

A Logical View of Assignments

Vipin Swarup*

The MITRE Corporation
Burlington Road
Bedford, MA 01730.
E-mail: swarup@mitre.org

Uday S. Reddy†

Dept. of Computer Science
University of Illinois
at Urbana-Champaign
Urbana, IL 61801.
E-mail: reddy@cs.uiuc.edu

August 21, 1991

Abstract

Imperative lambda calculus (ILC) is an abstract formal language obtained by extending the typed lambda calculus with imperative programming features, namely references and assignments. The language shares with typed lambda calculus important properties such as the Church-Rosser property and strong normalization. In this paper, we describe the logical symmetries that underly ILC by exhibiting a constructive logic for which ILC forms the language of constructions. Central to this formulation is the view that references play a role similar to that of variables. References can be used to range over values and instantiated to specific values. Thus, we obtain a new form of universal quantification that uses references instead of variables. The essential term forms of ILC are then obtained as the constructions for the introduction and elimination of this quantifier. While references duplicate the role of variables, they also have important differences. References are *semantic* values whereas variables are syntactic entities and, secondly, references are *reusable*. These differences allow references to be used in a more flexible fashion leading to efficiency in constructions and algorithms.

1 Introduction

Reasoning about imperative programs has proven hard to formalize. While Hoare Logic [Hoa69] made an impressive beginning, it proved unsuitable for treating higher-order functions, pointer structures and a variety of other popular features. Reynolds extended Hoare Logic in a fundamental way in his formulation of Specification Logic [Rey82, Ten89], but the complexity

*Supported by NASA grant NAG-1-613 (while at the University of Illinois at Urbana-Champaign).

†Supported by a grant from Motorola Corporation.

of this system is deterring. Further, Specification Logic is unable to handle pointer structures. Other approaches, such as dynamic logic [Pra76] and algorithmic logics [Eng75, MS87], are close to Hoare logic and suffer from the same limitations.

In our recent work [SRI91], we attempted a new approach based on the insights obtained from applicative programming over the last two decades. Our formulation, called *Imperative Lambda Calculus* (ILC), extends typed lambda calculus with references and assignments, and uses a layered type system to prohibit all forms of side effects. We were able to show that, in spite of the radical extensions, the language retains the salient properties of typed applicative languages such as the Church-Rosser property and strong normalization.

In this paper, we report on the fundamental logical symmetries brought forth by this formulation of assignments. References play a role similar to that of variables in logic. They can be used to range over sets of values and instantiated to specific values. Thus, we obtain a new form of universal quantification that uses references instead of variables. The essential operators of imperative lambda calculus can be treated as the proof terms for the introduction and elimination rules of this quantifier. Thus, imperative lambda calculus can be regarded as the language of constructions for a suitably formulated constructive logic.

After a brief overview of imperative lambda calculus, we discuss the basic concepts involved in such a formulation in Section 3. In Section 4, we present a formal system called *observation type theory* which incorporates these concepts.

2 Overview of Imperative Lambda Calculus

Imperative lambda calculus (ILC) [SRI91] extends the simply typed lambda calculus with first-class references and assignments. The fundamental principle underlying ILC is that the only manipulation needed for states is *observation* (i.e. inspection). Consider the fact that in typed lambda calculus, environments are implicitly extended and observed (via the use of variables), but are never explicitly manipulated. Similarly, in ILC, states are implicitly extended and observed (via the use of references), but are never explicitly manipulated. Thus, in a sense, *the world exists only to be observed*.

The types of ILC are stratified into three layers as follows:

$$\begin{array}{ll}
 \text{(Applicative types)} & \tau ::= \beta \mid \tau_1 \times \tau_2 \mid \tau_1 \rightarrow \tau_2 \\
 \text{(Storage types)} & \theta ::= \tau \mid \mathbf{Ref} \theta \mid \theta_1 \times \theta_2 \mid \theta_1 \rightarrow \theta_2 \\
 \text{(Imperative types)} & \omega ::= \theta \mid \mathbf{Obs} \tau \mid \omega_1 \times \omega_2 \mid \omega_1 \rightarrow \omega_2
 \end{array}$$

All three layers are closed under product and function space constructions. *Applicative* types are those of typed lambda calculus. *Storage* types include reference types of the form $\mathbf{Ref} \theta$; such types contain references that range over values of type θ . *Imperative* types include observation

types of the form $\mathbf{Obs} \tau$; such types contain terms (called *observers*) that observe the state and return values of applicative type τ .

The terms belonging to these types have the following syntax:

$$\begin{aligned} \text{Terms } e ::= & k \mid x \mid v^* \mid \lambda x.e \mid e_1(e_2) \mid \langle e_1, e_2 \rangle \mid e.1 \mid e.2 \\ & \mathbf{letref} \ v^* := e \ \mathbf{in} \ t \mid \mathbf{get} \ x \leftarrow l \ \mathbf{in} \ t \mid l := e ; t \end{aligned}$$

where l, t also range over these terms, l ranging over reference-valued terms and t ranging over observer terms. Terms use two countable sets of variables: *conventional variables* and *reference variables*. Conventional variables x are the usual variables of the typed lambda calculus. Reference variables v^* are a new set of variables that have all the properties of conventional variables and, in addition, the property that distinct reference variables always denote distinct references in any term. This property permits us to reason about the equality of references without recourse to reference constants (which are absent from the language).

In addition to the conventional term forms of typed lambda calculus, there are three new forms which are used to construct observer terms. An observer of type $\mathbf{Obs} \tau$ observes a state to yield a value of type τ . Trivially, any term of type τ is a term of type $\mathbf{Obs} \tau$. In addition:

- an observer ($\mathbf{letref} \ v^* := e \ \mathbf{in} \ t$) allocates a new reference that is not used in the current environment, binds it to the variable v^* , initializes it to the value of e and evaluates the observer t in the extended environment and state.
- an observer ($\mathbf{get} \ x \leftarrow l \ \mathbf{in} \ t$) extends the environment by binding the value referred to by l in the state to the variable x , and evaluates the observer t in the extended environment.
- An observer ($l := e ; t$) updates the state so that reference l refers to the value of e , and evaluates the observer t in the resultant state.

Note that “ $l := e$ ” is not a term by itself (in contrast to Algol-like imperative languages). The modification of l is observable only within t , and there are no side effects produced by an assignment observer. The type system of ILC ensures that l and e are not state-dependent terms, thus ensuring that the state is used in a single-threaded fashion. The state can thus be implemented efficiently as a store.

The typing rules, denotational semantics, and reduction semantics of the language are presented in [SRI91, Swa91]. It is also proved that

- Well-typed terms are strongly normalizing, and
- Reduction has the Church-Rosser property.

The latter means that an outermost evaluation order can be used and so this form of assignment can be used with lazy functional languages like Haskell [HW90].

