
---

                               THM Owfd_qstar_Vif {449}
∀b:Qind, X:OV List, R,P:OE, ∈:(OV ⟶ OTS).
(∈[X ⟵ ind]) [R] :: bool & (∈[X ⟵ ind]) [P] :: bool ⇒
(∈) [(*_b X | R : P)] :: bool
---

                      THM Owfd_osubst1var_CDlemma {450}
∀P:OE, T:OTS, V:OV, tau:OTS, env:(OV ⟶ OTS), E:OE.
(env[V ⟵ tau]) [P] :: T & (env) [E] :: tau ⇒
(env) [(P)[V := E]] :: T
---
```

```
                    THM Owfd_bool_if_Othm_assump {451}
∀expn:OE, ep:(OV⟶OTS), L:OE List.
(ep) L ⊢ expn ⇒ (ep) L :: bool
---

                        THM Owfd_osubstn_CDlemma {452}
∀Vs:OV List, Es:OE List, Sigs:OTS List, P:OE, T:OTS
, env:(OV⟶OTS).
Distinct_elts(Vs,OV) & ||Vs|| = ||Es|| ∈ ℤ &
||Sigs|| = ||Es|| ∈ ℤ &
(env[Vs⟵ Sigs]) [P] :: T &
(∀ESigpair:(OE × OTS).
 ESigpair onlist(OE × OTS) zip(Es,Sigs) ⇒
 (ESigpair/E,Sig.(env) [E] :: Sig))
⇒
(env) [P[Vs := Es]] :: T
---

                               THM ChgD_step1_V2 {453}
∀x,y,f,f⁻:OV, α:OE List, R,P:OE, ∈:(OV⟶OTS), b:Qind.
(∀ [x; y] |:x = f(y) ≡ y = f⁻(x)) onlist(OE) α &
(∈[[x]⟵ ind]) [R; P] :: bool &
(∈) α :: bool &
¬(some [y] occur free in [R; P]) &
¬x = y &
¬(some [x; y] occur free in α) &
¬x = f &
(∈[[x; y]⟵ ind]) [f; f⁻] :: ind(1)
⇒

(∈) α ⊢ (*_b [y] | (R)[x := f(y)] : (P)[x := f(y)])
         =
         (*_b [y] | (R)[x := f(y)] :
              (*_b [x] | x = f(y) : P))
---
```

THM ChgD_step2 {454}

$\forall$x,y,f,f$^-$:OV, $\boldsymbol{\alpha}$:OE List, R,P:OE, $\in$:(OV$\rightarrow$OTS), b:Qind.
($\forall$ [x; y] |:x = f(y) $\equiv$ y = f$^-$(x)) onlist(OE) $\boldsymbol{\alpha}$ &
($\in$[[x]$\leftarrow$ ind]) [R; P] :: bool &
($\in$) $\boldsymbol{\alpha}$ :: bool &
$\neg$(some [y] occur free in [R; P]) &
$\neg$x = y & $\neg$y = f &
$\neg$(some [x; y] occur free in $\boldsymbol{\alpha}$) &
$\neg$x = f &
($\in$[[x; y]$\leftarrow$ ind]) [f; f$^-$] :: ind(1)
$\Rightarrow$

($\in$) $\boldsymbol{\alpha}$ $\vdash$ (*_b [y] | (R)[x := f(y)] :
(*_b [x] | x = f(y) : P))
=
(*_b [x; y] | (R)[x := f(y)] $\wedge$ x = f(y) : P)

THM ChgD_step3 {455}

$\forall$x,y,f,f$^-$:OV, $\boldsymbol{\alpha}$:OE List, R,P:OE, $\in$:(OV$\rightarrow$OTS), b:Qind.
($\forall$ [x; y] |:x = f(y) $\equiv$ y = f$^-$(x)) onlist(OE) $\boldsymbol{\alpha}$ &
($\in$[[x]$\leftarrow$ ind]) [R; P] :: bool &
($\in$) $\boldsymbol{\alpha}$ :: bool &
$\neg$(some [y] occur free in [R; P]) &
$\neg$x = y & $\neg$y = f &
$\neg$(some [x; y] occur free in $\boldsymbol{\alpha}$) &
$\neg$x = f &
($\in$[[x; y]$\leftarrow$ ind]) [f; f$^-$] :: ind(1)
$\Rightarrow$

($\in$) $\boldsymbol{\alpha}$ $\vdash$ (*_b [x; y] | (R)[x := f(y)] $\wedge$ x = f(y) : P)
=
(*_b [x; y] | (R)[x := x] $\wedge$ x = f(y) : P)

