Justifying Calculational Logic by a Conventional Metalinguistic Semantics

Eric Aaron[1] and Stuart Allen[2]
Department of Computer Science
Cornell University
Ithaca, NY 14853
aaron@cs.cornell.edu, sfa@cs.cornell.edu

September 1999

Abstract

We provide a metalinguistic formalization of calculational logic, an alternative to higher-order logic for escaping the restrictions of first-order logic. We show that conventional semantic techniques can provide an adequate foundation for calculational logic, even its atypical metalinguistic features.

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 first-order semantics does not accommodate substitution. In this paper, 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 [1]. 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

[1] http://www.cs.cornell.edu/home/aaron. Supported by NSF grant GER-9454149.
[2] http://www.cs.cornell.edu/home/sfa. Supported by DARPA grant EIA-9812630.

theorem, as in [3]. For example, the following metatheorem,

(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 [1] —see also the IPL issue [4], 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, (1) is treated as an axiom:

(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 [1] 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 paper 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:

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

which in the calculational notation of [1] 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 [1], it is given explicitly and concisely as:

(3) **Change of dummy:** Provided $\neg occurs(\text{“}j\text{”}, \text{“}R, P\text{”})$ 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 [1], the theorem is stated in terms of a general operator $*$ instead of the addition operator $+$; the theorem holds for various binary operators $*$ 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 $*$ to $\vee$ and $\wedge$ for the purposes of this paper.)

This theorem and its proof (Fig. 1, taken verbatim from [1]) 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 [1], 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 [1].

2 An Example Metalinguistic Explanation

The proof of Change of Dummy shown in Fig. 1 is a chain of seven equalities, where each equality is justified using inference rule "substitution of equals for equals" (called *Leibniz* in [1]). 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 gives the premise $X = Y$ (within braces "$\langle$" and "$\rangle$"). In this case, the premise is the instance $(*x \mid x = f.y : P) = P[x := f.y]$ of axiom One-point rule.

The transitivity of equality-derivability —known simply as "Transitivity" in [1]— 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 paper.) 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-order language by incorporating

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

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

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

Figure 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 present in section 7.

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 [1]. 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 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 [1]. 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 [1], 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 where 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 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 [1]. 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 $\}$.

3.1 Object language syntax

(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 3.2 on object language semantics.

(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 [1], equality is applied to both individual and propositional expressions.

(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 [1] 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 [1], 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.

3.2 Object language semantics

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

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

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

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

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

(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^x_d is the same as V except it assigns d to variable x. We also follow this convention for typing environments and other objects throughout the paper.

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

(9) **Definition.** For a domain of individuals D, typing environment $\sigma{:}(OV \rightarrow K(D))$, 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 $Osem(P, F, D \rightarrow T)$ 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 $\forall d{:}D.\ Osem(\sigma^x_D, V^x_d, P, g(d), \mathbb{B})$.
6. $\forall T{:}\{D, \mathbb{B}\}, p{:}T, q{:}T.\ Osem(P =_O Q, p = q, \mathbb{B})$ 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 [1], 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.

(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 common values, such as $\mathbb{N}$ and $\mathbb{Z}$. All of these are reasonable extensions if we were to include more of the discrete mathematics of [1] in our metalinguistic framework.

(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 $Osem(\sigma, V, E, d, T')$.

(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.

(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.

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.

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 $K(D)$:

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

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 $OV \to OT$. 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 further 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. $(\forall_O x \mid : A) :: bool$ 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 $\forall A \in \alpha.\ (\epsilon)\ A :: T$.

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)

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

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 motivation for choosing a particular definition of the object language and our motivation for choosing a particular definition of object-level derivability. Recall that, in many ways, the details of our particular choice of object language is 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 [1] 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.