3 Logical Concepts

In constructive logic, propositions are not simply deemed to be true or false, but their provability issue is considered. For a proposition to be considered true, it must be provable. Conversely, a false proposition is one which has no proof. This view-point stands in opposition to classical logic and gives rise to a separate branch of mathematics, *viz.*, constructive mathematics. From a programming point of view, constructive propositions may be thought of as *specifications* of computations and their proofs as *programs* for the specifications [Con86]. A programming language that forms the language of constructions (proofs) for a constructive logic exhibits logical symmetries and supports reasoning about programs.

The foremost constructive logic is the intuitionistic logic and functional programming forms the language of constructions for it [Abr90, ML82]. That is, the salient constructs of functional programming are obtained as the proof terms for the introduction and elimination rules of intuitionistic logical operators. In a similar fashion, it is appropriate to ask whether imperative programming forms the language of constructions for some constructive logic. We answer this in the affirmative and exhibit a constructive logic called *Observation Type Theory* for which ILC forms the language of constructions. This theory is presented in Section 4. First, we examine the logical concepts underlying the theory.

Central to the theory is a certain correspondence between variables and references. Ever since von Neumann [GvN47], researchers have noted a degree of similarity between variables and references. In fact, in common imperative programming parlance, references are simply called variables. Though the notion of “variance” suggested here is different from that of variables in logic, it is nevertheless true that techniques applicable to variables are often applied to references. For instance, in Hoare logic, references are treated just as variables of logic.

In our formulation, references share many of the important properties of variables. These are that

- variables range over the values of a type,
- they allow quantification of propositions, *e.g.*, $(\forall x: \tau)P$,
- they allow quantified propositions to be specialized,
- they allow abstraction of terms to form “methods”, *e.g.*, $\lambda x.t$, and
- they allow methods to be applied to values.

While references share all these properties, there are two important differences between variables and references. First, while variables are *lexical*, references are *semantic* values. This fact allows references to participate in constructions such as pairing (data structure building),

functional abstraction (parameter passing) and observer abstraction (storing). Variables cannot participate in such semantic constructions.¹ Secondly, references are *reusable*. This permits data structures to be updated without copying.

3.1 Quantification of references

A proof of a universally quantified proposition $(\forall x:\tau)P$ is a *method* (function or rule) which, given a value x of type τ , yields a proof of P . The quantified proposition can be obtained by *generalization* from a judgement of the form $\Gamma, x:\tau \vdash P$. If ϕ is a proof of P , then $\lambda x. \phi$ is a proof method for $(\forall x:\tau)P$. Given any term $e : \tau$, the quantified proposition can be *specialized* to $P[e/x]$. The proof of this is $f(e)$ where f is any proof of $(\forall x:\tau)P$. Thus, function abstraction and application are based on the variables of logic.

In contrast, imperative programming is based on the idea of *references*. A reference is a kind of token that ranges over the values of a certain type. In this respect, references are similar to variables. But, whereas variables are lexical, references are semantic values. They can be embedded in data structures, obtained as results of functions, or stored in other references. There are only two properties known about a reference: its identity and the type of values it ranges over. The latter is determined by the type of the reference itself: a reference of type $\mathbf{Ref} \theta$ ranges over values of type θ . But, the identity of a reference is a tricky issue, particularly because we don't wish to have reference constants. We assume that we have a separate class of variables called *reference variables* which have the property that any two distinct reference variables are always bound to distinct references. The reference variables are distinguished from ordinary variables in the formal treatment by an asterisk superscript, as in v^* .

References give us a new form of universal quantification written as

$$\mathbf{Get} \ x:\theta \Leftarrow l \ \mathbf{in} \ P$$

Here, quantification is over type θ and l must be a reference-valued term of type $\mathbf{Ref} \theta$. The quantified proposition is true iff, for all values v that l may refer to, $P[v/x]$ is true. Since l may refer to any value of type θ , this means the same as $(\forall x:\theta)P$ in classical (truth value) terms. The quantified proposition $\mathbf{Get} \ x:\theta \Leftarrow l \ \mathbf{in} \ P$ can be obtained by *generalization* from a judgement of the form $\Gamma, x:\theta \vdash P$. If ϕ is a proof of P , then the *observer method* ($\mathbf{get} \ x \Leftarrow l \ \mathbf{in} \ \phi$) is a proof of $(\mathbf{Get} \ x:\theta \Leftarrow l \ \mathbf{in} \ P)$.

The role of the reference l in $(\mathbf{Get} \ x:\theta \Leftarrow l \ \mathbf{in} \ P)$ is roughly the same as that of the bound variable x in $(\forall x:\theta)P$. Both are used to range over the values of type θ . (The presence of the

¹Girard's Linear Logic [Gir87] exploits the duality between inputs (variables) and outputs (values). Thus, some effects of references can be obtained in this setting. Logic programming can also obtain some of these effects.

variable x in $\mathbf{Get} \ x:\theta \Leftarrow l \ \mathbf{in} \ P$ may be confusing in seeing this correspondence. Note that x plays a purely *local* role in this expression and the quantification is indeed over l . In contrast, the quantification in $(\forall x:\tau)P$ is over the variable x). The important difference between the two quantifications is that l may be a term whereas x must be a variable. As a special case, l may also be a reference variable of the form v^* .

A **Get**-quantified proposition can be *specialized* in two ways. First, the reference l can be *assigned* the value of some term e . (This is similar to instantiating a variable, but since l is not a variable we use alternative terminology). If ϕ is a proof of $\mathbf{Get} \ x:\theta \Leftarrow l \ \mathbf{in} \ P$ then we can obtain $P[e/x]$ with the proof $(l := e ; \phi)$. Note that the reference l is still retained. It is available for further generalizations and specializations. The crucial benefits of references are obtained from the fact that the *same* reference can be reused for many generalizations and specializations. In contrast, the variable x involved in $(\forall x:\tau)P$ is lost after a specialization.

The second method of **Get**-specialization applies to the special case where l is a reference variable v^* . Here, we not only assign a value to v^* , but also discharge it so that it is not available for further generalizations. Given

$$\Gamma, v^*:\mathbf{Ref} \ \theta \vdash \mathbf{Get} \ x:\theta \Leftarrow v^* \ \mathbf{in} \ P$$

with evidence ϕ , we can derive

$$\Gamma \vdash P[e/x]$$

with proof $(\mathbf{letref} \ v^* := e \ \mathbf{in} \ \phi)$.

3.2 Linearity

The three constructions mentioned in the previous section, $(\mathbf{get} \ x \Leftarrow l \ \mathbf{in} \ \phi)$, $(l := e ; \phi)$, and $(\mathbf{letref} \ v^* := e \ \mathbf{in} \ \phi)$ are called *observer terms*. They have the property that each has a single observer subterm. Thus, observers can only be composed linearly. This is an important property. If multiple observer subterms were allowed, separate copies of the state would be required for each observer (or, side effects must be tolerated).