```
                                    THM ChgD_step4_1 {456}
∀x,y,f,f⁻:OV, α:OE List, R,P:OE, ∈:(OV⟶OTS), b:Qind.
(∀ [x; y] |:x = f(y) ≡ y = f⁻(x)) onlist(OE) α &
(∈[[x]⟵ ind]) [R; P] :: bool &
(∈) α :: bool &
¬(some [y] occur free in [R; P]) &
¬x = y & ¬y = f &
¬(some [x; y] occur free in α) &
¬x = f &
(∈[[x; y]⟵ ind]) [f; f⁻] :: ind(1)
⇒

(∈) α ⊢ (*_b [x; y] | (R)[x := x] ∧ x = f(y) : P)
         =
         (*_b [x; y] | R ∧ x = f(y) : P)
---

                                    THM ChgD_step4_2 {457}
∀x,y,f,f⁻:OV, α:OE List, R,P:OE, ∈:(OV⟶OTS), b:Qind.
(∀ [x; y] |:x = f(y) ≡ y = f⁻(x)) onlist(OE) α &
(∈[[x]⟵ ind]) [R; P] :: bool &
(∈) α :: bool &
¬(some [y] occur free in [R; P]) &
¬x = y & ¬y = f &
¬(some [x; y] occur free in α) &
¬x = f &
(∈[[x; y]⟵ ind]) [f; f⁻] :: ind(1)
⇒

(∈) α ⊢ (*_b [x; y] | R ∧ x = f(y) : P)
         =
         (*_b [x] | R : (*_b [y] | x = f(y) : P))
---
```

THM Assump_to_thm_for_ChgD_step6 {458}

$\forall$x,y,f,f$^-$:OV, $\boldsymbol{\alpha}$:OE List, $\in$:(OV $\longrightarrow$ OTS).
($\forall$ [x; y] |:x = f(y) $\equiv$ y = f$^-$(x)) onlist(OE) $\boldsymbol{\alpha}$ &
($\in$) $\boldsymbol{\alpha}$ :: bool &
$\neg$(some [x; y] occur free in $\boldsymbol{\alpha}$)
$\Rightarrow$
($\in$[[x; y]$\longleftarrow$ ind]) $\boldsymbol{\alpha}$ $\vdash$ x = f(y) = y = f$^-$(x)

THM ChgD_step6 {459}

$\forall$x,y,f,f$^-$:OV, $\boldsymbol{\alpha}$:OE List, R,P:OE, $\in$:(OV $\longrightarrow$ OTS), b:Qind.
($\forall$ [x; y] |:x = f(y) $\equiv$ y = f$^-$(x)) onlist(OE) $\boldsymbol{\alpha}$ &
($\in$[[x]$\longleftarrow$ ind]) [R; P] :: bool &
($\in$) $\boldsymbol{\alpha}$ :: bool &
$\neg$(some [y] occur free in [R; P]) &
$\neg$x = y & $\neg$y = f &
$\neg$(some [x; y] occur free in $\boldsymbol{\alpha}$) &
$\neg$x = f &
($\in$[[x; y]$\longleftarrow$ ind]) [f; f$^-$] :: ind(1)
$\Rightarrow$

($\in$) $\boldsymbol{\alpha}$ $\vdash$ (*_b [x] | R : (*_b [y] | x = f(y) : P))
=
(*_b [x] | R : (*_b [y] | y = f$^-$(x) : P))

```
                                    THM ChgD_step8 {460}
∀x,y,f,f⁻:OV, α:OE List, R,P:OE, ∈:(OV⟶OTS), b:Qind.
(∀ [x; y] |:x = f(y) ≡ y = f⁻(x)) onlist(OE) α &
(∈[[x]⟵ ind]) [R; P] :: bool &
(∈) α :: bool &
¬(some [y] occur free in [R; P]) &
¬x = y & ¬y = f &
¬(some [x; y] occur free in α) &
¬x = f & ¬y = f⁻ &
(∈[[x; y]⟵ ind]) [f; f⁻] :: ind(1)
⇒

