

©Copyright by Vipin Swarup, 1992

TYPE THEORETIC PROPERTIES OF ASSIGNMENTS

BY

VIPIN SWARUP

B.Tech., Indian Institute of Technology, 1984
M.S., University of Illinois, 1988

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1992

Urbana, Illinois

Type Theoretic Properties of Assignments
Vipin Swarup, Ph.D.
University of Illinois at Urbana-Champaign, 1992

This thesis is concerned with extending the correspondence between intuitionistic logic and functional programming to include assignments and dynamic data. We propose a theoretical framework for adding these imperative features to functional languages without violating their semantic properties. We also describe a constructive programming logic that embodies the principles for reasoning about the extended language.

We present an abstract formal language, called Imperative Lambda Calculus (ILC), that extends 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. Thus, programs produce the same results with eager and lazy evaluation orders. ILC permits mutable data structures such as arrays, linked lists, trees, and graphs to be constructed and used. Shared values may be updated destructively rather than by copying. This permits pure functional languages to have efficient implementations of problems such as topological sort, graph reduction, and unification.

We describe the logical symmetries that underlie ILC by exhibiting a constructive logic, called Observation Type Theory (OTT), 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 can be instantiated to specific values. Thus, we obtain a new form of universal quantification that uses references instead of variables. The 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; further, references are *reusable*. These differences allow us to use references in a more flexible fashion, leading to efficiency in constructions and algorithms.

Finally, we describe a higher-order type theory, namely the Nuprl type theory, and illustrate how its inductive types can be used to define well-founded orderings. These can then be used to construct recursive programs.

To my parents,
Bina and Govind Swarup.

Acknowledgments

My thesis advisors, Sam Kamin and Uday Reddy, have influenced me enormously with their inspiration and guidance. Working with them has been an extremely rewarding experience. I thank them for all their help. I owe a special debt of gratitude to Uday Reddy — this thesis would never have been completed without the countless hours of thought and effort devoted by him.

I thank Matthias Felleisen, Ian Mason, Carolyn Talcott, and Phil Wadler for discussions that led to vast improvements in both my understanding and the presentation of this material. I thank John Gray for positive encouragement and discussions that provided me with a uniquely different perspective. I thank Nachum Dershowitz and Simon Kaplan for serving on my committee. I thank NASA and the Computer Science Department at the University of Illinois for financial support. I thank Mark Nadel and Joshua Guttman for giving me the leeway to complete this thesis while working at the MITRE Corporation.

Finally, I thank all my friends during my numerous years in Urbana-Champaign; they made life in that small town interesting, and helped me through the rough times of graduate school. Completing this thesis while working has not been easy — I thank my friends in Boston and my family for never letting me forget about the thesis, much as I wanted to on occasion.

Table of Contents

Chapter

1	Introduction	1
1.1	Foundations	1
1.2	Related Work	4
1.3	An Overview	7
2	Imperative Lambda Calculus (ILC)	9
2.1	Types	9
2.2	Terms	11
2.3	ILC as a Programming Language	15
2.3.1	Notation	15
2.3.2	Examples	16
2.4	Discussion of ILC	18
2.5	Semantics of ILC	21
2.5.1	Denotational Semantics	21
2.5.2	Reduction Semantics	25
2.6	An Abstract View of References	32
2.7	Example: Queues	33
2.8	Example: Unification	36
3	Strong Normalization for ILC	39
3.1	Introduction	39
3.2	Strong Normalization	40
3.3	Reducibility	41
3.4	Neutrality	42
3.5	Properties of Reducibility	42
3.6	Admissibility	48
3.6.1	Pairing	48
3.6.2	Abstraction	48
3.6.3	Dereference	48
3.6.4	Assignment	49
3.7	Reducibility Theorem	51
4	Observation Type Theory (OTT)	52
4.1	Introduction	52
4.2	Simply Typed Lambda Calculus	52

4.3	Simple Constructive Type Theory	53
4.3.1	Types	54
4.3.2	Type Inhabitance	54
4.3.3	Propositional Interpretation	57
4.4	Observation Type Theory	60
4.5	Logical Concepts of OTT	62
4.5.1	Quantification of References	62
4.5.2	Benefits of References	64
4.5.3	State-assertions and State-judgements	66
4.5.4	Noninterference	68
4.5.5	Linearity	70
4.6	Formal System for OTT	71
4.6.1	Types	71
4.6.2	Type Inhabitance	74
4.6.3	Assertion Logic	77
4.7	Modeling Hoare Logic	80
4.8	Examples	82
4.8.1	Notation	82
4.8.2	Example: Factorial	83
4.8.3	A Derived Rule for Sugared Assignments	85
4.8.4	Example: Queues	88
4.9	Predicativity	94
5	Constructive Type Theory (CTT)	96
5.1	Introduction	96
5.2	Type Systems and Type Theories	96
5.3	Constructive Type Theory	97
5.3.1	Typing and Computation Rules	99
5.3.2	Informal Description	99
5.4	The Propositions-as-Types Principle	102
5.5	Notation	104
6	Well-founded Orderings	106
6.1	Introduction	106
6.2	Well-founded Orderings	108
6.3	Proving Orderings to be Well-founded	111
6.3.1	Natural Numbers	111
6.3.2	Cartesian Product	112
6.3.3	Example: Greatest Common Divisor	113
6.3.4	Example: Ackermann's Function	114
6.3.5	Disjoint Union	114
6.3.6	Subset	115
6.3.7	Mapping	115
6.3.8	Example: Quicksort	115
6.3.9	Lists	118
6.4	Comments	120

7	Conclusion	121
	7.1 Results	121
	7.2 Future Directions	122

Appendix

A	Rules for Constructive Type Theory	124
B	Rules for Observation Type Theory	133
	Bibliography	142
	Vita	150

Chapter 1

Introduction

From a broad perspective, this thesis extends the duality between logic and programming languages to include dynamic data, i.e., data that changes over time. Computationally speaking, this thesis adds first-class references and assignments to strongly-typed, pure functional languages without destroying the desirable computational and mathematical properties of these languages. Our motivation is to permit the efficient, yet abstract, manipulation of shared data structures using functional languages. From a logical viewpoint, this thesis extends conventional logic with references (which are entities that describe dynamic data) and a new form of universal quantification that quantifies over the contents of references. References thus play a role similar to that of variables in conventional logic. The proof constructions of the new quantifier correspond to reference creation, dereference and assignment. Our motivation is to provide a clean, logical description of assignments.

In this chapter, we elaborate on our motivation and goals, and compare our approach with related work in the literature. We conclude with an overview of the remaining chapters of this thesis.

1.1 Foundations

Functional languages are popular among computer scientists because of their strong support of modularity. They possess two powerful glues, higher-order functions and laziness, which permit us to modularize programs in new, useful ways. Hughes [38] convincingly argues that “...lazy evaluation is too important to be relegated to second-class citizenship. It is perhaps the most

powerful glue functional programmers possess. One should not obstruct access to such a vital tool.” However, side-effects are incompatible with laziness: programming with them requires knowledge of global context, defeating the very modularity that lazy evaluation is designed to enhance.

Pure functional languages have nice properties that make them easy to reason about. For instance, $+$ is commutative, $=$ is reflexive, and most other familiar mathematical properties hold of the computational operators. This is a consequence of expressions representing *static* values: values that do not change over time. Thus, an expression’s value is independent of the order in which its sub-expressions are evaluated. Side-effects are incompatible with these properties, as side-effects change the values of other expressions, making the order of evaluation important.

Assignments are a means of describing *dynamic* data : data whose values change over time (for example, I/O devices and persistent data bases). In their conventional form, assignments have side-effects on their environment, making their order of evaluation important. Not only are such assignments incompatible with laziness, but they also destroy the nice mathematical properties of pure languages. Hence lazy functional languages shun assignments.

However, since assignments directly model the dynamic behavior of a physical computer’s store, they yield efficient implementations of dynamic data. In contrast, one models dynamic data in functional languages by representing the state explicitly or, possibly, by creating streams of states. Compilation techniques and language notations have been proposed to permit explicit state manipulation to be implemented efficiently [34, 27, 86, 85]. Unfortunately, these methods do not achieve all the linguistic effects of true dynamic data. For instance, dynamic data may be “shared”, i.e., embedded in data structures and accessed via different access paths. When shared dynamic data are updated using assignments, the change is visible to all program points that have access to the data. In contrast, when state is being manipulated explicitly, updating shared data involves constructing a fresh copy of the entire data structure in which the data are embedded, and explicitly passing the copy to all program points that need access to the data. This tends to be tedious and error-prone, and results in poor modularity. An alternative technique is to explicitly represent the shared data structure within a single array, thus circumventing the linguistic absence of shared data in the programming language. This leads to a low-level “Fortran” style of programming, and it is not apparent how to accommodate

heterogeneous data structures. One particularly faces these problems while encoding graph traversal algorithms such as topological sort, unification, and the graph reduction execution model of lazy functional languages.

In this thesis, we propose a theoretical framework for extending functional languages with dynamic data and assignments while retaining the desirable properties of static values. The formal language (called Imperative Lambda Calculus (ILC)) has the following key properties:

- Expressions have static values.

State-dependent and state-independent expressions are distinguished via a type system. The former are viewed as functions from states to values and the functions themselves are static. (Such functions are called *observers* and resemble classical continuations [77, 78]). The type system ensures that this view can be consistently maintained, and limits the interaction between observers in such a way that expressions do not have side-effects.

In the contemporary functional programming community, the terms *assignment* and *side-effect* are sometimes used synonymously. We use the term *side-effect* in its original meaning: an expression has a side-effect if, in addition to yielding a value, it changes the state in a manner that affects the values of other expressions in the context. Assignments in our proposed language do not have such side-effects. Similar comments apply to other terms such as *procedure* and *object*.

- ILC is a strict extension of lambda calculus.

Function abstraction and application have precisely the same meaning as in lambda calculus. This is a key property which is not respected by call-by-value languages like Scheme (even in the absence of side-effects). The operational semantics is presented as a reduction system that consists of the standard reduction rules of lambda calculus together with a set of additional rules; these rules exhibit symmetries similar to those of lambda calculus. The reduction system is *confluent* (or, equivalently, Church-Rosser), and recursion-free terms are *strongly normalizing*.