To ensure that observer terms are linearly composed, we allow such terms to be proofs of only selected forms of propositions. Conventional propositions (of, say, intuitionistic logic) cannot have observer terms as proofs. **Get**-quantified propositions and, in addition, propositions of the form $\mathbf{Obs} \ P$ can have observer proofs. A proposition of the form $\mathbf{Obs} \ P$ means that a proof of P is observable in every state. Under the types-as-propositions correspondence, this is the same as the \mathbf{Obs} type constructor of ILC. Further, $\mathbf{Obs} \ P$ is equivalent to every quantification of the form $\mathbf{Get} \ x:\theta \Leftarrow l \ \mathbf{in} \ P$ where x does not occur free in P . Thus, **Get**-quantification

generalizes the `Obs` type constructor in much the same fashion as \forall quantification generalizes the \rightarrow type constructor of functional programs.²

3.3 Benefits of references

As noted above, references play essentially the same role as variables in conventional logic and `Get`-types play the same role as universal quantification. Why, then, do we need these new forms?

There are several reasons. First, the implications for storage reuse are already apparent. Whenever we use a variable x for generalization and instantiation via the \forall quantifier, we are obliged to use a location for x that is different from the locations for all other variables. On the other hand, the same reference v^* can be used for multiple generalizations and instantiations, indicating to the language processor that the same storage location can be used in each case.³

Second, we are able to construct data structures with references. Given a data structure, we can reuse some of its references for new bindings and retain the old bindings of other references. This saves not only storage locations, but also the computation involved in binding. For example, let l be a pair of type `Ref nat × Ref nat`. Given a construction t of `(Get x:nat ← l.1 in Get y:nat ← l.2 in Q[x,y])`, we can form a construction for `(Get x:nat ← l.1 in Get y:nat ← l.2 in Q[x + 1,y])` as `(get x ← l.1 in l.1 := x + 1 ; t)`. In contrast, if we consider the corresponding logical proposition $(\forall p:\text{nat} \times \text{nat})Q[p.1, p.2]$ with a construction f , the construction for $(\forall p':\text{nat} \times \text{nat})Q[p'.1 + 1, p'.2]$ would be the function $\lambda p'. f(\langle p'.1 + 1, p'.2 \rangle)$. This involves consuming the input pair p' and constructing a new pair $\langle p'.1 + 1, p'.2 \rangle$, whereas the construction for the `Get`-proposition involves only one binding and no construction of new data structures. Such saved effort can grow arbitrarily in large data structures. For instance, consider a specification that quantifies over all linked lists. The evidence for the specification would be indifferent to the length of the linked list and its elements. In instantiating the evidence with a specific list, we can form the list from an existing linked list by extending it. In doing so, we save not only the recycling of storage, but also the *computation* of reconstructing the old portion of the linked list.

Third, references allow *sharing*. Consider a pair of the form $\langle v^*, v^* \rangle$. By instantiating v^* , we achieve the instantiation of *both* the elements of the pair. If such shared references are embedded deep in data structures, their shared instantiation can save arbitrary amounts of computation. For a concrete example, consider the unification problem. The specification is that for every pair of terms t and u , either there exists a most general common instance of t

²The analogy with \forall and \rightarrow is not perfect. `Obs` cannot be *defined* to be a special case of `Get` because it is also needed in contexts where no references are available, but `Get` necessarily involves a reference.

³Compilers for functional languages use a variety of techniques to reuse storage locations allocated for function parameters, but it is doubtful if they can achieve all the storage reuse expressible in a language like ILC.

and u or there is no common instance. There would, in general, be multiple occurrences of a variable z in t . During unification, we have to ensure that all the occurrences z correspond to identical subterms of u . If term variables are modeled by references in the term representations, then mapping an occurrence of z to a subterm of u merely involves assigning the subterm to the reference representing z . This has the effect of constructing a new term t' with all the occurrences of z replaced by this subterm. Such effects cannot be achieved by variables of conventional logic.

In summary, variables of logic have certain properties that are essential to them. These are that variables range over values, propositions can be formed by quantifying over such variables, and quantified propositions can be instantiated with specific values. In the domain of constructions, one can abstract over variables to construct functions, and such functions can be applied to values to obtain specific constructions. All these properties are retained by references. They range over values. Propositions can be formed by quantifying over references (using **Get**), and such quantified propositions can be instantiated. In the domain of constructions, one can construct “observer methods” using **get**, in effect abstracting over references. Such methods can be applied to specific values using assignment or **letref**.

The essential difference between variables and references is that variables are syntactic entities whereas references are semantic values. Therefore, references can participate in semantic constructions like pairs and functions while variables cannot. This allows the various flexibilities outlined above.

3.4 State-assertions

Consider the difference between the following judgements:

$$\begin{aligned} \Gamma, x: \mathbf{nat} &\vdash P[x] \\ \Gamma &\vdash (\forall x: \mathbf{nat})P[x] \end{aligned}$$

Both of them say that there is evidence for $P[x]$ for all natural numbers x (in the context of Γ). However, in the first judgement, the quantification over numbers is contained in the judgement whereas in the latter it is contained in the proposition itself. The former kind of judgements are unavoidable in logic because they form the raw material from which quantified propositions can be obtained by generalization. Formally, a judgement $\Gamma, x: \mathbf{nat} \vdash P$ is valid iff there is a proof term ϕ such that, for all environments η satisfying $(\Gamma, x: \mathbf{nat})$, $\llbracket \phi \rrbracket \eta$ is a proof of $\llbracket P \rrbracket \eta$. A judgement $\Gamma \vdash (\forall x: \mathbf{nat})P$ is valid iff there is a proof method f such that, for all environments η satisfying Γ and all natural numbers i , $\llbracket f \rrbracket \eta i$ is a proof of $\llbracket P \rrbracket \eta[i/x]$.

When we consider observers and **Get**-propositions, judgements involve not only environments but also states. The latter map references to values. As usual, a judgement of the form $\Gamma \vdash Q$ is valid iff there is an observer proof method ϕ such that $\llbracket \phi \rrbracket \eta$ is a proof of $\llbracket Q \rrbracket \eta$, for

all environments η satisfying Γ . However, $\llbracket\phi\rrbracket\eta$ as well as $\llbracket Q\rrbracket\eta$ are both functions of state. This means that the judgement is valid iff, for all environments η satisfying Γ and for all states σ , $\llbracket\phi\rrbracket\eta\sigma$ is a member of $\llbracket Q\rrbracket\eta\sigma$. Note that the states are not constrained by Γ . So, as for the proposition $(\forall x:\mathbf{nat})P$ above, the quantification over states is contained in the proposition rather than the judgement.