(∈) α ⊢ (*_b [x] | R : (P)[y := f⁻(x)])
         =
         (*_b [x] | R : P)
---

                                    THM ChgD_step7 {461}
∀x,y,f,f⁻:OV, α:OE List, R,P:OE, ∈:(OV⟶OTS), b:Qind.
(∀ [x; y] |:x = f(y) ≡ y = f⁻(x)) onlist(OE) α &
(∈[[x]⟵ ind]) [R; P] :: bool &
(∈) α :: bool &
¬(some [y] occur free in [R; P]) &
¬x = y &
¬(some [x; y] occur free in α) &
¬x = f & ¬y = f⁻ &
(∈[[x; y]⟵ ind]) [f; f⁻] :: ind(1)
⇒

(∈) α ⊢ (*_b [x] | R : (*_b [y] | y = f⁻(x) : P))
         =
         (*_b [x] | R : (P)[y := f⁻(x)])
---
```

THM FullPf_ChgD {462}

```
∀x,y,f,f⁻:OV, α:OE List, R,P:OE, ∈:(OV→OTS), b:Qind.
(∀ [x; y] |:x = f(y) ≡ y = f⁻(x)) onlist(OE) α &
(∈[[x]← ind]) [R; P] :: bool &
(∈) α :: bool &
¬(some [y] occur free in [R; P]) &
¬x = y & ¬y = f &
¬x = f & ¬y = f⁻ &
¬(some [x; y] occur free in α) &
(∈[[x; y]← ind]) [f; f⁻] :: ind(1)
⇒

(∈) α ⊢ (*_b [y] | (R)[x := f(y)] : (P)[x := f(y)])
         =
         (*_b [x] | R : P)
```

```
                  MLDEF solve_not_occ_via_noteqOV {463}
let solve_not_occ_via_noteqOV {463}≡
 WithPf p.
 let occexpn ≡ subterm (concl p) 1 ,
     [varlist;expnlist] ≡ subterms occexpn in
 if opid_of_term expnlist=‘cons‘
 then let [var;nilcomp] ≡ subterms expnlist in
      if nilcomp=nil
      then let occvar ≡ subterm varlist 1 in
           if some (upto 1 (num_hyps p))
              (λi.
               type_of_hyp i p
               =mk_not_term
                (mk_simple_term ‘equal‘
                 [OV;occvar;var]))
           then BackThruLemma
                Thm* ∀v,x:OV.
                     ¬v = x ⇒
                     ¬(some [v] occur free in [x])
                {243}
                THENM
                Complete Hypothesis
           else if some (upto 1 (num_hyps p))
                   (λi.
                    type_of_hyp i p
                    =mk_not_term
                     (mk_simple_term ‘equal‘
                      [OV;var;occvar]))
           then BackThruLemma
                Thm* ∀v,x:OV.
                     ¬x = v ⇒
                     ¬(some [v] occur free in [x])
                {247}
                THENM
                Complete Hypothesis
           else
           FailWith
           ‘solve_NotOcc_via_noteqOV:
            No applicable noteqOV hypothesis‘
      else
      FailWith
      ‘solve_NotOcc_via_noteqOV:
       expnlist not singleton‘
 else
```

```
 FailWith
 `solve_NotOcc_via_noteqOV: explist not a cons list` ;;
```

MLDEF solve_not_occ_concl_v1 {464}

```
let solve_not_occ_concl_v1 {464} ≡
 WithPf p.
 NotOccHD 1
 THENM
 (WithPf q.
  Complete Hypothesis
  ORELSE
  (Not_OccCD
   THENM
   (Complete Hypothesis
    ORELSE
    Onlist_cleanup
    ORELSE
    solve_not_occ_via_noteqOV {463}))) ;;
---
```

MLDEF ProveEqModIffAssoc {465}