The utility of ILC is characterized by the following properties:

- Shared dynamic data are available.

Dynamic data are represented by typed objects called *references*. References can refer to

other references as well as to functions. They can be embedded in data structures and used as inputs and outputs of functions.

- Dynamic data may be implemented by a store.

This is achieved by a type system that sequentializes access to regions of the state, much as in effect systems [23] and the languages based on linear logic [27, 86, 87].

- ILC is higher-order.

References, data structures, functions, and observers are all permissible as arguments and results of functions. This permits, for instance, the definition of new control structures, new storage allocation mechanisms, and an object-oriented style of programming.

- ILC is integrated symmetrically.

The applicative sublanguage and the imperative sublanguage are equally powerful and they embed each other. Not only can applicative terms be embedded in imperative terms, but imperative terms can also be embedded in applicative terms. This allows the definition of functions that create and use state internally but are state-independent externally.

This work is also given a logical interpretation: it extends the correspondence between logic and programming to include dynamic data. Entities that describe dynamic data, namely references, play a role similar to that of variables in conventional logic. They range over sets of values and can be instantiated to specific values. Thus, we obtain a new form of universal quantification that uses references instead of variables. The proof terms for the introduction and elimination rules of this quantifier correspond to reference creation, dereference, and assignment. We present a constructive type theory (called *observation type theory* (OTT)) that incorporates these concepts. The language of constructions of this theory is ILC. Just as ILC is a strict extension of simply typed lambda calculus, OTT is a strict extension of simple type theory.

1.2 Related Work

In this thesis, we present both a programming language and its logic. The programming language extends a pure functional language with first-class references and assignments, while the logic embodies the principles for reasoning about this language. These results have been reported in [81] and [80]. In this section, we compare our research with related work in both

areas: programming languages and programming logics. We organize this comparison based on the broad approach taken by the related work.

Linearity Substantial research has been devoted to determining when values of pure functional languages can be modified destructively rather than by copying. Guzman and Hudak [27] propose a typed extension of functional languages called *single threaded lambda calculus* that can express the sequencing constraints required for the in-place update of array-like data structures. Wadler [86] proposes a similar solution using types motivated by Girard’s Linear Logic and, later, shows the two approaches to be equivalent [87]. He also proposes an alternate solution inspired by monad comprehensions [85].

These approaches differ radically from ours in that they do not treat references as values. Programming is still done in the functional style (that is, using our τ types). *Pointers* (references to references) and *objects* (mutable data structures with function components) are absent. Although it is possible to represent references as indices into an array called the store, the result is a low-level “Fortran-style” of programming, and it is not apparent how references of *different types* can be accommodated.

Continuation-based effects Our approach to incorporating state changes is closely related to (and inspired by) continuation-based input/output methods used in functional languages [39, 35, 40, 60, 50]. The early proposal of Haskell incorporated continuation-based I/O as a primitive mechanism, but Haskell version 1.0 defines it in terms of stream-based I/O [35, 36]. Our `Obs` types are a generalization of the Haskell type `Dialog`. In ILC, `Dialog` can be defined as `(Obs Unit)` where `Unit` is a one-element type.

Effect systems An effect system of Gifford and Lucassen [23] is a type system that describes the side-effects that expressions can have. A compiler can then use this information to determine when expressions can be evaluated in parallel, or when they can be memoized without altering the meaning of the program. The side-effect information computed by Gifford and Lucassen assumes an eager order of evaluation; this contrasts with our goal of handling assignments in lazy languages.

Equational axiomatizations Felleisen [19, 20, 21], Mason and Talcott [48, 49] give equational calculi for untyped Scheme-like languages with side effects. The calculi are based on the notion of *observational equivalence*: two terms are equivalent if they yield the same result in all contexts of atomic type. Our reduction system bears some degree of similarity to these calculi. However, the calculi are considerably more complex than our reduction system because of the possibility of side effects. We are investigating the formal relationships between the different approaches.

Laws of programming In a recent paper, Hoare et. al. [32] present an equational calculus for a simple imperative language without procedures. The equations can be oriented as reduction rules and used to normalize recursion-free command phrases. Our work is inspired, in part, by this equational calculus.

Algol-like languages In a series of papers [66, 67], Reynolds describes a language framework called *Idealized Algol* which is later developed into the programming language *Forsythe* [69]. Forsythe has a two-layered operational semantics: the reduction semantics of the typed lambda calculus, and a state transition semantics. The former expands procedure calls to (potentially infinite) normal forms, while the latter executes the commands that occur in the normal forms. Forsythe is based on the principle that the lambda calculus layer is independent of the state transition layer. In particular, references to functions are not permitted because assignments to such references would affect β -expansion.

In contrast, our operational semantics involves a single *unified* reduction system that includes both β -expansion and command execution. Therefore, Forsythe’s restrictions do not appear in our formulation. At the level of terms, ILC contains an applicative sublanguage (in terms of τ types) which is absent in Forsythe. Further, ILC permits state-independent imperative terms to be coerced to applicative types, thereby allowing functions that create and use local state. No similar coercion is available in Forsythe.

Programming logics Hoare Logic [31] was the first significant attempt to formalize reasoning about imperative programs. Although impressive, 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 [67, 83], but the complexity

of this system is deterring. Further, Specification Logic is unable to handle pointer structures. Other approaches, such as weakest precondition logic [15], dynamic logic [62], and algorithmic logics [17, 54], are close to Hoare logic and suffer from the same limitations.

In contrast, Observation Type Theory is motivated by the fundamental logical symmetries that underlie the language ILC. References play a role similar to that of variables in conventional logic. They range over sets of values and can be instantiated to specific values. Thus, we obtain a new form of universal quantification that uses references instead of variables. The proof terms for the introduction and elimination rules of this quantifier correspond to reference creation, dereference, and assignment.

In ILC, expressions do not have side-effects and the reduction system is confluent. This permits OTT to be presented as a strict extension of simple type theory; that is, OTT rules for program constructs like functions are the same as those in purely functional systems like simple type theory.

1.3 An Overview

In Chapter 2, we add references and assignments to the simply typed lambda calculus. We call the extended language *imperative lambda calculus* (ILC). ILC's type system describes the context-sensitive syntax of the language. In addition, it is responsible for ensuring that the language is efficiently implementable. We define ILC's denotational and reduction semantics and prove several properties such as soundness and confluence. We demonstrate the use of ILC through a number of examples. In Chapter 3, we present a proof of strong normalization for the reduction semantics of ILC. This proof is based on Tait-Girard's proof of strong normalization of the second-order lambda calculus. In Chapter 4, we specify a logic for ILC in the framework of constructive type theory. We first demonstrate how the type system of simply typed lambda calculus may be enhanced to yield a simple constructive type theory. An analogous enhancement of ILC's type system yields a constructive type theory for ILC. We call this theory *observation type theory* (OTT). OTT includes entities called assertions that permit it to reason about the state.

Chapter 5 provides a brief review of the concepts and formalisms of a full-fledged constructive type theory. We adopt the Nuprl type theory as the basis of our description. Chapter 6

examines how the inductive type constructors of Nuprl can be used to define well-founded orderings. We present constructive proofs of several well-founded ordering constructors. Examples demonstrate how these orderings can be used to construct well-founded recursive programs within constructive type theory. Chapter 7 concludes this thesis and outlines future directions of research.

Appendix A contains the proof rules for the full-fledged type theory used in the examples of this thesis; these proof rules are drawn from the Nuprl type theory. Appendix B summarizes the formal definition of observation type theory.

Chapter 2

Imperative Lambda Calculus (ILC)

Imperative Lambda Calculus (ILC) is an abstract formal language obtained by extending the typed lambda calculus [55] with imperative programming features. Its main property is that, in spite of this extension, its applicative sublanguage has the same semantic properties as the typed lambda calculus (e.g. confluence and strong normalization). Furthermore, these same properties also hold for the entire language of ILC.

In this chapter, we present ILC's type system and formal semantics, and establish various formal properties such as type soundness, confluence, and strong normalization. Assignments are used meaningfully through a continuation-passing style of programming. This is demonstrated with several examples, including a data type for queues, and unification of first-order terms.

2.1 Types

Let β represent the primitive types of ILC. These may include the natural numbers, characters, strings, etc. The syntax of ILC types is as follows:

$$\begin{aligned}\tau & ::= \beta \mid \tau \times \tau \mid \tau \rightarrow \tau && \text{(Applicative types)} \\ \theta & ::= \tau \mid \mathbf{Ref} \theta \mid \theta \times \theta \mid \theta \rightarrow \theta && \text{(Storage types)} \\ \omega & ::= \theta \mid \mathbf{Obs} \tau \mid \omega \times \omega \mid \omega \rightarrow \omega && \text{(Observer types)}\end{aligned}$$

The type system is stratified into three layers. The *applicative* layer τ contains the types of the simply typed lambda calculus (extended with pairs). These applicative types include the primitive types β and are closed under the product and function space constructions. Note that

we use the term “applicative” to refer to the *classical* values manipulated in lambda calculus; semantically, all three layers of ILC are applicative.

The *storage* layer θ extends the applicative layer with objects called *references*. References are typed values that refer (i.e. point) to values of a particular type. $(\mathbf{Ref} \theta)$ denotes the type of references that refer to values of type θ . References are used to construct a world (called a *store*) that is used for imperative programming. The world is mutable and goes through *states*. The storage layer includes all applicative types and is closed under the type constructors \times, \rightarrow and \mathbf{Ref} . Note that references can point to other references, thereby permitting linked data structures. Tuples of references denote mutable records, while reference-returning functions denote mutable arrays.

Finally, the world of the storage layer needs to be manipulated. In ILC, we take the position that the only manipulation needed for states of the world is *observation* (i.e. inspection). A state may be mutated while being observed, but the mutation is restricted to the observation and is not visible to expressions outside the observation. Thus, in a sense, *the world exists only to be observed*.