This suggests the need for judgements which involve quantification over states. Propositional expressions appearing in such judgements refer to propositions in specific states. We call such expressions *state-assertions* (*assertions* for short) and write them enclosed in braces as $\{Q\}$.⁴ A judgement of the form $\Gamma \vdash \{Q\}$ is valid iff there is an observer proof method ϕ such that, for all environments η and states σ satisfying Γ , $\llbracket\phi\rrbracket\eta\sigma$ is a proof of $\llbracket Q\rrbracket\eta\sigma$. As an example of such a judgement, consider

$$\dots, v^*:\mathbf{Ref\ nat}, \{\mathbf{Get\ } y:\mathbf{nat} \leftarrow v^* \mathbf{in\ } y = t\} \vdash \{\mathbf{Obs\ } (\exists z:\mathbf{nat})z = t\}$$

It states that in every state in which v^* refers to the value of t , it is possible to observe a natural number equal to t . If ϕ is the evidence for the hypothesis **Get**-assertion, then the evidence for the right hand side is $(\mathbf{get\ } y \leftarrow v^* \mathbf{in\ } \langle y, \phi \rangle)$. Note that the corresponding judgement with state-universal propositions

$$\dots, v^*:\mathbf{Ref\ nat}, (\mathbf{Get\ } y:\mathbf{nat} \leftarrow v^* \mathbf{in\ } y = t) \vdash \mathbf{Obs\ } (\exists z:\mathbf{nat})z = t$$

would be trivial because the hypothesis $(\mathbf{Get\ } y:\mathbf{nat} \leftarrow v^* \mathbf{in\ } y = t)$ can have no evidence. (It means the same as $(\forall y:\mathbf{nat})y = t$).

State-assertions are common in informal reasoning about time-varying phenomena. For instance, consider the statement “a free-falling body accelerates at the rate of g ”. There are two possible interpretations of it:

1. For every body x , if x is falling freely in all states, then x accelerates at the rate of g in all states.
2. For every body x and every state σ of x , if x is falling freely in state σ , then x accelerates at the rate of g in σ .

Obviously, the second interpretation is the appropriate one because we want the statement to cover bodies which are not perpetually free-falling. The predicates “free-falling” and “accelerates at the rate of g ” are state-dependent properties or *state-assertions*. However, the entire statement is not a state-assertion but is state-universal: it incorporates an implicit

⁴The notion of state-assertions closely corresponds to the notion of “assertions” in Hoare logic [Hoa69]. So, we use notation reminiscent of the latter.

quantification over all states of x . In reasoning about imperative programs, we require similar quantifications of state-assertions and state-universal propositions.

The following inference rules capture the required forms of reasoning:

Assertion-introduction

$$\frac{\Gamma \vdash Q}{\Gamma \vdash \{Q\}}$$

Assertion-elimination

$$\frac{\Gamma \vdash \{Q\}}{\Gamma \vdash Q} \quad \text{if } \Gamma \text{ has no state-assertions}$$

For the second rule, note that if Γ has no state-assertions and ϕ forms evidence for $\{Q\}$, ϕ is, in fact, evidence for $\{Q\}$ in all states. Thus, ϕ is also evidence for the proposition Q .

Now, state-assertions are rather like propositions except that they are state-dependent. For every logical operator on propositions, there is a corresponding operator for state-assertions. The rules for the state-assertional operators would be the same as those for propositional operators except that they deal with state-assertions. The copy of the standard logic used for state-assertions is called *assertion logic*.

3.5 Noninterference

We must now address a special concern that arises with the use of references. Consider a proposition of the form

$$\mathbf{Get } x:\theta \Leftarrow l \mathbf{ in } \mathbf{Get } y:\theta \Leftarrow l \mathbf{ in } Q[x, y]$$

This is a well-formed proposition. However, it uses the same reference twice for quantification. Even though it lexically appears to have two quantifiers, it is equivalent to the following proposition with one quantification:

$$\mathbf{Get } x:\theta \Leftarrow l \mathbf{ in } Q[x, x]$$

But, this kind of reduction of quantifications is not always possible. A formula of the form

$$\mathbf{Get } x:\theta \Leftarrow l \mathbf{ in } \mathbf{Get } y:\theta \Leftarrow l' \mathbf{ in } Q[x, y]$$

may use two different expressions l and l' which denote the same reference. Or, they may denote the same reference in some states and different references in others. Thus, the degree of quantification involved in a formula is not syntactically discernible.

In a sense, this is the cost we must pay for generalizing syntactic variables to semantic references. But, note that it is precisely this feature — the ability to write distinct expressions which may denote the same reference — that allows the flexibilities mentioned in Section 3.3.

The ambiguity involved in **Get**-quantifications surfaces only when such quantifications are specialized by **Get**-elimination. The inference

$$\frac{\Gamma \vdash t : (\mathbf{Get} \ x : \theta \Leftarrow l \ \mathbf{in} \ Q)}{\Gamma \vdash (l := e ; t) : Q[e/x]}$$

is sound only if Q does not have further quantifications over the reference l . In that case, we say that l does not *interfere* with Q and write it as $l \sim Q$. (This notion of noninterference is similar to that used in Reynolds’s Specification Logic [Rey82, Ten89]). The proposition $l \sim Q$ holds iff l is distinct from each reference used for **Get**-quantification in Q .

To reason about distinctness of references, we stipulate that all the reference variables in an environment denote distinct references. That is, each **letref** operator binds its reference variable to a reference that is not bound to any other reference variable in the environment. This corresponds to the notion of *block structure* in Algol-like languages [Rey81, Ten89].

Assertion logic, the copy of standard logic used for state-assertions, also contains introduction and elimination rules for **Get**. Therefore, we also need noninterference state-assertions of the form $\{l \sim Q\}$. Such an assertion means that l is distinct from all the references used for **Get**-quantification in the assertion $\{Q\}$, i.e., Q interpreted in the current state. To see the need for this, consider an assertion of the form:

$$\{\mathbf{Get} \ y : \mathbf{Ref} \ \theta \Leftarrow w^* \ \mathbf{in} \ \mathbf{Get} \ z : \theta \Leftarrow y \ \mathbf{in} \ Q\}$$

We can say that a reference $v^* : \mathbf{Ref} \ \theta$ does not interfere with this assertion only if w^* does not refer to v^* in the current state. Obviously, such noninterference does not hold for all states (since there are states in which w^* refers to v^*).⁵

4 Observation Type Theory

We now present a formal system based on the ideas presented in the previous section. Following Martin-Löf [ML84], we use the Curry-Howard isomorphism [CF58, How80] and identify propositions and types to achieve a uniform treatment. The constructive logic for which ILC forms the language of constructions is defined as a type theory called *observation type theory* (OTT). The type system of OTT extends the basic type system of ILC without altering the language of terms in a significant way.