```
let ProveEqModIffAssoc {465} ≡
 WithPf p.
 let c ≡ concl p in
 if edit_match ⌈equal_mod_AC_props(<A>;<B>)⌉ c
 then let F X T
          ≡ Assert (mk_member_term ⌈OE⌉ X)
            THENL
            [AddHiddenLabel `wf`;T] in
      F (subterm c 1) (F (subterm c 2) Fiat)
 else
 FailWith
 `ProveEqModIffAssoc: `^`Concl of wrong form. ` ;;
---
```

```
                    MLDEF Othm_oeq_refl_mod_AC_tac {466}
let Othm_oeq_refl_mod_AC_tac {466}≡
 {NOTE: only called when we know thmexpn has opid
  ‘oeq‘ }
 WithPf p.
 let conc ≡ concl p ,
     [thmexpn;otse;assumps] ≡ subterms conc ,
     [leftside;rightside] ≡ subterms thmexpn ,
     eq_type ≡ OEsig_guess_1sig leftside otse (hs p) in
 if termeql_v1 leftside rightside
    & (eq_type=ind or eq_type=bool)
 then BackThruLemma
      Thm* ∀A,B:OE, ∈:(OV→OTS), α:OE List.
           equal_mod_AC_props(A;B) &
           (∈) [A = B] :: bool &
           (∈) α :: bool
           ⇒
           (∈) α ⊢ A = B {403}
      THENM
      (D 0
       THENL
       [ProveEqModIffAssoc {465} ....
       ;D 0
        THEN
        (solve_Owfd_v1
         ORELSE
         (OwfdHD THENM solve_Owfd_v1))])
      THEN
      IfLab ‘wf‘ Id
      (FailWith
       ‘oeq_refl_mod_AC:
        cannot prove type correctness‘)
 else
 FailWith
 ‘Othm_oeq_refl_mod_AC_tac_v1:
  two sides of =_oe not AC termeql or
  unable to guess type signature‘ ;;
---
```

```
                              MLDEF leibrewrite_aux {467}
let leibrewrite_aux {467} newvarname i ≡
 WithPf p.
 (Rewrite
  (SweepDnC
   (IfC
    (λe.λt.
     opid_of_term t=‘osubst_1var‘
     & subterm t 2=newvarname)
    (RevLemmaC
     Thm* ∀P:OE, V:OV, E,Q:OE.
          ((P)[V := E] ⇒ (Q)[V := E])
          =
          (P ⇒ Q)[V := E] {405})))
  i
  ORELSE
  Rewrite
  (SweepDnC
   (IfC
    (λe.λt.
     opid_of_term t=‘osubst_1var‘
     & subterm t 2=newvarname)
    (RevLemmaC
     Thm* ∀P:OE, V:OV, E,Q:OE.
          (P)[V := E] ∧ (Q)[V := E] = (P ∧ Q)[V := E]
     {412})))
  i
  ORELSE
  Rewrite
  (SweepDnC
   (IfC
    (λe.λt.
     opid_of_term t=‘osubst_1var‘
     & subterm t 2=newvarname)
    (RevLemmaC
     Thm* ∀P:OE, V:OV, E,Q:OE.
          (P)[V := E] ∨ (Q)[V := E] = (P ∨ Q)[V := E]
     {411})))
  i
  ORELSE
  Rewrite
  (SweepDnC
   (IfC
    (λe.λt.
```

```
   opid_of_term t=‘osubst_1var‘
   & subterm t 2=newvarname)
  (RevLemmaC
   Thm* ∀P:OE, V:OV, E,Q:OE.
        (P)[V := E] ≡ (Q)[V := E] = (P ≡ Q)[V := E]
   {410})))
i
ORELSE
Rewrite
(SweepDnC
 (IfC
  (λe.λt.
   opid_of_term t=‘osubst_1var‘
   & subterm t 2=newvarname)
  (RevLemmaC
   Thm* ∀P:OE, V:OV, E:OE.
        (¬(P)[V := E]) = (¬P)[V := E] {409})))
i
ORELSE
Rewrite
(SweepDnC
 (IfC
  (λe.λt.
   opid_of_term t=‘osubst_1var‘
   & subterm t 2=newvarname)
  (RevLemmaC
   Thm* ∀P:OE, V:OV, E,Q:OE.
        (P)[V := E] = (Q)[V := E] = (P = Q)[V := E]
   {408})))
i
ORELSE
Rewrite
(SweepDnC
 (IfC
  (λe.λt.
   opid_of_term t=‘osubst_1var‘
   & subterm t 2=newvarname)
  (LemmaC
   Thm* ∀f,x:OE, V:OV, E:OE.
        (f(x))[V := E] = (f)[V := E]((x)[V := E])
   {407})))
i
ORELSE
Rewrite
```