Observation of the state is accommodated in the *observer* layer ω . This layer includes all applicative and storage types. In addition, it includes a new type constructor denoted “ $\mathbf{Obs} \tau$ ”. A value of type $\mathbf{Obs} \tau$ is called an *observer*. Such a value observes (i.e. views or inspects) a state and returns a value of type τ . It is significant that the value returned in this fashion is of an applicative type τ . Since a state exists only to be observed, all information about the state is lost when its observation is completed. So, the values observed in this fashion should be meaningful independent of the state, i.e., they should be applicative. An observer type $\mathbf{Obs} \tau$ may be viewed as an implicit function space from the set of states to the type τ .

The three layers can be characterized as *kinds* and given category-theoretic semantics. $\tau, \theta,$ and ω types populate three universes called APP, REF, and OBS respectively. The three universes form cartesian closed categories. Further, APP is a full subcategory of REF, and REF is a full subcategory of OBS. The product and function space constructions have the same meaning in all three universes (see Section 2.5). Thus, there is no ambiguity in treating τ types as also being θ and ω types. The name “Imperative Lambda Calculus” is justified by the property that the semantics of functions in all three layers is the same as that of lambda calculus.

It is worth noting that the θ and ω layers may be conflated into a single layer — the only consequence of this will be that the language will have non-normalizing terms, i.e., there will be some nonrecursive programs whose computations do not terminate. This may be deemed acceptable in a practical programming language.

2.2 Terms

The abstract syntax of unchecked “preterms” is as follows:

$e ::= k$	Constants
x	Conventional variables
v^*	Reference variables
$\lambda x:\omega.e$	Function abstraction
$e e$	Function application
$\langle e, e \rangle$	Pairs
$e.1$	Projection
$e.2$	
$\text{letref } v^*:\text{Ref } \theta := e \text{ in } e$	Creation
$\text{get } x:\theta \Leftarrow e \text{ in } e$	Dereference
$e := e ; e$	Assignment
$\text{obs}(e)$	Type coercion
$\text{app}(e)$	Type coercion

where k ranges over constants, x, v^* range over variables, and ω, θ range over ω, θ types respectively.

The constants of ILC are limited to be of applicative type. Permissible constants include numbers, booleans, characters, and primitive functions on these values. No imperative constants (i.e. no constants involving storage or observer types) are permitted. This enables us to carefully control the creation and use of the state. We (partially) relax this restriction in Section 2.5.1.

The terms of ILC use two countable sets of variables: *conventional variables* and *reference variables*. Conventional variables are the usual variables of the typed lambda calculus. Reference variables are a new set of variables that share all the properties of conventional variables. Further, distinct reference variables within a term always denote distinct references. References are always introduced by binding them to reference variables; conventional variables can then be bound to such references. This property permits us to reason about the equality of references without recourse to reference constants (which are absent from the language). In the

<p><i>Constant</i></p> $\frac{\Gamma \vdash k : \tau}{\text{(if } k \text{ is a constant of type } \tau)}$	<p><i>Weakening</i></p> $\frac{\Gamma \vdash e : \omega}{\Gamma, x : \omega' \vdash e : \omega} \quad \frac{\Gamma \vdash e : \omega}{\Gamma, v^* : \text{Ref } \theta \vdash e : \omega}$
<p><i>Variable hypothesis</i></p> $\Gamma, x : \omega, \Gamma' \vdash x : \omega$	<p><i>Reference hypothesis</i></p> $\Gamma, v^* : \text{Ref } \theta, \Gamma' \vdash v^* : \text{Ref } \theta$
<p>\rightarrow-intro</p> $\frac{\Gamma, x : \omega_1 \vdash e : \omega_2}{\Gamma \vdash (\lambda x : \omega_1. e) : \omega_1 \rightarrow \omega_2}$	<p>\rightarrow-elim</p> $\frac{\Gamma \vdash e_1 : \omega_1 \rightarrow \omega_2 \quad \Gamma \vdash e_2 : \omega_1}{\Gamma \vdash e_1 e_2 : \omega_2}$
<p>\times-intro</p> $\frac{\Gamma \vdash e_1 : \omega_1 \quad \Gamma \vdash e_2 : \omega_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \omega_1 \times \omega_2}$	<p>\times-elim</p> $\frac{\Gamma \vdash e : \omega_1 \times \omega_2}{\Gamma \vdash e.i : \omega_i} \quad \text{for } i = 1, 2$
<p><i>Obs-intro</i></p> $\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \text{obs}(e) : \text{Obs } \tau}$	<p><i>Obs-elim</i></p> $\frac{\Gamma \vdash e : \text{Obs } \tau}{\Gamma \vdash \text{app}(e) : \tau} \quad \text{(if } \Gamma \text{ has only } \tau \text{ types)}$
<p><i>Creation</i></p> $\frac{\Gamma, v^* : \text{Ref } \theta \vdash e_1 : \theta \quad \Gamma, v^* : \text{Ref } \theta \vdash e_2 : \text{Obs } \tau}{\Gamma \vdash (\text{letref } v^* : \text{Ref } \theta := e_1 \text{ in } e_2) : \text{Obs } \tau}$	
<p><i>Dereference</i></p> $\frac{\Gamma \vdash e_1 : \text{Ref } \theta \quad \Gamma, x : \theta \vdash e_2 : \text{Obs } \tau}{\Gamma \vdash (\text{get } x : \theta \leftarrow e_1 \text{ in } e_2) : \text{Obs } \tau}$	
<p><i>Assignment</i></p> $\frac{\Gamma \vdash e_1 : \text{Ref } \theta \quad \Gamma \vdash e_2 : \theta \quad \Gamma \vdash e_3 : \text{Obs } \tau}{\Gamma \vdash (e_1 := e_2 ; e_3) : \text{Obs } \tau}$	

Figure 2.1: Typing rules for ILC terms.

formal presentation, we use an asterisk superscript to distinguish reference variables u^*, v^*, w^* from conventional variables x, y, z . Since the context of a variable determines whether it is a reference or conventional variable, we do not use asterisk superscripts in any of our examples.

Figure 2.1 presents the context sensitive syntax of ILC terms. The syntax is expressed as axioms and inference rules for judgements of the form $(\Gamma \vdash e:\omega)$, where e is a term, ω is a type, and Γ is a set of typing assumptions of the forms $(x:\omega)$ and $(v^*:\mathbf{Ref} \theta)$. Γ contains typing assumptions for all the free variables in e . This notation uses the fact that τ and θ types are also ω types. See [55] for more discussion on this notation.

ILC includes the simply-typed lambda calculus extended with pairs. These terms have their usual meaning in all three layers (rules \rightarrow -*intro*, \rightarrow -*elim*, \times -*intro*, and \times -*elim*). In addition, ILC contains three new observer terms to create a new reference (*creation*), access a reference's content (*dereference*), and modify a reference's content (*assignment*). ILC also contains explicit coercion operators to coerce between applicative and observer types. We now discuss these terms in more detail.

We have seen that there are no reference constants in the language. All references have to be explicitly allocated and bound to a reference variable. This is done by the `letref` construct:

$$\mathbf{letref} \ v^*:\mathbf{Ref} \ \theta := e_1 \ \mathbf{in} \ e_2$$

Such a term is an observer of the same type as e_2 (rule *Creation*). When used to observe a state, it extends the state by creating a new reference, extends the environment by binding v^* to the reference, initializes the reference to the value of e_1 in the extended environment, and finally observes the value of e_2 in the extended environment and state.

The mutable world of references may be inspected by dereferencing a reference, i.e., by inspecting the value that the reference points to, or, using alternate terminology, by inspecting the reference's *content*. If e_1 is a reference-valued expression of type $\mathbf{Ref} \ \theta$, then a term of the form

$$\mathbf{get} \ x:\theta \leftarrow e_1 \ \mathbf{in} \ e_2$$

binds x to the content of e_1 , and denotes the value of e_2 in the extended environment. Here, e_2 must be an observer of type $\mathbf{Obs} \ \tau$, and the entire term is again an observer of type $\mathbf{Obs} \ \tau$ (rule *Dereference*).

Finally, the content of a reference may be modified via assignment observers of the form

$$e_1 := e_2 ; e_3$$

where e_1 is of type $\text{Ref } \theta$ and e_2 is of type θ , for some θ (rule *Assignment*). When used to observe a state, an assignment observer modifies the reference e_1 to refer to e_2 , and observes the value of e_3 in the modified state. Note that “ $e_1 := e_2$ ” is not a term by itself as in conventional languages. The state is modified *for* the observer e_3 , and the entire construct is again an observer.

The lifetime of a mutable world (i.e., a collection of references) is limited to its observation. So, the creation of v^* is observable only within the body e_2 of the creation observer, and the modification of e_1 is observable only within the body e_3 of the assignment observer — there are no side effects produced by the observers. If there are no free occurrences of reference variables or other state-dependent variables in an observer term, then the term is a trivial observer that is independent of any state. Such an observer can be coerced to an applicative term (rule *Obs-elim*). Conversely, every applicative term (every term of a τ type) is trivially coercible to an observer (rule *Obs-intro*).

It is important to note that all the primitive constructions on observers (get, letref and assignment) involve exactly one subterm of an observer type. This reflects the requirement that manipulations of state should be performed in a sequential fashion, similar in spirit to the proposal of single-threaded lambda calculus [27]. Even though it is possible to express functions that accept more than one observer, the state manipulations of such observers have to be eventually sequentialized because there are no multi-ary primitives on observers. (Recall that there are no constants of storage and observer types). This fact has two consequences. First, programming in the imperative sublanguage of ILC requires a continuation-passing style. Second, the state can be implemented efficiently by means of a global store. We return to these issues in Section 2.4.

In the sequel, we shall use l -like variables for terms that denote references, f -like variables for terms that denote functions, t -like variables for terms that denote observers, and e -like variables for arbitrary terms.

2.3 ILC as a Programming Language

ILC can be used as a programming language in different styles. It can be used as a purely applicative language by restricting oneself to applicative types. It can be used as a purely imperative language by mainly using observers (this requires a continuation-passing style of programming). These styles correspond to traditional programming paradigms.