4.1 Terms

The abstract syntax of *observation type theory* (OTT) is as follows:

⁵Specification logic does not allow state-specific noninterference assertions. As this example shows, full use of references (arrays, graphs, reference functions) cannot be made without them.

Terms	$e ::= k \mid x \mid v^* \mid \lambda x.e \mid e_1(e_2) \mid \langle e_1, e_2 \rangle \mid e.1 \mid e.2 \mid \mathbf{abort}(e) \mid \mathbf{fact} \mid \mathbf{letref} \ v^* := e \ \mathbf{in} \ t \mid \mathbf{get} \ x \Leftarrow l \ \mathbf{in} \ t \mid l := e ; t$
Applicative types	$\tau ::= \beta \mid \mathbf{void} \mid (\Pi x:\tau_1)\tau_2 \mid (\Sigma x:\tau_1)\tau_2 \mid e_1 = e_2 \ \mathbf{in} \ \tau$
Storage types	$\theta ::= \tau \mid \mathbf{Ref} \ \theta \mid (\Pi x:\theta_1)\theta_2 \mid (\Sigma x:\theta_1)\theta_2 \mid e_1 = e_2 \ \mathbf{in} \ \theta \mid (e_1 =_{\mathit{Ref}} e_2)$
Assertion types	$\alpha ::= \tau \mid \mathbf{Obs} \ \alpha \mid \mathbf{Get} \ x:\theta \Leftarrow l \ \mathbf{in} \ \alpha \mid (\mathbf{All} \ x:\alpha_1)\alpha_2 \mid (\mathbf{Some} \ x:\alpha_1)\alpha_2 \mid e_1 = e_2 \ \mathbf{in} \ \alpha \mid (e_1 =_{\mathit{Ref}} e_2) \mid e \sim \alpha$
Imperative types	$\omega ::= \theta \mid \alpha \mid (\Pi x:\omega_1)\omega_2 \mid (\Sigma x:\omega_1)\omega_2 \mid e_1 = e_2 \ \mathbf{in} \ \omega$

The type system of ILC is extended with empty (**void**) and equality ($e_1 = e_2 \ \mathbf{in} \ T$) types. The product and function space constructions are generalized to dependent product and dependent function space constructions by the following correspondence:

$$\begin{aligned} T_1 \times T_2 &\stackrel{\text{def}}{=} (\Sigma x:T_1)T_2 \quad \text{where } x \notin V(T_2) \\ T_1 \rightarrow T_2 &\stackrel{\text{def}}{=} (\Pi x:T_1)T_2 \quad \text{where } x \notin V(T_2) \end{aligned}$$

where $V(T)$ denotes the set of variables free in T . The Σ and Π operators represent existential and universal quantifications as in Martin-Löf type theory.

The **Obs** τ construct of ILC is enlarged into a new intermediate layer of *assertion types*. It includes, in addition to the **Obs** constructor, the **Get**-quantification discussed in section 3.1, the state-assertional quantifiers **All** and **Some** (which are the state-assertional analogs of the Π and Σ operators), a new reference equality type, and a noninterference type. The following abbreviations are defined for the quantifiers:

$$\begin{aligned} \alpha_1 \ \mathbf{and} \ \alpha_2 &\stackrel{\text{def}}{=} (\mathbf{Some} \ x:\alpha_1)\alpha_2 \quad \text{where } x \notin V(\alpha_2) \\ \alpha_1 \ \mathbf{implies} \ \alpha_2 &\stackrel{\text{def}}{=} (\mathbf{All} \ x:\alpha_1)\alpha_2 \quad \text{where } x \notin V(\alpha_2) \end{aligned}$$

An α type-term can be used as a type as well as a state-assertion. In the latter role, we enclose it in braces $\{\alpha\}$.

The reference equality type ($e_1 =_{\mathit{Ref}} e_2$) is well-formed only if e_1 and e_2 are of some reference types **Ref** θ and **Ref** θ' . It denotes the proposition that e_1 and e_2 denote the same reference. Its main use is to reason about inequality of references in establishing noninterference.

The type formation rules needed to complete the description of the syntax of type terms may be found in [Swa91].

The type-terms of the α layer have a considerable degree of redundancy because they capture implicit dependence on the state. Figure 1 lists the equivalences imposed on these

$$\begin{aligned}
(1) \quad & \text{Obs } \alpha = \text{Get } x:\theta \Leftarrow l \text{ in } \alpha \quad (\text{if } x \notin V(\alpha)) \\
(2) \quad & \text{Obs } (\text{Obs } \alpha) = \text{Obs } \alpha \\
(3) \quad & \text{Get } x_1:\theta_1 \Leftarrow l_1 \text{ in } (\text{Get } x_2:\theta_2 \Leftarrow l_2 \text{ in } \alpha) = \text{Get } x_2:\theta_2 \Leftarrow l_2 \text{ in} \\
& \quad (\text{Get } x_1:\theta_1 \Leftarrow l_1 \text{ in } \alpha) \\
(4) \quad & \text{Get } x:\theta \Leftarrow l \text{ in } (\text{Get } y:\theta \Leftarrow l \text{ in } \alpha) = \text{Get } x:\theta \Leftarrow l \text{ in } \alpha[x/y] \\
(5) \quad & (\text{All } x : (\text{Get } y:\theta \Leftarrow l \text{ in } \alpha_1))\alpha_2 = \text{Get } y:\theta \Leftarrow l \text{ in } (\text{All } x : \alpha_1)\alpha_2 \\
(6) \quad & (\text{All } x : \alpha_1)(\text{Get } y:\theta \Leftarrow l \text{ in } \alpha_2) = \text{Get } y:\theta \Leftarrow l \text{ in } (\text{All } x : \alpha_1)\alpha_2 \\
(7) \quad & (\text{Some } x : (\text{Get } y:\theta \Leftarrow l \text{ in } \alpha_1))\alpha_2 = \text{Get } y:\theta \Leftarrow l \text{ in } (\text{Some } x : \alpha_1)\alpha_2 \\
(8) \quad & (\text{Some } x : \alpha_1)(\text{Get } y:\theta \Leftarrow l \text{ in } \alpha_2) = \text{Get } y:\theta \Leftarrow l \text{ in } (\text{Some } x : \alpha_1)\alpha_2 \\
(9) \quad & (\text{All } x : \tau_1)\tau_2 = (\Pi x : \tau_1)\tau_2 \\
(10) \quad & (\text{Some } x : \tau_1)\tau_2 = (\Sigma x : \tau_1)\tau_2
\end{aligned}$$

Figure 1: Equivalence of α type-terms

terms to eliminate this redundancy. The equivalences 1–4 state the relationship between **Get** and **Obs** operators. The equivalences 4–8 state that a **Get** operator in an argument position of **All** or **Some** can be pulled out and the equivalences 9 and 10 state that **All** and **Some** acting on applicative types are equivalent to Π and Σ .