```
(SweepDnC
 (IfC
  (λe.λt.
   opid_of_term t=‘osubst_1var‘
   & subterm t 2=newvarname)
  (LemmaC Thm* ∀x:OV, E:OE. (x)[x := E] = E {368})))
i
ORELSE
Rewrite
(SweepDnC
 (IfC
  (λe.λt.

   opid_of_term t=‘osubst_1var‘
   & subterm t 2=newvarname)
  (RevLemmaC
   Thm* ∀b:Qind, P:OE, V:OV, E,Q:OE.
         (P)[V := E] *_b (Q)[V := E]
         =
         (P *_b Q)[V := E] {413})))
i
ORELSE
Rewrite
(SweepDnC
 (IfC
  (λe.λt.
   opid_of_term t=‘osubst_1var‘
   & subterm t 2=newvarname)
  (RevLemmaC
   Thm* ∀V:OV, E:OE. f = (f)[V := E] {416})))
i
ORELSE
Rewrite
(SweepDnC
 (IfC
  (λe.λt.
   opid_of_term t=‘osubst_1var‘
   & subterm t 2=newvarname)
  (RevLemmaC
   Thm* ∀V:OV, E:OE. t = (t)[V := E] {415})))
i
ORELSE
Rewrite
(SweepDnC
```

```
  (IfC
   (λe.λt.
    opid_of_term t=`osubst_1var`
    & subterm t 2=newvarname)
   (RevLemmaC
    Thm* ∀b:Qind, V:OV, E:OE. u_b = (u_b)[V := E]
    {414})))
 i
 ORELSE
 (Rewrite
  (SweepDnC
   (IfC
    (λe.λt.
     opid_of_term t=`osubst_1var`
     & subterm t 2=newvarname
     & member (opid_of_term (subterm t 1))
       [`variable`;`ovar_lit`])
    (LemmaC
     Thm* ∀X:OV, P,E:OE.
           ¬(some [X] occur free in [P]) ⇒
           (P)[X := E] = P {371})))
  i
  THEN
  IfLab `rewrite subgoal`
  (Try
   (Complete Hypothesis
    ORELSE
    (solve_newvar_notocc_v2
     THEN
     IfLab `wf` Id
     (FailWith
      `leibrewrite:
       solve_newvar didnt complete goal`))))
  Id)
 ORELSE
 FailWith `leibrewrite:No lemma to apply`)
THEN
IfLab `antecedent` Trivial Id ;;
```

```
                                MLDEF leibrewrite {468}
letrec leibrewrite {468} newvarname i≡
 WithPf p.
 leibrewrite_aux {467} newvarname i
 THENM
 Try leibrewrite {468} newvarname i ;;
```

```
                           MLDEF lame_LeibQBap_v1 {469}
let lame_LeibQBap_v1 {469}≡
 WithPf q.
 setupnewvar (listforsetupnewvar (subterm (concl q) 1))
 THENM
 (WithPf p.
  let conc ≡ concl p ,
      [thmexpn;otse;assumps] ≡ subterms conc ,
      [leftside;rightside] ≡ subterms thmexpn ,
      contguess,[lguess;rguess],varname
      ≡ find_context_match_v1 leftside rightside
        (mk_variable_term (getnewvarname p)) ,
      [qind;vars;range;realcontguess]
      ≡ subterms contguess in
  Leib_quant_body_apply_v1 realcontguess varname lguess
  rguess) ;;
```