ILC also permits an interesting new style of programming. It permits closed imperative observers to be embedded in applicative terms (via the rule *Obs-elim*). Applicative terms can be freely embedded in imperative observers (via the rule *Obs-intro*). Higher-order functions and laziness can be used to glue together both imperative and applicative subcomputations, though imperative computation is restricted to continuation-passing style.

One extreme of this paradigm is to use ILC with imperative observers at the top level, but with nontrivial applicative subcomputations involving higher-order functions. This use is similar to that of Haskell where state-oriented input/output operations are usually carried out at the top level. More generally, ILC can be used with imperative observers embedded in applicative expressions. This corresponds to the use of side-effect-free function procedures in Algol-like languages.

The examples in this chapter illustrate the use of ILC. Example 1 (factorial) displays how imperative computations can be embedded in applicative terms. Example 2 (a movable point object) exhibits how imperative computations can be encapsulated as closures. The queue and unification examples of Sections 2.7 and 2.8 are significant examples that demonstrate the ability to perform shared updates of data structures. The resulting programs are more efficient than purely functional solutions.

2.3.1 Notation

The need to use `get`'s for dereferencing is rather tedious: it forces us to choose new names, and more importantly, it clutters up the code. The tedium can be alleviated to a large extent through a simple notational mechanism.

Abbreviation: If $(\text{get } x \leftarrow l \text{ in } t[x])$ is an observer term with no occurrence of x in a proper observer subterm of t , then we allow it to be abbreviated as $t[l \uparrow]$. $l \uparrow$ may be read informally as “the current content of reference l ”.

Expansion: If t is an observer term with a particular occurrence of $l \uparrow$, then it is expanded by introducing a “`get`” at the smallest observer subterm of t containing the occurrence of $l \uparrow$.

The intuition behind this abbreviation is that an observer term (`get $x \leftarrow l$ in t`) is a program point at which the content of l is observed, while occurrences of x in t are program points at which that content is used. If l is never modified between the point of dereference and a point of use, then we can safely view the dereference as taking place at the point of use. For example,

$$\begin{aligned}
& (p := n \uparrow * p \uparrow; n := n \uparrow - 1; c) \\
& = \text{get } x \leftarrow n \text{ in get } y \leftarrow p \text{ in} \\
& \quad p := x * y; \\
& \quad \text{get } z \leftarrow n \text{ in} \\
& \quad \quad n := z - 1; c \\
\\
\text{obs}(l \uparrow) & = (\text{get } x \leftarrow l \text{ in obs}(x \uparrow)) \\
& = (\text{get } x \leftarrow l \text{ in get } y \leftarrow x \text{ in obs}(y)) \\
\\
f(l \uparrow) & = (\text{get } x \leftarrow l \text{ in } f(x)) \\
\\
f(\text{obs}(l \uparrow)) & = f(\text{get } x \leftarrow l \text{ in obs}(x))
\end{aligned}$$

2.3.2 Examples

For our examples, we assume that ILC is enhanced with user-defined type constructors and record types (drawn from standard ML [53]). We also assume that ILC is enhanced with *explicit* parametric polymorphism with types ranging over the universe of applicative (τ) types. Implicit polymorphism is problematic in the presence of references and assignments [84] — explicit polymorphism does not suffer from these problems. In our examples, we erase explicit type quantification and type application, and leave it to the reader to fill in the missing information.

For example, we might define a type constructor for binary trees of numbers as follows:

```
datatype BTree = Niltree | Mktree of (nat × BTree × BTree)
```

where `Niltree` is a constant type constructor representing the empty tree. As another example, the record term $\{a = 3, d = \text{true}, c = \lambda x.x\}$ has the record type $\{a: \text{nat}, d: \text{bool}, c: (T \rightarrow T)\}$, where the type variable T is universally quantified over all applicative types.

We also assume primitives such as `case`, `let`, `letrec` and `if-then-else` for all types. Note that these primitives violate our earlier prohibition of primitives over storage and observer types. In Section 2.5.1, we shall see that such primitives are indeed permissible.

Example 1: Factorial

This trivial example is meant to provide an initial feel for the language, and illustrate how imperative observers can be embedded in applicative expressions. This example is not meant to illustrate the benefits of ILC; indeed, a preferred solution is to write this as a tail-recursive applicative function and have the compiler optimize the code into an iterative loop.

```
factorial = λm:nat. app(letref n:Ref nat := m in
                      letref acc:Ref nat := 1 in
                      letrec fact:Obs nat = if (n↑ < 2) then obs(acc↑)
                                             else acc := n↑ * acc↑;
                                             n := n↑ - 1;
                                             fact
                      in fact)
```

The function `factorial` has no free references or state-dependent variables, and so has the applicative type $(\text{nat} \rightarrow \text{nat})$. This means that `factorial` can be freely embedded in applicative expressions even though it contains imperative subcomputations.

Example 2: Points

We implement a point object that hides its internal state and exports operations. Let `Point` be the type of objects that represent movable planar points.

```
Point = {x_coord : (real → Obs T) → Obs T,
         y_coord : (real → Obs T) → Obs T,
         move   : (real × real) → Obs T → Obs T,
         equal  : Point → (bool → Obs T) → Obs T}
```

The function `mkpoint` implements objects of type `Point`.

```

mkpoint  : (real → real → (Point → Obs T) → Obs T)
          = λx. λy. λk.
            letref xc:real := x in
            letref yc:real := y in
            k({x_coord = λk. k(xc↑),
              y_coord = λk. k(yc↑),
              move = λ(dx, dy). λc. xc := xc↑+dx; yc := yc↑+dy; c,
              equal = λ{x_coord, y_coord, move, equal}. λk.
                    x_coord(λx. y_coord(λy.
                    k(x = xc↑ and y = yc↑)))
            })

```

Note, first of all, that the `mkpoint` operation cannot simply yield a value of type `Point` because it is not an applicative value. The extent of `xc` and `yc` is limited to the bodies of the `letrefs` that allocate these references; hence the entire computation that uses these references must occur in these bodies. Therefore, `mkpoint` is defined to accept a point observer function `k` and pass it the newly created point. This is similar to the continuation-passing style of programming. Observers here play the role of continuations. Technically speaking, observers are not continuations because they return values. But, they can be thought of as continuations in the imperative sublanguage so that the “answers” produced can then be consumed in the applicative sublanguage. Such continuation-passing style functions can be defined more conveniently using Wadler’s monad comprehension notation [85]. Note that each operation in the object is similarly defined in continuation-passing style.

This example demonstrates that state-encapsulating closures are available in ILC, albeit in continuation-passing style. Such closures are also representable in semi-functional languages like Scheme and Standard ML, but usually involve side-effects.

2.4 Discussion of ILC

The motivation behind ILC’s type system is threefold. First, we wish to exclude imperative terms that “export” their local effects. Consider the unchecked preterm (`letref v := 0 in v`). This term, if well-typed, would export the locally created reference `v` outside its scope resulting in a dangling pointer. Closures that capture references are prohibited for the same reason —

they export state information beyond its local scope. For example, the unchecked preterm

$$\mathbf{letref} \ v := 0 \ \mathbf{in} \ \lambda x.(v := v \uparrow + x ; v \uparrow)$$

exports information about the reference v outside its scope. The type system prohibits such terms by requiring the value returned by an observer to be applicative and hence free of state information. (Recall that observer types are of the form $\mathbf{Obs} \ \tau$ where τ is an applicative type).

Second, we wish to ensure that the imperative sublanguage can be implemented efficiently without causing side-effects. Consider the unchecked preterm

$$v := 0 ; ((v := 2 ; \mathbf{get} \ x \leftarrow v \ \mathbf{in} \ \mathbf{obs}(x)) + (\mathbf{get} \ x \leftarrow v \ \mathbf{in} \ \mathbf{obs}(x)))$$

In a language with a global store and global assignments (e.g. ML or Scheme), the value of the term depends on the order of evaluation of $+$'s arguments. Further, the term has the side-effect of changing the value of the global reference v to 2. On the other hand, if assignments are interpreted to have local effects, then the value of the term would be 2 regardless of the order of evaluation, and the term would not have any side-effects. However, the state can no longer be implemented by a (global) store. The state needs to be copied and passed to each argument of $+$, making the language quite inefficient.

The type system of ILC excludes such terms from the language by requiring that all state-manipulations be performed in a sequential fashion. Well-typed terms of ILC do not require the state to be copied, and hence the state can be implemented by a (global) store. The only legal way to express the above example in ILC, is to sequentialize its assignments. For example,

$$v := 0 ; \mathbf{get} \ x \leftarrow v \ \mathbf{in} \ (v := 2 ; \mathbf{get} \ y \leftarrow v \ \mathbf{in} \ \mathbf{obs}(y + x))$$

is a well-typed term.

ILC distinguishes between state-dependent observers and applicative values. Both observers and values can be passed to functions and returned as results — it is not necessary to evaluate an observer to a value before passing it. In fact, an observer of the form $(\mathbf{get} \ x \leftarrow l \ \mathbf{in} \ t)$ is in head normal form, just as a lambda expression is in head normal form (see Section 2.5.2). This is a form of laziness and, in fact, directly corresponds to Algol's call by name. However, an

observer passed to a function can only be evaluated in a single state due to the single-threaded nature of the type system. So, the ambiguities caused by Algol’s call-by-name are not shared by ILC.

Finally, we wish the type system to ensure that all recursion-free terms are strongly normalizable, i.e., their evaluation always terminates. We postpone a discussion of this issue to Section 2.5.2. For now, we merely note that strong normalization is achieved by making observers non-storable values. If strong normalization is not considered critical, the θ and ω layers may be conflated.

The type system described thus far is overly restrictive. It prohibits all nonsequential combinations of observers in order to ensure that the state is never copied. For example, a preterm of the form

$$(v := 0 ; (\mathbf{get} \ x \Leftarrow v \ \mathbf{in} \ \mathbf{obs}(x)) + (\mathbf{get} \ x \Leftarrow v \ \mathbf{in} \ \mathbf{obs}(x)))$$

is excluded because the observer arguments of $+$ are combined nonsequentially. However, this term does not require the state to be copied since the arguments of $+$ do not locally modify the state. This suggests that the type system could be relaxed by distinguishing between:

- *creators*, that locally extend the state;
- *pure observers*, that observe the state without locally extending or modifying it; and
- *mutators*, that locally modify the state.