4.2 Type inhabitation

The inference rules of the constructive logic are presented as rules for type inhabitation. These take the form of introduction and elimination rules for various operators. We only present the rules that have direct relevance to this paper. The full set of rules may be found in [Swa91].

The rules for **void**, Π , Σ and $=$ types (in all layers) are conventional. **Ref** θ types have no rules because their members are only available in the language by allocation of reference variables. The reference equality type has the single introduction rule

$$\frac{e_1 = e_2 \text{ in Ref } \theta}{e_1 =_{\text{Ref}} e_2}$$

$$\frac{\text{Obs-intro} \quad \Gamma \vdash t : \alpha}{\Gamma \vdash t : \mathbf{Obs} \alpha} \quad \frac{\text{Obs-elim} \quad \Gamma \vdash t : \mathbf{Obs} \tau}{\Gamma \vdash t : \tau} \quad (\text{if } \Gamma \text{ has only } \tau \text{ types})$$

Dereference

$$\frac{\Gamma \vdash l : \mathbf{Ref} \theta \quad \Gamma, x : \theta \vdash t : \mathbf{Obs} \alpha}{\Gamma \vdash (\mathbf{get} \ x \leftarrow l \ \mathbf{in} \ t) : (\mathbf{Get} \ x : \theta \leftarrow l \ \mathbf{in} \ \alpha)}$$

Assignment

$$\frac{\Gamma \vdash l : \mathbf{Ref} \theta \quad \Gamma, x : \theta \vdash \mathbf{fact} : l \sim \alpha \quad \Gamma \vdash e : \theta \quad \Gamma \vdash t : (\mathbf{Get} \ x : \theta \leftarrow l \ \mathbf{in} \ \alpha)}{\Gamma \vdash (l := e ; t) : \mathbf{Obs} \alpha[e/x]}$$

Creation (if $v^* \notin V(\alpha)$)

$$\frac{\Gamma \vdash e : \theta \quad \Gamma \vdash l_1 : \mathbf{Ref} \theta_1 \quad \dots \quad \Gamma \vdash l_n : \mathbf{Ref} \theta_n \quad \Gamma, v^* : \mathbf{Ref} \theta, \neg(v^* = l_1), \dots, \neg(v^* = l_n) \vdash t : (\mathbf{Get} \ x : \theta \leftarrow v^* \ \mathbf{in} \ \alpha)}{\Gamma \vdash (\mathbf{letref} \ v^* := e \ \mathbf{in} \ t) : \mathbf{Obs} \alpha[e/x]}$$

Figure 2: Inference rules for observation type theory

Figure 2 presents the introduction and elimination rules for the α layer of OTT. As such, these represent the core of the logic.

Rules *Obs-intro* and *Obs-elim* are coercion rules between applicative terms and observers. *Obs-intro* prescribes that all applicative and state-dependent terms may be coerced to observers, while *Obs-elim* prescribes that state-independent observers may be coerced back to applicative types. These rules are directly carried over from ILC [SRI91].

The *Dereference*, *Assignment*, and *Creation* rules provide for the introduction and elimination of the **Get** operator. These rules generalize the corresponding rules of ILC. The *Dereference* rule is similar to the introduction rule for Π . The difference is that rather than abstract over a variable, we abstract over the content of a reference.

The *Assignment* rule eliminates **Get** operators that represent state-dependence on unaliased references. The premise $t : (\mathbf{Get} \ x : \theta \leftarrow l \ \mathbf{in} \ \alpha)$ means that t is an observer method that proves $\mathbf{Get} \ x : \theta \leftarrow l \ \mathbf{in} \ \alpha$. That is, if t is executed in a state where l refers to, say, a , then it yields a

value that belongs to $\alpha[a/x]$. Thus, assigning e to l before t provides a proof of $\alpha[e/x]$. The additional **Obs** operator in the conclusion **Obs** $\alpha[e/x]$ takes care of the special case where α may be an applicative or storage type (which is not permitted to have an observer term as a member). The premise $l \sim \alpha$ ensures that α has no further **Get**-quantifications over the reference l . So, assigning to l does not alter the meaning of α .

The *Creation* rule is fairly similar to the *Assignment* rule described above. The primary difference is that the reference used for quantification in the premise is a reference variable v^* which is discharged by the inference. The proof term for the conclusion, *viz.*, (**letref** $v^* := e$ **in** t), is responsible for creating a new reference for the discharged variable. The semantics of **letref** guarantees that the reference allocated for v^* is distinct from all other references. Hence, if l_1, \dots, l_n are any references available from the context Γ , the proof of (**Get** $x:\theta \Leftarrow v^*$ **in** α) can assume that v^* is distinct from them. Such assumptions are quite important because they are the only information available for proving noninterference of v^* in a use of the *Assignment* rule. A second difference from the *Assignment* rule is that the noninterference premise $l \sim \alpha$ is not present in the *Creation* rule. Since v^* is distinct from all references available from the context Γ (and α must be well-formed under Γ), indeed, v^* cannot interfere with α .

The noninterference type ($l \sim \alpha$) asserts that the meaning of the type α does not depend on the content of reference l . In our language, a **Get** is the only way for a type to access a reference's content. $l \sim \alpha$ requires that there be no meaningful **Get**'s to the reference l in α . This condition is axiomatized by a collection of rules. We only describe the most interesting rule, namely \sim -**intro-Get**. If α' is of the form **Get** $x:\theta_2 \Leftarrow l'$ **in** α , the meaning of α' may depend on the content of reference l' . Thus, for l to not interfere with α' , l should be distinct from the reference l' :

$$\frac{\Gamma \vdash l : \mathbf{Ref} \theta_1 \quad \Gamma \vdash l' : \mathbf{Ref} \theta_2 \quad \Gamma \vdash \mathbf{fact} : \neg(l = l') \quad \Gamma, x : \theta_2 \vdash \mathbf{fact} : (l \sim \alpha)}{\Gamma \vdash \mathbf{fact} : (l \sim (\mathbf{Get} x:\theta_2 \Leftarrow l' \mathbf{in} \alpha))}$$

It is nontrivial to show that two references l and l' are distinct since l and l' are both expressions and may be aliases of the same reference. The only way to show that they are distinct is to show that they were created by separate instances of the **letref** construct. The inequalities introduced by the *Creation* rule can be used for this purpose.

The **Some** and **All** operators of the α layer are state-assertional operators, *i.e.*, both the arguments to such operators are interpreted in the same state. These operators are handled by a separate set of inference rules collectively referred to as *assertion logic*.