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

1. $\alpha \vdash P$ if $P \in \alpha$ and $\alpha :: bool$
2. $\alpha \cup \{A\} \vdash P$ if $\alpha \vdash P$ and $A :: bool$
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 $A :: ind$
7. $\alpha \vdash P$ if $\alpha \vdash (\forall_O x.P)$ and $\neg(x$ free in $P)$
8. $\alpha \vdash (\forall_O x.P)$ if $(\epsilon^x_{ind})\ \alpha \vdash P$ and $\neg(x$ free in $\alpha)$
9. $\alpha \vdash P$ if $\alpha \vdash \mathbf{f}_O$ and $P :: bool$

10. $\forall r{:}\{bool, ind\}.\ \alpha\ \vdash P[x := B]$ if $\alpha\ \vdash P[x := A]$ and $\alpha\ \vdash A =_O B$ and $A, B :: r$ and $(\epsilon^x_r)\ P :: bool$

11. $\forall r{:}\{bool, ind\}.\ \alpha\ \vdash A =_O A$ if $A :: r$ and $\alpha :: bool$

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 $\epsilon{:}OV \rightarrow OT$ 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 $\mathbf{t}_O$).

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 [1] 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:

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

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.

5 Quantification and Iteration

One of the advances in [1] 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 [1], 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, $(\mathtt{M}\ xs \mid R : B)$, 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 $(\mathtt{M}\ v \mid R : B)$ is the iteration of a binary operator $x\ \mathtt{M}\ y$ 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 [1], $(\exists_O v \mid R.v : f.v) = f(i_0) \vee_O f(i_1)$, and similarly for $\forall_O$.

Part of extending the formalization beyond predicate logic would be to give a semantics to $(\mathtt{M}\ xs \mid R : B)$ 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 [1] states that a quantifier expression $(\mathtt{M}\ xs \mid R : B)$ with an empty range gets the value $\mathtt{id}(\mathtt{M})$, 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 $\mathtt{M} = \wedge^M$, $P\ \mathtt{M}\ Q$ is $P \wedge_O Q$ and $\mathtt{id}(\mathtt{M})$ is $\mathbf{t}_O$. Similarly, for $\mathtt{M} = \vee^M$, $P\ \mathtt{M}\ Q$ is $P \vee_O Q$ and $\mathtt{id}(\mathtt{M})$ is $\mathbf{f}_O$. Using this informal metalanguage, we may state uniformities such as:

$$((\mathtt{M}\ xs \mid R : P)\ \mathtt{M}\ (\mathtt{M}\ xs \mid R : Q)) =_O (\mathtt{M}\ xs \mid R : P\ \mathtt{M}\ Q).$$

The calculational style of [1] is designed to exploit such properties.

6 The Data Language

We use the term *data language* for the expressions of [1] 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 [1] (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 [1] 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 [1], which influenced some of our design decisions.

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.

(15) **Definition.** *DT* is a class of type expressions used in data language quantifiers. It comprises the constants *OV*, *OE*, *OT*, $\mathbb{N}$, { bool , 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.

(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 $a, b, c, *, \epsilon, \alpha \in DE$, $x \in DV$, and $t \in DT$):

- Various standard logical operations (including identity): *if a then b*; *a & b*; *a or b*; *not a*; *a is b*; *for x : t. a*; *a 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: $a(b); \mathbf{t}; \mathbf{f}$; $a \Rightarrow b; a \Leftarrow b; a = b; \neg a; a \equiv b; a \wedge b; a \vee b; (*a \mid b : c); a * b; \mathtt{id}(*); a[b := c]$.
- 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: 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 .

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.

(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 15. We do not expect it to cause confusion.

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

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

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

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

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.

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

- $\gamma : DV \to \texttt{Type}$ 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 $\gamma(x)$.