The type system could then permit certain state-dependent terms to be combined. For example, two pure observers could be safely combined. To be effective, such a solution would also have to incorporate a comprehensive type system that captures “effects” of expressions on “regions” of references (similar to that of FX [42]). This would permit combining mutators that mutate disjoint regions of the state. We do not explore this solution in this thesis because it is orthogonal to the issues considered here. It is also clear that such a type system does not completely eliminate the need for sequencing.

2.5 Semantics of ILC

We present the denotational and operational semantics of ILC, and detail the proofs of several important properties including soundness, strong normalization and confluence.

2.5.1 Denotational Semantics

The denotational semantics is defined using complete partial orders (cpo's) as domains. For every primitive type β , choose a domain D_β . $D_{\tau \times \tau}$ and $D_{\tau \rightarrow \tau}$ are defined by the standard product and continuous function space constructions on cpo's.

For every reference type $\mathbf{Ref} \theta$, choose a countable flat domain $D_{\mathbf{Ref} \theta}$. The defined elements of a $D_{\mathbf{Ref} \theta}$ domain should be disjoint from those of any other such domain. The defined elements of these domains may be thought of as “locations”. \mathbf{State} is the set of partial mappings σ from $\bigcup_\theta D_{\mathbf{Ref} \theta}$ to $\bigcup_\theta D_\theta$ with the constraint that, whenever $\alpha \in D_{\mathbf{Ref} \theta}$, $\sigma(\alpha) \in D_\theta$ and $\sigma(\perp_{\mathbf{Ref} \theta}) = \perp_\theta$. The subset of $\bigcup_\theta D_{\mathbf{Ref} \theta}$ mapped by σ is denoted $dom(\sigma)$. σ_0 is the “empty” state, i.e., $dom(\sigma_0)$ contains only $\perp_{\mathbf{Ref} \theta}$ elements.

The domain for an observation type is $D_{\mathbf{Obs} \tau} = [\mathbf{State} \rightarrow D_\tau]$.

An environment η is a mapping from variables to $\bigcup_\omega D_\omega$. If Γ is a type assignment, we say that η satisfies Γ if

- $\eta(x) \in D_\omega$ for every $x:\omega \in \Gamma$,
- $\eta(v^*) \in D_{\mathbf{Ref} \theta}$ for every $v^*:\mathbf{Ref} \theta \in \Gamma$, and
- $\eta(v^*) \neq \eta(w^*)$ for every $v^*, w^*:\mathbf{Ref} \theta \in \Gamma$.

We write $\eta[x \rightarrow v]$ to denote the environment η' that maps the variable x to v , and all other variables y to $\eta(y)$. Similarly, we write $\sigma[\alpha \rightarrow v]$ to denote the state σ' that maps the location α to v , and all other locations α' to $\sigma(\alpha')$.

The denotational semantics of ILC (see Figure 2.2) is defined by induction on type derivations. The meaning of an expression $(\Gamma \vdash e:\omega)$ is a mapping from environments satisfying Γ to D_ω .

Lemma 1 $\llbracket \Gamma \vdash e:\omega \rrbracket$ is well-defined.

This involves showing that continuous functions in the interpretation of λ are unique and that the choice of α in the interpretation of \mathbf{letref} is immaterial.

$$\begin{aligned}
\llbracket \Gamma \vdash x : \omega \rrbracket \eta &= \eta x \\
\llbracket \Gamma \vdash (\lambda x : \omega_1. e) : \omega_1 \rightarrow \omega_2 \rrbracket \eta &= \lambda v \in D_{\omega_1}. \llbracket \Gamma, x : \omega_1 \vdash e : \omega_2 \rrbracket (\eta[x \rightarrow v]) \\
\llbracket \Gamma \vdash e_1 e_2 : \omega_2 \rrbracket \eta &= (\llbracket \Gamma \vdash e_1 : \omega_1 \rightarrow \omega_2 \rrbracket \eta) (\llbracket \Gamma \vdash e_2 : \omega_1 \rrbracket \eta) \\
\llbracket \Gamma \vdash \langle e_1, e_2 \rangle : \omega_1 \times \omega_2 \rrbracket \eta &= \langle \llbracket \Gamma \vdash e_1 : \omega_1 \rrbracket \eta, \llbracket \Gamma \vdash e_2 : \omega_2 \rrbracket \eta \rangle \\
\llbracket \Gamma \vdash e.1 : \omega_1 \rrbracket \eta &= \mathbf{fst}(\llbracket \Gamma \vdash e : \omega_1 \times \omega_2 \rrbracket \eta) \\
\llbracket \Gamma \vdash e.2 : \omega_2 \rrbracket \eta &= \mathbf{snd}(\llbracket \Gamma \vdash e : \omega_1 \times \omega_2 \rrbracket \eta) \\
\\
\llbracket \Gamma \vdash \mathbf{app}(e) : \tau \rrbracket \eta &= \llbracket \Gamma \vdash e : \mathbf{Obs} \tau \rrbracket \eta \sigma_0 \\
\llbracket \Gamma \vdash \mathbf{obs}(e) : \mathbf{Obs} \tau \rrbracket \eta &= \lambda \sigma. \llbracket \Gamma \vdash e : \tau \rrbracket \eta \\
\\
\llbracket \Gamma \vdash (\mathbf{letref} \ v^* : \mathbf{Ref} \ \theta := e_1 \ \mathbf{in} \ e_2) : \mathbf{Obs} \ \tau \rrbracket \eta &= \lambda \sigma. \llbracket \Gamma, v^* : \mathbf{Ref} \ \theta \vdash e_2 : \mathbf{Obs} \ \tau \rrbracket (\eta[v^* \rightarrow \alpha]) (\sigma[\alpha \rightarrow v_{e_1}]) \\
&\quad \text{where } \alpha \text{ is any element of } D_{\mathbf{Ref} \ \theta} \text{ not in } \mathit{dom}(\sigma) \\
&\quad \text{and } v_{e_1} = \llbracket \Gamma, v^* : \mathbf{Ref} \ \theta \vdash e_1 : \theta \rrbracket (\eta[v^* \rightarrow \alpha]) \\
\llbracket \Gamma \vdash (\mathbf{get} \ x : \theta \leftarrow e_1 \ \mathbf{in} \ e_2) : \mathbf{Obs} \ \tau \rrbracket \eta &= \lambda \sigma. \llbracket \Gamma, x : \theta \vdash e_2 : \mathbf{Obs} \ \tau \rrbracket (\eta[x \rightarrow \sigma(\llbracket \Gamma \vdash e_1 : \mathbf{Ref} \ \theta \rrbracket \eta)]) \sigma \\
\llbracket \Gamma \vdash (e_1 := e_2; e_3) : \mathbf{Obs} \ \tau \rrbracket \eta &= \lambda \sigma. \llbracket \Gamma \vdash e_3 : \mathbf{Obs} \ \tau \rrbracket \eta (\sigma[v_{e_1} \rightarrow v_{e_2}]) \\
&\quad \text{where } v_{e_1} = \llbracket \Gamma \vdash e_1 : \mathbf{Ref} \ \theta \rrbracket \eta \text{ and } v_{e_2} = \llbracket \Gamma \vdash e_2 : \theta \rrbracket \eta
\end{aligned}$$

Figure 2.2: Denotational semantics.

Theorem 2 $\llbracket \Gamma \vdash e : \omega \rrbracket \eta \in D_\omega$ whenever η satisfies Γ .

Proof: This is proved by induction on type derivations. The main property to be verified is that η and σ are always extended or modified in a manner type-consistent with Γ . Assume that η satisfies Γ and that the induction hypothesis holds. The proof cases are as follows:

- $\llbracket \Gamma \vdash x : \omega \rrbracket \eta \in D_\omega$
 $\llbracket \Gamma \vdash x : \omega \rrbracket \eta = \eta(x) \in D_\omega$ (since η satisfies Γ).
- $\llbracket \Gamma \vdash (\lambda x : \omega_1. e) : \omega_1 \rightarrow \omega_2 \rrbracket \eta \in D_{\omega_1 \rightarrow \omega_2}$

Let $v \in D_{\omega_1}$. If η satisfies Γ , then $\eta[x \rightarrow v]$ satisfies $(\Gamma, x : \omega_1)$. Then,

$$\begin{aligned}
&\llbracket \Gamma \vdash (\lambda x : \omega_1. e) : \omega_1 \rightarrow \omega_2 \rrbracket \eta \\
&= \lambda v \in D_{\omega_1}. \llbracket \Gamma, x : \omega_1 \vdash e : \omega_2 \rrbracket (\eta[x \rightarrow v]) \\
&\in [D_{\omega_1} \rightarrow D_{\omega_2}] \quad (\text{by induction hypothesis}) \\
&= D_{\omega_1 \rightarrow \omega_2}
\end{aligned}$$

- $\llbracket \Gamma \vdash e_1 e_2 : \omega_2 \rrbracket \eta \in D_{\omega_2}$

By induction hypothesis, $\llbracket \Gamma \vdash e_1 : \omega_1 \rightarrow \omega_2 \rrbracket \eta \in D_{\omega_1 \rightarrow \omega_2} = [D_{\omega_1} \rightarrow D_{\omega_2}]$, and $\llbracket \Gamma \vdash e_2 : \omega_1 \rrbracket \eta \in D_{\omega_1}$. Thus,

$$\llbracket \Gamma \vdash e_1 e_2 : \omega_2 \rrbracket \eta = (\llbracket \Gamma \vdash e_1 : \omega_1 \rightarrow \omega_2 \rrbracket \eta)(\llbracket \Gamma \vdash e_2 : \omega_1 \rrbracket \eta) \in D_{\omega_2}$$

- $\llbracket \Gamma \vdash \langle e_1, e_2 \rangle : \omega_1 \times \omega_2 \rrbracket \eta \in D_{\omega_1 \times \omega_2}$