```
let Leib_tac_v1 {470}≡          MLDEF Leib_tac_v1 {470}
 WithPf p.
 let conc ≡ concl p in
 if opid_of_term conc=`Othm`
 then let [thmexpn;otse;assumps] ≡ subterms conc in
      if opid_of_term thmexpn=`oeq`
      then let [leftside;rightside]
               ≡ subterms thmexpn ,
               lopid ≡ opid_of_term leftside ,
               ropid ≡ opid_of_term rightside in
           if lopid=`qstar` & ropid=`qstar`
           then let [lqind;lquantvarlist;lrange;lbody]
                    ≡ subterms leftside ,
                    [rqind;rquantvarlist;rrange;rbody]
                    ≡ subterms rightside in
                if lqind=rqind
                   & lquantvarlist=rquantvarlist
                then if {this case is now covered in
                         lame_Leib_tac_ap_v1; do not
                         take it out of here,
                         though--may be useful in
                         recursive case }
                        termeql_v1 lrange rrange
                        & termeql_v1 lbody rbody
                     then Othm_oeq_refl_mod_AC_tac
                          {466}
                     else if termeql_v1 lrange rrange
                     then lame_LeibQBap_v1 {469}
                     else if termeql_v1 lbody rbody
                     then lame_LeibQRap_v1
                     else lame_Leibsimpap_v1
                else lame_Leibsimpap_v1
           else lame_Leibsimpap_v1
      else
      FailWith `Leib_tac_v1: concl not form A=_oeB`
 else FailWith `Leib_tac_v1: concl not Othm expn` ;;
```

A.1 Index of Nuprl Highlights

ABS

OEcase {133}	Case of term:OE
	Ovar(x) $\longrightarrow$ varfn(x)
	f $\longrightarrow$ falsecase
	u(v) $\longrightarrow$ apfn(u;v)
	u $\Rightarrow$ v $\longrightarrow$ impfn(u;v)
	u=v $\longrightarrow$ eqlfn(u;v)
	($\forall$ x $\vert$:v) $\longrightarrow$ allfn(x;u)
alleltsdiff_gen {3}	Distinct_elts(l,T)
env_type {173}	env(V)
eqovsep {122}	eq_OV
assoc_mod_iff {63}	equal_mod_AC_props(A;B)
Es_notoccurs {150}	Es_not_occurs(v,Es)
Esoccurs {158}	Esoccurs(v,Es)
ofalse {81}	f
findfirst {193}	findfirst(L; x.P(x))
fn_of_OE {129}	fn_of_OE(obexpn)
hnum {142}	hnum(es)
hnum_aux {140}	hnum_aux(e)
ident {44}	IDENT
ident2 {53}	ident2(identifier)
IDENTnums_gtr_n {177}	IDENTnums_gtr_n(L,n)
identofovar {110}	identofovar(ov)
indsig {22}	ind(n)
oindtype {10}	IND_O
IndBool {28}	{ind, bool}
is_a_ovar {69}	is_a_ovar(ov)
is_bvar_in_OE {136}	is_bvar_in_OE(id,exp)
IsOvar {105}	IsOvar(Y)
lookup_env {174}	lookup_env(ident,env)
max_ident_num {57}	max_ident_num(L)
mk_nonOV {168}	mk_nonOV(opid,fn_of_oe)
no_2ofs_equal {45}	no_2ofs_equal(L)
non_ov {126}	NonOV
NotOcc_Esfirst {156}	(no vs occur free in Es$^{\lceil + \rceil}$)
NotOcc_vsfirst {152}	(no vs$^{\lceil + \rceil}$ occur free in Es)
notonlist_gen {1}	Not_onlist(elt,l:T)
nthsbtm {131}	nthsbtm(oe,posn)
Ofree {146}	(v occurs free in e)
obexpn {62}	OE
oebv {30}	oebv(opid; posn)
oesbtms {42}	oesbtms(opid)
Ogdtm {221}	Ogdtm(Es; env)
Ogdtmelt {213}	Ogdtmelt(exp,env)
OGT {9}	OGT

ol_all {118}	ol_all(xs; r; p)
ol_exists {116}	ol_exists(xs; r; p)
opid_all {34}	opid_all
opid_ap {40}	opid_ap
opid_eq {36}	opid_eq
opid_false {32}	opid_f
opid_imp {38}	opid_imp
opid_of_OE {127}	opid_of_OE(obexpn)
OPIDS {29}	OPIDS
osubstn_ready {138}	Osubstn_ready(E,n)
Othmaux_v2 {229}	Othmaux_2(exp,env,assumps,n)
obtypsigs {19}	OTS
ots_form {20}	ots(gdtype,degree)
obvar {107}	OV
ovar {101}	Ovar(X)