(20) **Definition.** The data language semantics is given as a recursive definition of $\mathit{Dsem}(\gamma, V, A, v, T)$ for $\gamma{\in}DV \to \texttt{Type}, V{\in}(\Pi x{:}DV.\ \gamma(x)), A{\in}DE, T{\in}\texttt{Type}, v{\in}T$. We intend $\mathit{Dsem}(\gamma, V, A, v, T)$ 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. $\mathit{Dsem}(A, a, [t2])$ if $\mathit{Dsem}(A, a, [t1])$ and $t1 \subseteq t2$.
2. $\forall x{:}DV.\ \mathit{Dsem}(x, V(x), \gamma(x))$.
3. $\mathit{Dsem}(\mathit{if}\ P\ \mathit{then}\ Q, q \text{ if } p, \mathbb{B})$ if $\mathit{Dsem}(P, p, \mathbb{B})$ and $\mathit{Dsem}(Q, q, \mathbb{B})$,
   and similarly for the other connectives.
4. $\mathit{Dsem}(A\ \mathit{is}\ B, a = b, \mathbb{B})$ if $\mathit{Dsem}(A, a, T)$ and $\mathit{Dsem}(B, b, T)$.
5. $\forall g{:}[t] \to \mathbb{B}.\ \mathit{Dsem}((\mathit{for}\ x : t.\ a), (\forall v{:}[t].g(v)), \mathbb{B})$ if $\forall v{:}[t].\mathit{Dsem}(\gamma^x_{[t]}, V^x_v, a, g(v), \mathbb{B})$.
6. $\forall T{:}\texttt{Type}.\ \mathit{Dsem}([\,], \mathrm{nil}, T \text{ List})$; note the type polymorphism.
7. $\mathit{Dsem}(A \cdot B, \text{the concatenation of } a \text{ to } b, T \text{ List})$
   if $\mathit{Dsem}(A, a, T)$ and $\mathit{Dsem}(B, b, T \text{ List})$.
8. $\mathit{Dsem}(\langle A, B \rangle, \langle a, b \rangle, T1 \times T2)$ if $\mathit{Dsem}(A, a, T1)$ and $\mathit{Dsem}(B, b, T2)$.
9. $\mathit{Dsem}(abc, abc_O, OV)$; similarly for other object variable literals.
10. $\mathit{Dsem}(\mathbf{t}, \mathbf{t}_O, OE)$ and $\mathit{Dsem}(\mathbf{f}, \mathbf{f}_O, OE)$.
11. $\mathit{Dsem}(A \equiv B, a \equiv_O b, OE)$ if $\mathit{Dsem}(A, a, OE)$ and $\mathit{Dsem}(B, b, OE)$;
    similarly for other OE-constructors.

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

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.

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

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

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

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

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

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

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

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 [1] 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 $\mathit{if}\ P\ \mathit{then}\ Q$; 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 $P \mathbin{\&} Q$, etc.

Our syntax and semantics omitted a few expressions that appear in [1] that are easily definable in terms of the language we presented. For instance, $(X{:}OV\ \mathit{free\ in}\ A{:}OE\ \mathrm{List})$ is extended to the data-level predicate $\mathit{occurs}(V{:}OV\ \mathrm{List}, E{:}OE\ \mathrm{List})$, which holds when some V' in V is such that $(V\ \mathit{free\ in}\ E)$. It is typically negated, as $\mathit{not\ occurs}(V, E)$ to indicate 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 $X{:}OV\ \mathrm{List}$ and $r{:}OT$, ϵ_r^X denotes a copy of $\epsilon{:}(OV \times OT)\ \mathrm{List}$ 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 7.

Relatedly, we have not been explicit about certain conventions in [1] 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 only used 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 [1]. 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 only occur for easily understood standard notation, and they should not pose difficulties for readers. Unless otherwise explained, standard notations have their expected meanings.

7 Using the Data Language

Our goal in this paper is to demonstrate how calculational predicate logic (as presented in chapters 3, 8, and 9 of [1]) could be easily read as a restricted, formalized metalanguage. We have formalized this data language, and we have also already illustrated the use of the data language with our treatment of the proof of Theorem Change of Dummy. We do not attempt to formalize the actual proof structure used in [1]; 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 and read using our data language; the names and numbers labeling our examples are taken directly from the text [1]. 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 [1]. 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.

7.1 Theorems

Here we present data language expressions for a few objects classified as theorems in [1]. In that text, 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 [1]) 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 (Fig. 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, $*{:}\{\wedge, \vee\}, x{:}OV, E, P{:}OE$.

if not occurs(x, E) & (ϵ) $\alpha ::$ bool & (ϵ) $E ::$ ind & $(\epsilon^{x}_{\text{ind}})$ $P ::$ bool
then (ϵ) $\alpha \vdash (*x \mid x = E : P) = P[x := E]$.