Any assertion type α can be enclosed in braces to form a state-assertion $\{\alpha\}$. A judgement that has a state-assertion on either side of “ \vdash ” is called a *state-judgement*. State-judgements quantify over all states and every state-assertion appearing in such a judgement is interpreted

$$\begin{array}{c}
\textit{Assertion-intro} \\
\frac{\Gamma \vdash t : \alpha}{\Gamma \vdash t : \{\alpha\}}
\end{array}
\quad
\begin{array}{c}
\textit{Assertion-elim} \\
\frac{\Gamma \vdash t : \{\alpha\}}{\Gamma \vdash t : \alpha} \quad (\text{if } \Gamma \text{ has no state-assertions})
\end{array}$$

$$\begin{array}{c}
\textit{Obs-intro} \\
\frac{\Gamma \vdash t : \{\alpha\}}{\Gamma \vdash t : \{\mathbf{Obs} \alpha\}}
\end{array}
\quad
\begin{array}{c}
\textit{Obs-elim} \\
\frac{\Gamma \vdash t : \{\mathbf{Obs} \tau\}}{\Gamma \vdash t : \{\tau\}} \quad (\text{if } \Gamma \text{ has only } \tau \text{ types})
\end{array}$$

Dereference

$$\frac{\Gamma \vdash l : \mathbf{Ref} \theta \quad \Gamma, x : \theta, \mathbf{fact} : \{\mathbf{Get} z : \theta \Leftarrow l \text{ in } x = z\} \vdash t : \{\mathbf{Obs} \alpha\}}{\Gamma \vdash (\mathbf{get} x \Leftarrow l \text{ in } t) : \{\mathbf{Get} x : \theta \Leftarrow l \text{ in } \alpha\}}$$

Assignment

$$\frac{\Gamma \vdash l : \mathbf{Ref} \theta \quad \Gamma, x : \theta \vdash \mathbf{fact} : \{l \sim \alpha\} \quad \Gamma \vdash e : \theta \quad \Gamma \vdash t : (\mathbf{Get} x : \theta \Leftarrow l \text{ in } \alpha)}{\Gamma \vdash (l := e ; t) : \{\mathbf{Obs} \alpha[e/x]\}}$$

Creation if $v^* \notin V(\alpha)$

$$\frac{\Gamma \vdash e : \theta \quad \Gamma \vdash l_1 : \mathbf{Ref} \theta_1 \quad \dots \quad \Gamma \vdash l_n : \mathbf{Ref} \theta_n \quad \Gamma, v^* : \mathbf{Ref} \theta, \neg(v^* = l_1), \dots, \neg(v^* = l_n) \vdash t : (\mathbf{Get} x : \theta \Leftarrow v^* \text{ in } \alpha)}{\Gamma \vdash (\mathbf{letref} v^* := e \text{ in } t) : \{\mathbf{Obs} \alpha[e/x]\}}$$

Figure 3: Inference rules for state-assertions in OTT

with respect to the state pervasive in the judgement. Thus, a judgement of the form $\{\alpha_1\} \vdash \{\alpha_2\}$ means that, in all states satisfying α_1 , there is evidence for α_2 . Types appearing in state-judgements are, however, interpreted in their usual fashion. Assertion logic deals with inferences for state-judgements.

Figure 3 presents the important rules of assertion logic. *Assertion-intro* asserts that if α holds in all states, then α holds in states satisfying Γ . *Assertion-elim* asserts that if α holds in all states that satisfy Γ , and if Γ has no state-assertions (and hence is satisfied by all states), then α holds in all states. These rules coerce between the interpretations of α terms

as state-assertions and types, and are essentially specialization and generalization rules for the implicit quantification over the state.

The remaining rules are the counterparts of the type inferences rules of Fig. 2 for assertion logic. Some of these rules are stronger than the corresponding type inference rules. For instance, the major premise of the *Dereference* rule assumes that x is equal to the content of l in the pervasive state. The *Dereference* rule of Fig. 2, in contrast, requires the premise to be established for all states. Similarly, the *Assignment* rule of assertion logic requires the noninterference premise to be only a state-assertion $\{l \sim \alpha\}$, i.e., it needs to be proved only for the pervasive state, not for all states. This is a powerful rule which is required for reasoning about state-dependent references. Suppose $A : \text{nat} \rightarrow \text{Ref nat}$ is an “array”. If α is of the form

$$\text{Get } x:\text{nat} \Leftarrow l \text{ in Get } y:\text{nat} \Leftarrow A(x) \text{ in } T$$

then $A(0) \sim \alpha$ is not provable because $\neg(A(0) =_{\text{Ref}} A(x))$ does not hold for all states. (There is a state in which l refers to 0). However, the corresponding state-assertion $\{A(0) \sim \alpha\}$ is provable for states in which l does not refer to 0. Thus, reasoning about array-subscripted references (and pointers etc.) requires state-dependent noninterference assertions.

The rules for **Some** and **All** are similar to those for the Σ and Π operators. However, note that in a type of the form $(\text{Some } x : \alpha_1)\alpha_2$, both α_1 and α_2 should be interpreted in the *same* state. Correspondingly, the construction for the type, which would be a pair of the form $\langle t_1, t_2 \rangle$, should also interpret both its components in the same state. Since each component may involve assignments to references, this means that the state must be copied and passed to each component. This is not computationally practical. However, in practice, only one of the components is of computational significance. Typically, t_1 has computational content and t_2 is merely a verification of the fact that t_1 satisfies α_2 . Thus, the constructions produced for assertion types can be tested for their computational feasibility. Alternatively, different type constructors, such as the subset type constructor of Nuprl [Con86], can be used to suppress the noncomputational constructions.

The following rules express the inferences required for **Some** and **All** in assertion logic.

<p style="text-align: center;"><i>Some -introduction</i></p> $\frac{\Gamma \vdash s : \{S\} \quad \Gamma \vdash t : \{T[s/x]\}}{\Gamma \vdash \langle s, t \rangle : \{(\text{Some } x : S)T\}}$	<p style="text-align: center;"><i>Some -elimination</i></p> $\frac{\Gamma \vdash e : \{(\text{Some } x : S)T\}}{\Gamma \vdash e.1 : \{S\}}$ $\frac{\Gamma \vdash e : \{(\text{Some } x : S)T\}}{\Gamma \vdash e.2 : \{T[e.1/x]\}}$
<p style="text-align: center;"><i>All -introduction</i></p> $\frac{\Gamma, x : \{S\} \vdash t : \{T\}}{\Gamma \vdash \underline{\lambda}x.t : \{(\text{All } x : S)T\}}$	<p style="text-align: center;"><i>All -elimination</i></p> $\frac{\Gamma \vdash f : \{(\text{All } x : S)T\} \quad \Gamma \vdash s : \{S\}}{\Gamma \vdash f(\underline{s}) : \{T[s/x]\}}$

The underscored constructors: “·”, “·”, “λ” and “()”, signify the fact that they are “state-indexical”, *i.e.*, all their components are interpreted in the same state.