PossBV {49}	PossBV(opid;posn)
quantind {14}	Qind
qind_all {17}	Qind∀
qind_exists {15}	Qind∃
renamebdvars {185}	rename(e,num,rewrites)
env_for_rename_lemma {176}	rename_lem_env(env,L)
renamed_var {51}	ren'dvar(x,num)
Occurs {164}	(some vs occur free in Es)
Occurs_vsfirst {160}	(some vs⌈+⌉ occur free in Es)
suchthat_qind {97}	s.t.(b,r,p)
otrue {83}	t
thmclause_all_elim {223}	thm_all_elim(exp,e,x,f,env, reccall)
thmclause_all_elim2 {148}	thm_all_elim2(exp,e,x, reccall)
thmclause_all_intro {166}	thm_all_intro(exp,x,e, assumps, reccall)
thmclause_arrow_intro {79}	thm_arrow_intro(exp,e,f, reccall)
thmclause_cbv {197}	thm_cbv(exp,f,rec_call)
thmclause_eq_elim {219}	thm_eq_elim(exp,e,x,a,b,env, reccall1, reccall2)
thmclause_eq_intro {217}	thm_eq_intro(exp,a,env, assumps)
thmclause_equiv_as_eq {75}	thm_equiv_as_eq(exp,a,b, reccall)

Bibliography

[AA99] E. Aaron and S. F. Allen. Justifying calculational logic by a conventional metalinguistic semantics. Technical Report TR99-1771, Department of Computer Science, Cornell University, 1999.

[Aar] E. Aaron. A Nuprl implementation of calculational logic inference. To appear. Available via Nuprl webpage http://cs.cornell.edu/Info/Projects/NuPrl.

[ACHA90] S. F. Allen, R. L. Constable, D. J. Howe, and William E. Aitken. The semantics of reflected proof. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 95–105, Philadelphia, Pennsylvania, 4–7 June 1990. IEEE Computer Society Press.

[All87] S. F. Allen. A Non-type-theoretic Definition of Martin-Löf's Types. In *Proceedings of the Second Symposium on Logic in Computer Science*, pages 215–224. IEEE, June 1987.

[All98] S. F. Allen. From dy/dx to []P: a matter of notation. In *Proceedings of the 4th Workshop on User Interfaces for Theorem Provers, UITP-98*, Eindhoven, Netherlands, 1998. Lawrence Erlbaum Associates.

[And93] J. R. Anderson. *Rules of the Mind*. Lawrence Erlbaum Associates, Hillsdale, NJ, 1993.

[AS98] E. Aaron and M. Spivey. Designing a calculational logic theorem prover: Insight into search procedure via eye movements. Technical Report TR98-1680, Department of Computer Science, Cornell University, 1998.

[Bac95] R. Backhouse, editor. *Information Processing Letters, Special issue on The Calculational Method*, volume 53. February 1995.

[BHP95] D. Ballard, M. Hayhoe, and J. Pelz. Memory representations in natural tasks. *Journal of Cognitive Neuroscience*, 7:68–82, 1995.

[C$^+$86] R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1986.

[CKB84] R. L. Constable, T. Knoblock, and J.L. Bates. Writing programs that construct proofs. *J. Automated Reasoning*, 1(3):285–326, 1984.

[DS90] E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, New York, 1990.

[ES96] J. Epelboim and P. Suppes. Window on the mind? What eye movements reveal about geometric reasoning. In *Proceedings of the 18th Annual Conference of the Cognitive Science Society*, Mahwah, NJ, 1996. Lawrence Erlbaum Associates.

[GM93] M. J. C. Gordon and T. F. Melham. *Introduction to HOL*. Cambridge University Press, Cambridge, 1993.

[GMW79] M. Gordon, A. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1979.

[Gri] D. Gries. Personal correspondence regarding capture-permitting substitution.

[GS93a] D. Gries and F. B. Schneider. Instructor's manual for "A Logical Approach to Discrete Math", 1993.

[GS93b] D. Gries and F. B. Schneider. *A Logical Approach to Discrete Math*. Springer-Verlag, New York, 1993.