By induction hypothesis, $\llbracket \Gamma \vdash e_1 : \omega_1 \rrbracket \eta \in D_{\omega_1}$ and $\llbracket \Gamma \vdash e_2 : \omega_2 \rrbracket \eta \in D_{\omega_2}$. Thus,

$$\begin{aligned} \llbracket \Gamma \vdash \langle e_1, e_2 \rangle : \omega_1 \times \omega_2 \rrbracket \eta &= \langle \llbracket \Gamma \vdash e_1 : \omega_1 \rrbracket \eta, \llbracket \Gamma \vdash e_2 : \omega_2 \rrbracket \eta \rangle \\ &\in D_{\omega_1} \times D_{\omega_2} = D_{\omega_1 \times \omega_2} \end{aligned}$$

- $\llbracket \Gamma \vdash e.1 : \omega_1 \rrbracket \eta \in D_{\omega_1}$

$$\llbracket \Gamma \vdash e.1 : \omega_1 \rrbracket \eta = \mathbf{fst}(\llbracket \Gamma \vdash e : \omega_1 \times \omega_2 \rrbracket \eta) \in D_{\omega_1}$$

Similarly, $\llbracket \Gamma \vdash e.2 : \omega_2 \rrbracket \eta \in D_{\omega_2}$

- $\llbracket \Gamma \vdash \mathbf{app}(e) : \tau \rrbracket \eta \in D_{\tau}$

By induction hypothesis, $\llbracket \Gamma \vdash e : \mathbf{Obs} \tau \rrbracket \eta \in D_{\mathbf{Obs} \tau} = [\mathbf{State} \rightarrow D_{\tau}]$. Thus,

$$\llbracket \Gamma \vdash \mathbf{app}(e) : \tau \rrbracket \eta = \llbracket \Gamma \vdash e : \mathbf{Obs} \tau \rrbracket \eta \sigma_0 \in D_{\tau}$$

- $\llbracket \Gamma \vdash \mathbf{obs}(e) : \mathbf{Obs} \tau \rrbracket \eta \in D_{\mathbf{Obs} \tau}$

By induction hypothesis, $\llbracket \Gamma \vdash e : \tau \rrbracket \eta \in D_{\tau}$. Thus,

$$\begin{aligned} \llbracket \Gamma \vdash \mathbf{obs}(e) : \mathbf{Obs} \tau \rrbracket \eta &= \lambda \sigma. \llbracket \Gamma \vdash e : \tau \rrbracket \eta \\ &\in [\mathbf{State} \rightarrow D_{\tau}] = D_{\mathbf{Obs} \tau} \end{aligned}$$

- $\llbracket \Gamma \vdash (\mathbf{letref} \ v^* : \mathbf{Ref} \ \theta := e_1 \ \mathbf{in} \ e_2) : \mathbf{Obs} \ \tau \rrbracket \eta \in D_{\mathbf{Obs} \ \tau}$

Let $\alpha \in D_{\mathbf{Ref} \ \theta}$. Then, since η satisfies Γ , $(\eta[v^* \rightarrow \alpha])$ satisfies $(\Gamma, v^* : \mathbf{Ref} \ \theta)$. Thus, by induction hypothesis,

$$\begin{aligned} \llbracket \Gamma, v^* : \mathbf{Ref} \ \theta \vdash e_2 : \mathbf{Obs} \ \tau \rrbracket (\eta[v^* \rightarrow \alpha]) &\in D_{\mathbf{Obs} \ \tau}, \text{ and} \\ v_{e_1} = \llbracket \Gamma, v^* : \mathbf{Ref} \ \theta \vdash e_1 : \theta \rrbracket (\eta[v^* \rightarrow \alpha]) &\in D_{\theta} \end{aligned}$$

Thus, if σ is a well-formed state, then so is $(\sigma[\alpha \rightarrow v_{e_1}])$, and hence

$$\begin{aligned} \llbracket \Gamma \vdash (\mathbf{letref} \ v^* : \mathbf{Ref} \ \theta := e_1 \ \mathbf{in} \ e_2) : \mathbf{Obs} \ \tau \rrbracket \eta \\ = \lambda \sigma. \llbracket \Gamma, v^* : \mathbf{Ref} \ \theta \vdash e_2 : \mathbf{Obs} \ \tau \rrbracket (\eta[v^* \rightarrow \alpha]) (\sigma[\alpha \rightarrow v_{e_1}]) &\in D_{\mathbf{Obs} \ \tau} \end{aligned}$$

- $\llbracket \Gamma \vdash (\text{get } x: \theta \Leftarrow e_1 \text{ in } e_2): \text{Obs } \tau \rrbracket \eta \in D_{\text{Obs } \tau}$

By induction hypothesis, $\llbracket \Gamma \vdash e_1: \text{Ref } \theta \rrbracket \eta \in D_{\text{Ref } \theta}$. If σ is a well-formed state, then $\sigma(\llbracket \Gamma \vdash e_1: \text{Ref } \theta \rrbracket \eta) \in D_\theta$. Thus $\eta[x \rightarrow \sigma(\llbracket \Gamma \vdash e_1: \text{Ref } \theta \rrbracket \eta)]$ satisfies $(\Gamma, x: \theta)$. Thus,

$$\begin{aligned} & \llbracket \Gamma \vdash (\text{get } x: \theta \Leftarrow e_1 \text{ in } e_2): \text{Obs } \tau \rrbracket \eta \\ &= \lambda \sigma. \llbracket \Gamma, x: \theta \vdash e_2: \text{Obs } \tau \rrbracket (\eta[x \rightarrow \sigma(\llbracket \Gamma \vdash e_1: \text{Ref } \theta \rrbracket \eta)]) \sigma \\ &\in D_{\text{Obs } \tau} \end{aligned}$$

- $\llbracket \Gamma \vdash (e_1 := e_2; e_3): \text{Obs } \tau \rrbracket \eta \in D_{\text{Obs } \tau}$

By induction hypothesis, $v_{e_1} = \llbracket \Gamma \vdash e_1: \text{Ref } \theta \rrbracket \eta \in D_{\text{Ref } \theta}$ and $v_{e_2} = \llbracket \Gamma \vdash e_2: \theta \rrbracket \eta \in D_\theta$. Thus, if σ is a well-formed state, then so is $(\sigma[v_{e_1} \rightarrow v_{e_2}])$. Thus,

$$\begin{aligned} & \llbracket \Gamma \vdash (e_1 := e_2; e_3): \text{Obs } \tau \rrbracket \eta \\ &= \lambda \sigma. \llbracket \Gamma \vdash e_3: \text{Obs } \tau \rrbracket \eta (\sigma[v_{e_1} \rightarrow v_{e_2}]) \\ &\quad \text{where } v_{e_1} = \llbracket \Gamma \vdash e_1: \text{Ref } \theta \rrbracket \eta \text{ and } v_{e_2} = \llbracket \Gamma \vdash e_2: \theta \rrbracket \eta \\ &\in D_{\text{Obs } \tau} \end{aligned}$$

□

The above property ensures that every expression of an applicative type τ is free of state information. It also shows that observers (of type $\text{Obs } \tau$) do not have any visible side-effects. This proves our claim that ILC is free of side-effects.

We note that the semantics uses the state in a single-threaded fashion [71]. Whenever a state is updated, the old state is discarded. Thus, the semantics can indeed be realized by a global store and no side effects need enter the implementation through the “back door”.

At this stage, we can also point out what kind of primitive constants of mutable and observer types may be added to the language without violating the basic framework. The acceptable constants should be purely *combinatorial*, i.e., they should not use any information about the semantic interpretations of their parameters. For example, the constants $if_{\text{Obs } \tau}: \text{bool} \times \text{Obs } \tau \times \text{Obs } \tau \rightarrow \text{Obs } \tau$ defined by

$$if_{\text{Obs } \tau}(p, t_1, t_2) = \begin{cases} t_1, & \text{if } p = \text{true} \\ t_2, & \text{if } p = \text{false} \end{cases}$$

(1)	$(\lambda x:\omega.e_1)(e_2)$	\longrightarrow	$e_1[e_2/x]$	
(2)	$\langle e_1, e_2 \rangle.i$	\longrightarrow	e_i	for $i = 1, 2$
(3)	letref $v^*:\text{Ref } \theta := e_1$ in obs (e_2)	\longrightarrow	obs (e_2)	if $v^* \notin V(e_2)$
(4)	letref $v^*:\text{Ref } \theta := e_1$ in get $x:\theta \Leftarrow v^*$ in e_2	\longrightarrow	letref $v^*:\text{Ref } \theta := e_1$ in $e_2[e_1/x]$	
(5)	letref $v^*:\text{Ref } \theta_1 := e_1$ in get $x:\theta_2 \Leftarrow w^*$ in e_2	\longrightarrow	get $x':\theta_2 \Leftarrow w^*$ in letref $v^*:\text{Ref } \theta_1 := e_1$ in $e_2[x'/x]$	if $v^* \neq w^*$ and $x' \notin V(e_1) \cup V(e_2)$
(6)	$v^* := e_1$; obs (e_2)	\longrightarrow	obs (e_2)	
(7)	$v^* := e_1$; get $x:\theta \Leftarrow v^*$ in e_2	\longrightarrow	$v^* := e_1$; $e_2[e_1/x]$	
(8)	$v^* := e_1$; get $x:\theta \Leftarrow w^*$ in e_2	\longrightarrow	get $x':\theta \Leftarrow w^*$ in $v^* := e_1$; $e_2[x'/x]$	if $v^* \neq w^*$ and $x' \notin V(e_1) \cup V(e_2)$
(9)	app (obs (e))	\longrightarrow	e	

Figure 2.3: Reduction rules.

are acceptable because they are not dependent on the semantic interpretation of the **Obs** τ parameters. On the other hand, the constant $add:\text{Obs } \tau \times \text{Obs } \tau \rightarrow \text{Obs } \tau$ defined by

$$add(t_1, t_2) = \lambda\sigma. (t_1\sigma + t_2\sigma)$$

is not acceptable as it interprets **Obs** τ parameters to be functions of type $[\text{State} \rightarrow D_\tau]$.