5 Modelling Hoare Logic

Hoare logic and its extension, Specification logic, can be modelled in observation type theory. Commands, which are conventionally treated as transformers of state, are treated in OTT as observer transformers. A command s is a function that maps an observer c to enhanced observer $s(c)$ which first carries out the actions of s and then continues with the observer c . With this reinterpretation, the Hoare triple $\{P\}s\{Q\}$ corresponds precisely to a judgement of the form

$$\Gamma \vdash \bar{s} : (Q \text{ implies } C) \rightarrow (P \text{ implies } C)$$

where \bar{s} is a function from observers to observers. The argument of \bar{s} , when evaluated in a state satisfying Q , yields a value of some (polymorphic) type C . The observer returned by \bar{s} , when evaluated in a state satisfying P , yields a value of the same type C . The relationship between Hoare-triples and OTT judgements reflects the duality between traditional state-dependent values and ILC’s observers: the value s transforms states satisfying P to states satisfying Q , while the observer transformer \bar{s} transforms observers of states that satisfy Q to observers of states that satisfy P .

The Hoare assignment axiom

$$\{P[e]\} l := e \{P[l]\}$$

is modelled by the following inference using the *Assignment* rule:

$$\frac{\Gamma \vdash t : \text{Get } x:\theta \Leftarrow l \text{ in } P[x] \text{ implies } C}{\Gamma \vdash (l := e ; t) : \text{Obs } (P[e] \text{ implies } C)}$$

So, forward deduction using the rule **Assignment** is remarkably similar to the Floyd-Hoare technique of pushing assertions backwards through assignments.

Since judgements such as the above are no different from other judgements in OTT, it is possible to derive rules such as antecedent-strengthening. The main advantage of OTT over a Hoare-style logic is that its constructions do not have side-effects; state information is localized and we can reason about the state independent of the context. Moreover, since all state-dependencies need to be explicitly stated by means of dereferences, reasoning about aliasing of references is simplified.

6 Conclusion

The fundamental logical symmetries that underly our formulation of references and assignments are remarkably similar to the symmetries that underly variables and functions. These symmetries are embodied in *observation type theory* (OTT), which is a strict extension of a Martin-Löf-style constructive type theory. OTT’s language of constructions (ILC) satisfies all the important properties of type theory’s language of constructions, such as confluence and strong normalization of the reduction relation.

Compared to the conventional type theories, OTT appears somewhat complex. Much of the complexity is in the stratification of the type system into four layers and the need for a separate assertion logic. However, a comparison with other formulations of imperative programming logics, such as Specification logic, suggests that some of this complexity may be inevitable. We continue to investigate further refinement of the present theory.

Much work remains to be done regarding the practical application of the theory for carrying out proofs and program derivations. Some examples may be found in [Swa91]. But, much experience needs to be obtained and it is still too early to pass this as the final word on constructive formulation of imperative programs. The most rewarding aspect of the present formulation is the insight it brings into the nature of references, observer methods and the assignment as a form of method application. It would also be worthwhile to study programming paradigms weaker than references to obtain a better understanding of their nature, *e.g.*, “logic variables” of logic programming [Lin85, WPP77] and the negative types of linear logic [Abr90, Gir87].

The issue of aliasing of references needs to be examined further. Since OTT requires all state dependencies to be specified explicitly by means of dereference operators, it may be possible to reason about noninterference even in the presence of aliasing. This is likely to be aided by modularizing data structures based on their interference characteristics — a systematic study of this would be beneficial. The state-indexical constructions of state-assertions also bear further investigation since they require the state to be copied. For efficiency considerations, their use should be restricted to parts of the proof that do not have meaningful computational content.

References

- [Abr90] S. Abramsky. Computational interpretations of linear logic. Research Report DOC 90/20, Imperial College, London, Oct 1990.
- [CF58] H. B. Curry and R. Feys. *Combinatory Logic*. North-Holland, Amsterdam, 1958.

- [Con86] R. L. Constable, *et. al.* *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, New York, 1986.
- [Eng75] E. Engeler. Algorithmic logic. In J. W. de Bakker, editor, *Foundations of Computer Science*, pages 57–85. Mathematisch Centrum, Amsterdam, 1975. (Mathematical Centre Tracts 63).
- [Gir87] J.-Y. Girard. Linear logic. *Theoretical Comp. Science*, 50:1–102, 1987.
- [GvN47] H. H. Goldstine and J. von Neumann. Planning and coding problems for an electronic computing instrument, Part II. In *John von Neumann, Collected Works, Vol. V*, pages 80–151. Pergamon Press, 1963, Oxford, 1947.
- [Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–583, 1969.
- [How80] W. A. Howard. The formulae-as-types notion of construction. In J. R. Hindley and J. P. Seldin, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, New York, 1980.
- [HW90] P. Hudak and P. Wadler (eds). Report on programming language Haskell, A non-strict purely functional language (Version 1.0). Technical Report YALEU/DCS/RR777, Yale University, Apr 1990.
- [Lin85] G. Lindstrom. Functional programming and the logical variable. In *ACM Symp. on Princ. of Program. Languages*, 1985.
- [ML82] P. Martin-Löf. Constructive mathematics and computer programming. In L. J. Cohen, J. Los, H. Pfeiffer, and K.-P. Podewski, editors, *Proc. Sixth Intern. Congress for Logic, Methodology and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North-Holland.
- [ML84] Per Martin-Löf. *Intuitionistic Type Theory*. Studies in Proof Theory. Bibliopolis, Napoli, 1984.
- [MS87] G. Mirkowska and A. Salwicki. *Algorithmic Logic*. PWN - Polish Scientific Publishers, Warszawa, Poland, 1987.
- [Pra76] V. Pratt. Semantic considerations on Floyd-Hoare logic. In *Symp. on Foundations of Computer Science*, pages 109–121. IEEE, 1976.
- [Rey81] J. C. Reynolds. The essence of Algol. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372. North-Holland, 1981.

- [Rey82] J. C. Reynolds. Idealized Algol and its specification logic. In D. Neel, editor, *Tools and Notions for Program Construction*, pages 121–161. Cambridge Univ. Press, 1982.
- [SRI91] V. Swarup, U. S. Reddy, and E. Ireland. Assignments for applicative languages. In R. J. M. Hughes, editor, *Conf. on Functional Program. Lang. and Comput. Arch.* Springer-Verlag, Berlin, 1991. (Earlier versions: University of Illinois at Urbana-Champaign, July 1990, Revised Nov 1990, Feb 1991).
- [Swa91] V. Swarup. *Type Theoretic Properties of Assignments*. PhD thesis, Univ. Illinois at Urbana-Champaign, 1991. (to appear).
- [Ten89] R. D. Tennent. Denotational semantics of Algol-like languages. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, Vol II*. Oxford University Press, 1989. (to appear).
- [WPP77] D. H. D. Warren, L. M. Pereira, and F. Pereira. Prolog - the language and its implementation compared with Lisp. *SIGPLAN Notices*, 12(8), 1977.