[GS94] D. Gries and F. B. Schneider. A new approach to teaching mathematics. Technical Report TR94-1411, Department of Computer Science, Cornell University, 1994.

[HC68] G.E. Hughes and M.J. Cresswell. *An Introduction to Modal Logic*. Mehuen and Co., New York, 1968.

[HMG92] M. Hegarty, R. Mayer, and C. Green. Comprehension of arithmetic word problems: Evidence from students' eye fixations. *Journal of Educational Psychology*, 84:76–84, 1992.

[How96] D. J. Howe. Semantic foundations for embedding HOL in Nuprl. In *Algebraic Methodology and Software Technology: 5th international conference*, New York, 1996. Springer-Verlag.

[Jac94] P. Jackson. The Nuprl proof development system reference manual and user's guide, 1994. http://cs.cornell.edu/Info/Projects/NuPrl/manual.with.index/it.html (May, 2000).

[JL83] P. N. Johnson-Laird. *Mental Models: Towards a Cognitive Science of Language, Inference, and Consciousness*. Harvard University Press, Cambridge, MA, 1983.

[Kli94] W. Klimesch. *The Structure of Long-term Memory: A Connectivity Model of Semantic Processing*. Lawrence Erlbaum Associates, Hillsdale, NJ, 1994.

[M$^+$] S.P. Marshall et al. Decision-making schemas in rapidly changing situations. http://www.sci.sdsu.edu/cerf/content/research.html (May, 2000).

[MA94] C. Mannion and S. F. Allen. A notation for computer aided mathematics. Technical report, Cornell University, Ithaca, NY, 1994.

[McA91] D. McAllester. Observations on cognitive judgements. In *AAAI-91*, pages 910–915, 1991.

[Mel94] E. Melis. How mathematicians prove theorems. In *Proceedings of the 16th Annual Conference of the Cognitive Science Society*, Hillsdale, NJ, 1994. Lawrence Erlbaum Associates.

[Min61] M. Minsky. Steps toward artificial intelligence. *Proceedings of the Institute of Radio Engineers*, 49:8–30, 1961.

[ML82] P. Martin-Löf. Constructive mathematics and computer programming. In *Proceedings of the Sixth International Congress of Logic, Methodology, and Philosophy of Science*, New York, 1982. North-Holland Publishing Co.

[MLm93] Nuprl ML manual, 1993. No author attributed.

[New90] A. Newell. *Unified Theories of Cognition*. Harvard University Press, Cambridge, MA, 1990.

[NS61] A. Newell and H. Simon. Gps, a program that simulates human thought. In *Lernende Automaten*. R. Oldenbourg KG, Munich, 1961.

[NSS57] A. Newell, J. C. Shaw, and H. Simon. Empirical explorations with the logic theory machine. In *Proceedings of the Western Joint Computer Conference*, volume 15, pages 218–239, 1957.

[Rip94] L. Rips. *The Psychology of Proof: Deductive Reasoning in Human Thinking*. MIT Press, Cambridge, MA, 1994.

[RM+86] D. E. Rumelhart, J. L. McClelland, et al. *Parallel Distributed Processing: Explorations in the Microstructures of Cognition*, volume 1. MIT Press, Cambridge, MA, 1986.

[SKM99] J.H. Siekmann, M. Kohlhase, and E. Melis. Omega — A mathematical assistant system. In *Essays dedicated to Johan van Benthem on the occasion of his 50th Birthday*. Amsterdam University Press, 1999. Published on CD-ROM.

[TSKES95] M. Tanenhaus, M. Spivey-Knowlton, K. Eberhard, and J Sedivy. Integration of visual and linguistic information during spoken language comprehension. *Science*, 268:1632–1634, 1995.

[WJL72] P. C. Wason and P. N. Johnson-Laird. *Psychology of Reasoning: Structure and Content*. Harvard University Press, Cambridge, MA, 1972.