2.5.2 Reduction Semantics

We now present reduction rules for terms of ILC. These rules are meant to reduce terms to normal form such that every closed term of a primitive type β reduces to a constant of that type. Let $V(e)$ be the set of free variables of term e . Let $e_1[e_2/x]$ be the result of substituting e_2 for free occurrences of x in e_1 (where bound variables of e_1 are renamed if necessary to avoid capture of free variables in e_2).

The reduction rules presented in Figure 2.3 propagate **get** terms outward (rules (5) and (8)) until they encounter a **letref** or assignment. The **get** construct is then discharged (rules (4) and (7)). The **letref** and assignment constructs can be discharged only after the state observation in their body is completed. At that stage, the body would be a coercion of an

applicative term to an observer, i.e., it would have the form $\text{obs}(e)$. Rules (3) and (6) handle this situation. Rule (9) cancels out inverse type coercions, while rules (1) and (2) are the usual lambda calculus rules of beta-reduction and projection.

We now prove several properties of ILC. The style of presentation follows Mitchell [55].

Lemma 3 (Substitution) *If $(\Gamma, x:\omega_2 \vdash e_1:\omega_1)$ and $(\Gamma \vdash e_2:\omega_2)$ are terms of ILC, then so is the substitution instance $(\Gamma \vdash e_1[e_2/x]:\omega_1)$.*

Let $\xrightarrow{*}$ be the reflexive, transitive closure of \longrightarrow . We can easily show that one step reduction preserves types, and by induction, so does $\xrightarrow{*}$.

Lemma 4 (Type preservation) *If $(\Gamma \vdash s:\omega)$ is a term, and $s \longrightarrow t$, then $(\Gamma \vdash t:\omega)$ is a term.*

Proof: Follows from the typing rules and Lemma 3.

Note that the reduction rules do not make use of types. This fact, together with the type preservation property, implies that type-independent execution of ILC terms is type-correct. Thus, ILC does not require run-time type checking.

Lemma 5 (Soundness of substitution) $\llbracket \Gamma, x:\omega_2 \vdash e_1:\omega_1 \rrbracket (\eta[x \rightarrow \llbracket \Gamma \vdash e_2:\omega_2 \rrbracket \eta]) = \llbracket \Gamma \vdash e_1[e_2/x]:\omega_1 \rrbracket \eta$

Proof: Follows by a simple induction on terms.

Using the above lemma, we can show that one step reduction preserves meaning.

Theorem 6 (Soundness) *Let $(\Gamma \vdash s:\omega)$ and $(\Gamma \vdash t:\omega)$ be terms, and let $s \longrightarrow t$. Then*

$$\llbracket \Gamma \vdash s:\omega \rrbracket \eta \equiv \llbracket \Gamma \vdash t:\omega \rrbracket \eta$$

Proof:

1. $\llbracket \Gamma \vdash (\lambda x:\omega_1.e_1)(e_2):\omega_2 \rrbracket \eta$
 - = $(\llbracket \Gamma \vdash (\lambda x:\omega_1.e_1):\omega_1 \rightarrow \omega_2 \rrbracket \eta)(\llbracket \Gamma \vdash e_2:\omega_1 \rrbracket \eta)$
 - = $\llbracket \Gamma, x:\omega_1 \vdash e_1:\omega_2 \rrbracket (\eta[x \rightarrow v])$
 - where $v = \llbracket \Gamma \vdash e_2:\omega_1 \rrbracket \eta$
 - = $\llbracket \Gamma \vdash e_1[e_2/x]:\omega_2 \rrbracket \eta$

$$\begin{aligned}
2. \quad & \llbracket \Gamma \vdash \langle e_1, e_2 \rangle.1 : \omega_1 \rrbracket \eta \\
& = \mathbf{fst}(\llbracket \Gamma \vdash \langle e_1, e_2 \rangle : \omega_1 \times \omega_2 \rrbracket \eta) \\
& = \mathbf{fst}(\langle \llbracket \Gamma \vdash e_1 : \omega_1 \rrbracket \eta, \llbracket \Gamma \vdash e_2 : \omega_2 \rrbracket \eta \rangle) \\
& = \llbracket \Gamma \vdash e_1 : \omega_1 \rrbracket \eta
\end{aligned}$$

Similarly for $\llbracket \Gamma \vdash \langle e_1, e_2 \rangle.2 : \omega_2 \rrbracket \eta$.

$$\begin{aligned}
3. \quad & \llbracket \Gamma \vdash (\mathbf{letref} \ v^* : \mathbf{Ref} \ \theta := e_1 \ \mathbf{in} \ \mathbf{obs}(e_2)) : \mathbf{Obs} \ \tau \rrbracket \eta \\
& = \lambda \sigma. \llbracket \Gamma, v^* : \mathbf{Ref} \ \theta \vdash \mathbf{obs}(e_2) : \mathbf{Obs} \ \tau \rrbracket (\eta[v^* \rightarrow \alpha])(\sigma[\alpha \rightarrow v_{e_1}]) \\
& \quad \text{where } \alpha \text{ is any element of } D_{\mathbf{Ref} \ \theta} \text{ not in } \mathit{dom}(\sigma) \\
& \quad \text{and } v_{e_1} = \llbracket \Gamma, v^* : \mathbf{Ref} \ \theta \vdash e_1 : \theta \rrbracket (\eta[v^* \rightarrow \alpha]) \\
& = \lambda \sigma. \llbracket \Gamma, v^* : \mathbf{Ref} \ \theta \vdash e_2 : \tau \rrbracket (\eta[v^* \rightarrow \alpha]) \\
& = \lambda \sigma. \llbracket \Gamma \vdash e_2 : \tau \rrbracket \eta \quad \text{since } v^* \notin V(e_2) \\
& = \llbracket \Gamma \vdash \mathbf{obs}(e_2) : \mathbf{Obs} \ \tau \rrbracket \eta
\end{aligned}$$

$$\begin{aligned}
4. \quad & \llbracket \Gamma \vdash (\mathbf{letref} \ v^* : \mathbf{Ref} \ \theta := e_1 \ \mathbf{in} \ (\mathbf{get} \ x : \theta \Leftarrow v^* \ \mathbf{in} \ e_2)) : \mathbf{Obs} \ \tau \rrbracket \eta \\
& = \lambda \sigma. \llbracket \Gamma, v^* : \mathbf{Ref} \ \theta \vdash (\mathbf{get} \ x : \theta \Leftarrow v^* \ \mathbf{in} \ e_2) : \mathbf{Obs} \ \tau \rrbracket (\eta[v^* \rightarrow \alpha])(\sigma[\alpha \rightarrow v_{e_1}]) \\
& \quad \text{where } \alpha \text{ is any element of } D_{\mathbf{Ref} \ \theta} \text{ not in } \mathit{dom}(\sigma) \\
& \quad \text{and } v_{e_1} = \llbracket \Gamma, v^* : \mathbf{Ref} \ \theta \vdash e_1 : \theta \rrbracket (\eta[v^* \rightarrow \alpha]) \\
& = \lambda \sigma. \llbracket \Gamma, v^* : \mathbf{Ref} \ \theta, x : \theta \vdash e_2 : \mathbf{Obs} \ \tau \rrbracket (\eta[v^* \rightarrow \alpha][x \rightarrow v_{e_1}])(\sigma[\alpha \rightarrow v_{e_1}]) \\
& = \lambda \sigma. \llbracket \Gamma, v^* : \mathbf{Ref} \ \theta \vdash e_2[e_1/x] : \mathbf{Obs} \ \tau \rrbracket (\eta[v^* \rightarrow \alpha])(\sigma[\alpha \rightarrow v_{e_1}]) \\
& = \llbracket \Gamma \vdash (\mathbf{letref} \ v^* : \mathbf{Ref} \ \theta := e_1 \ \mathbf{in} \ e_2[e_1/x]) : \mathbf{Obs} \ \tau \rrbracket \eta
\end{aligned}$$

5. Let σ be a well-formed state. Then

$$\begin{aligned}
& \llbracket \Gamma \vdash (\mathbf{letref} \ v^* : \mathbf{Ref} \ \theta_1 := e_1 \ \mathbf{in} \ (\mathbf{get} \ x : \theta_2 \Leftarrow w^* \ \mathbf{in} \ e_2)) : \mathbf{Obs} \ \tau \rrbracket \eta \ \sigma \\
& = \llbracket \Gamma, v^* : \mathbf{Ref} \ \theta_1 \vdash (\mathbf{get} \ x : \theta_2 \Leftarrow w^* \ \mathbf{in} \ e_2) : \mathbf{Obs} \ \tau \rrbracket (\eta[v^* \rightarrow \alpha])(\sigma[\alpha \rightarrow v_{e_1}]) \\
& \quad \text{where } \alpha \text{ is any element of } D_{\mathbf{Ref} \ \theta_1} \text{ not in } \mathit{dom}(\sigma) \\
& \quad \text{and } v_{e_1} = \llbracket \Gamma, v^* : \mathbf{Ref} \ \theta_1 \vdash e_1 : \theta_1 \rrbracket (\eta[v^* \rightarrow \alpha]) \\
& = \llbracket \Gamma, v^* : \mathbf{Ref} \ \theta_1, x : \theta_2 \vdash e_2 : \mathbf{Obs} \ \tau \rrbracket (\eta[v^* \rightarrow \alpha][x \rightarrow v_{w^*}])(\sigma[\alpha \rightarrow v_{e_1}]) \\
& \quad \text{where } v_{w^*} = (\sigma[\alpha \rightarrow v_{e_1}])(\eta[v^* \rightarrow \alpha])(w^*) \\
& \quad \quad = \sigma(\eta(w^*)) \quad \text{since } v^* \neq w^* \\
& = \llbracket \Gamma, v^* : \mathbf{Ref} \ \theta_1, x' : \theta_2 \vdash e_2[x'/x] : \mathbf{Obs} \ \tau \rrbracket (\eta[v^* \rightarrow \alpha][x' \rightarrow v_{w^*}])(\sigma[\alpha \rightarrow v_{e_1}]) \\
& \quad \text{since } x' \notin V(e_2)
\end{aligned}$$