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

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

7.2 Inference rules

The statements identified in [1] 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, $*{:}\{\wedge, \vee\}$,
  $Z{:}OV, X{:}OV$ List, $A, B, P, R{:}OE, r{:}\{$ bool , ind $\}$.
  if not occurs(X, α) &
  $(\epsilon^{X}_{\text{ind}})$ $\alpha \vdash (R \Rightarrow A = B)$ &
  $(\epsilon^{X}_{\text{ind}})$ $A, B :: r$ &
  $(\epsilon^{X,Z}_{\text{ind},r})$ $P ::$ bool
  then (ϵ) $\alpha \vdash (*X \mid R : P[Z := A]) = (*X \mid R : P[Z := B])$

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

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

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 [1]. Here, as a demonstration of adequacy, we present the central component propositions of the proof of Theorem Change of Dummy (see Fig. 1). Other proofs in [1] 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 [1] also used in Fig. 1. Note that the similarities would be even greater if we systematically abbreviated $(\epsilon)\ \alpha \vdash P$ to simply P.

This subsection also provides the most direct support for our claim that, although we do not formalize a proof structure for calculational logic in this paper, 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 paper.

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

1. *For* $\epsilon{:}(OV \times OT)$ List$, \alpha{:}OE$ List$, *{:}\{\wedge, \vee\}, P, R{:}OE, f, f^{-1}, x, y{:}OV.$
   assuming $(\in \alpha)\ (*x, y \mid \mathbf{t} : x = f.y \equiv y = f^{-1}.x)$,
   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 $(\epsilon)\ \alpha \vdash (*y \mid R[x := f.y] : P[x := f.y]) = (*y \mid R[x := f.y] : (*x \mid x = f.y : P))$
2. $(\epsilon)\ \alpha \vdash (*y \mid R[x := f.y] : (*x \mid x = f.y : P)) = (*x, y \mid R[x := f.y] \wedge x = f.y : P)$
3. $(\epsilon)\ \alpha \vdash (*x, y \mid R[x := f.y] \wedge x = f.y : P) = (*x, y \mid R[x := x] \wedge x = f.y : P)$
4. $(\epsilon)\ \alpha \vdash (*x, y \mid R[x := x] \wedge x = f.y : P) = (*x \mid R : (*y \mid x = f.y : P))$
5. $(\epsilon)\ \alpha \vdash (*x \mid R : (*y \mid x = f.y : P)) = (*x \mid R : (*y \mid y = f^{-1}.x : P))$
6. $(\epsilon)\ \alpha \vdash (*x \mid R : (*y \mid y = f^{-1}.x : P)) = (*x \mid R : P[y := f^{-1}.x])$
7. $(\epsilon)\ \alpha \vdash (*x \mid R : P[y := f^{-1}.x]) = (*x \mid R : P)$

8 Conclusion and Discussion

By formalizing a metalinguistic interpretation of the language of the calculational logic of [1], we believe 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 paper 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 7), but it is not necessary to be thoroughly familiar with that section.

8.1 Justifying calculational logic

Using the examples from section 7 as support, we feel the technical content of this paper is an adequate basis for justifying calculational logic. We explained the language of calculational logic using only standard, well-understood semantic methods. The semantics we gave resulted in a sensible reading of the predicate logic in [1], 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 [1] 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.

8.2 Calculational logic inferences

The inferences of calculational logic may seem odd based on the typical calculational proof structure presented in [1]. 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 propositions P and Q really are.

It would be both tiresome and outside the scope of the paper 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 be readily grasped, expressed, and applied using standard methods, however, given our clarification of the propositions underlying calculational logic.

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.

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 [1] is readily seen to be non-standard in the following ways:

1. Identifiers don't 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 which 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 paper 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 [1]. 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 the formulas designated as theorems in [1], it is explicitly used in statements designated as metatheorems. Readers of [1] 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 [1], 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.

Acknowledgements

We thank David Gries for his advice and criticism.

References

[1] Gries, D., and F.B. Schneider. *A Logical Approach to Discrete Math*. Springer-Verlag, New York, 1993.

[2] Gries, D., and F.B. Schneider. *A Logical Approach to Logic*. Springer-Verlag. To appear.

[3] Hughes, G.E., and M.J. Cresswell. *An Introduction to Modal Logic.* Mehuen and Co., New York, 1968.

[4] Backhouse, R. (editor). *Information Processing Letters 53*, 3 (February 1995).