$$\begin{aligned}
& \llbracket \Gamma \vdash (\text{get } x':\theta_2 \leftarrow w^* \text{ in } (\text{letref } v^*:\text{Ref } \theta_1 := e_1 \text{ in } e_2[x'/x]):\text{Obs } \tau) \rrbracket \eta \sigma \\
&= \llbracket \Gamma, x':\theta_2 \vdash (\text{letref } v^*:\text{Ref } \theta_1 := e_1 \text{ in } e_2[x'/x]):\text{Obs } \tau \rrbracket (\eta[x' \rightarrow v'_{w^*}])\sigma \\
&\quad \text{where } v'_{w^*} = \sigma(\eta(w^*)) = v_{w^*} \\
&= \llbracket \Gamma, x':\theta_2, v^*:\text{Ref } \theta_1 \vdash e_2[x'/x]:\text{Obs } \tau \rrbracket (\eta[x' \rightarrow v_{w^*}][v^* \rightarrow \alpha])(\sigma[\alpha \rightarrow v'_{e_1}]) \\
&\quad \text{where } v'_{e_1} = \llbracket \Gamma, x':\theta_2, v^*:\text{Ref } \theta_1 \vdash e_1:\theta_1 \rrbracket (\eta[x' \rightarrow v_{w^*}][v^* \rightarrow \alpha]) \\
&\quad = \llbracket \Gamma, v^*:\text{Ref } \theta_1 \vdash e_1:\theta_1 \rrbracket (\eta[v^* \rightarrow \alpha]) \quad \text{since } x' \notin V(e_1) \\
&\quad = v_{e_1} \\
&= \llbracket \Gamma, x':\theta_2, v^*:\text{Ref } \theta_1 \vdash e_2[x'/x]:\text{Obs } \tau \rrbracket (\eta[x' \rightarrow v_{w^*}][v^* \rightarrow \alpha])(\sigma[\alpha \rightarrow v_{e_1}]) \\
&= \llbracket \Gamma \vdash (\text{letref } v^*:\text{Ref } \theta_1 := e_1 \text{ in } (\text{get } x:\theta_2 \leftarrow w^* \text{ in } e_2)):\text{Obs } \tau \rrbracket \eta \sigma
\end{aligned}$$

$$\begin{aligned}
6. \quad & \llbracket \Gamma \vdash (v^* := e_1 ; \text{obs}(e_2)):\text{Obs } \tau \rrbracket \eta \\
&= \lambda\sigma. \llbracket \Gamma \vdash \text{obs}(e_2):\text{Obs } \tau \rrbracket \eta (\sigma[\eta(v^*) \rightarrow v_{e_1}]) \\
&\quad \text{where } v_{e_1} = \llbracket \Gamma \vdash e_1:\theta \rrbracket \eta \\
&= \lambda\sigma. \llbracket \Gamma \vdash e_2:\tau \rrbracket \eta \\
&= \llbracket \Gamma \vdash \text{obs}(e_2):\text{Obs } \tau \rrbracket \eta
\end{aligned}$$

$$\begin{aligned}
7. \quad & \llbracket \Gamma \vdash (v^* := e_1 ; (\text{get } x:\theta \leftarrow v^* \text{ in } e_2)):\text{Obs } \tau \rrbracket \eta \\
&= \lambda\sigma. \llbracket \Gamma \vdash (\text{get } x:\theta \leftarrow v^* \text{ in } e_2):\text{Obs } \tau \rrbracket \eta (\sigma[\eta(v^*) \rightarrow v_{e_1}]) \\
&\quad \text{where } v_{e_1} = \llbracket \Gamma \vdash e_1:\theta \rrbracket \eta \\
&= \lambda\sigma. \llbracket \Gamma, x:\theta \vdash e_2:\text{Obs } \tau \rrbracket (\eta[x \rightarrow v_{e_1}])(\sigma[\eta(v^*) \rightarrow v_{e_1}]) \\
&= \lambda\sigma. \llbracket \Gamma \vdash e_2[e_1/x]:\text{Obs } \tau \rrbracket \eta (\sigma[\eta(v^*) \rightarrow v_{e_1}]) \\
&= \llbracket \Gamma \vdash (v^* := e_1 ; e_2[e_1/x]):\text{Obs } \tau \rrbracket \eta
\end{aligned}$$

8. Let σ be a well-formed state. Then,

$$\begin{aligned}
& \llbracket \Gamma \vdash (v^* := e_1 ; (\text{get } x:\theta \leftarrow w^* \text{ in } e_2)):\text{Obs } \tau \rrbracket \eta \sigma \\
&= \llbracket \Gamma \vdash (\text{get } x:\theta \leftarrow w^* \text{ in } e_2):\text{Obs } \tau \rrbracket \eta (\sigma[\eta(v^*) \rightarrow v_{e_1}]) \\
&\quad \text{where } v_{e_1} = \llbracket \Gamma \vdash e_1:\theta_1 \rrbracket \eta \\
&= \llbracket \Gamma, x:\theta \vdash e_2:\text{Obs } \tau \rrbracket (\eta[x \rightarrow v_{w^*}])(\sigma[\eta(v^*) \rightarrow v_{e_1}]) \\
&\quad \text{where } v_{w^*} = (\sigma[\eta(v^*) \rightarrow v_{e_1}])(\eta(w^*)) \\
&\quad = \sigma(\eta(w^*)) \quad \text{since } v^* \neq w^* \\
&= \llbracket \Gamma, x':\theta \vdash e_2[x'/x]:\text{Obs } \tau \rrbracket (\eta[x' \rightarrow \sigma(\eta(w^*))])(\sigma[\eta(v^*) \rightarrow v_{e_1}]) \quad \text{since } x' \notin V(e_2)
\end{aligned}$$

$$\begin{aligned}
& \llbracket \Gamma \vdash (\text{get } x':\theta \Leftarrow w^* \text{ in } (v^* := e_1 ; e_2[x'/x])) : \mathbf{Obs } \tau \rrbracket \eta \sigma \\
&= \llbracket \Gamma, x':\theta \vdash (v^* := e_1 ; e_2[x'/x]) : \mathbf{Obs } \tau \rrbracket (\eta[x' \rightarrow \sigma(\eta(w^*))]) \sigma \\
&= \llbracket \Gamma, x':\theta \vdash e_2[x'/x] : \mathbf{Obs } \tau \rrbracket (\eta[x' \rightarrow \sigma(\eta(w^*))]) (\sigma[\eta(v^*) \rightarrow v'_{e_1}]) \\
&\quad \text{where } v'_{e_1} = \llbracket \Gamma, x':\theta \vdash e_1 : \theta_1 \rrbracket (\eta[x' \rightarrow \sigma(\eta(w^*))]) \\
&\quad = \llbracket \Gamma \vdash e_1 : \theta_1 \rrbracket \eta \quad \text{since } x' \notin V(e_1) \\
&\quad = v_{e_1} \\
&= \llbracket \Gamma, x':\theta \vdash e_2[x'/x] : \mathbf{Obs } \tau \rrbracket (\eta[x' \rightarrow \sigma(\eta(w^*))]) (\sigma[\eta(v^*) \rightarrow v_{e_1}]) \\
&= \llbracket \Gamma \vdash (v^* := e_1 ; (\text{get } x:\theta \Leftarrow w^* \text{ in } e_2)) : \mathbf{Obs } \tau \rrbracket \eta \sigma
\end{aligned}$$

$$\begin{aligned}
9. & \llbracket \Gamma \vdash \text{app}(\text{obs}(e)) : \tau \rrbracket \eta \\
&= \llbracket \Gamma \vdash \text{obs}(e) : \mathbf{Obs } \tau \rrbracket \eta \sigma_0 \\
&= \llbracket \Gamma \vdash e : \tau \rrbracket \eta
\end{aligned}$$

□

Strong normalization is considered to be a desirable property of typed programming languages. It asserts that the evaluation of a well-typed recursion-free term *always* terminates. Conceptually, its significance is that all terms are meaningful; there are no undefined terms [63]. Its pragmatic implication is that nontermination is limited to explicit recursion. Strong normalization is ensured in ILC by making observers nonstorable. If observers were storable, **Ref Obs nat** would be a well-formed type and the language would contain the following infinite reduction sequence:

$$\begin{aligned}
& \text{letref } u^* : \mathbf{Ref Obs nat} := (u^* \uparrow) \text{ in } (u^* \uparrow) \\
&\quad \longrightarrow \text{letref } u^* : \mathbf{Ref Obs nat} := (u^* \uparrow) \text{ in } (u^* \uparrow) \\
&\quad \longrightarrow \dots
\end{aligned}$$

Since $u^* : \mathbf{Ref Obs nat}$, we can store in it an observation of itself, namely $(u^* \uparrow)$. Indeed, recursion is defined in Scheme by a similar device [64].

In a typed language with references (e.g. ML), the Y combinator can be expressed using references as follows:

$$\begin{aligned}
Y &\stackrel{\text{def}}{=} \lambda N : (S \rightarrow T) \rightarrow (S \rightarrow T). \lambda f_0 : (S \rightarrow T). \\
&\quad \text{let } u = \text{ref } f_0 \text{ in } (u := \lambda x : S. (N(!u)(x))); !u
\end{aligned}$$

where f_0 is some dummy function of type $S \rightarrow T$ that is used to initialize the reference u . As usual, the Y combinator can be used to define recursive functions. For example, if $\text{id} : (\text{nat} \rightarrow \text{nat})$ is the identity function, we can define the factorial function as follows:

$$\text{factorial} \equiv Y(\lambda F : (\text{nat} \rightarrow \text{nat}). \lambda n : \text{nat}. \text{if } (n = 0) \text{ then } 1 \text{ else } n * F(n - 1))(\text{id})$$

Y is not a well-typed ILC term, nor would it be well-typed even if observers were storable. However, the semantics of ILC interprets the type $\text{Obs } \tau$ as a function space domain, namely $[\text{State} \rightarrow \tau]$. We can thus specialize the Y combinator to:

$$\begin{aligned} Y' \equiv & \lambda N : (\text{Obs } \tau \rightarrow \text{Obs } \tau). \lambda f_0 : \text{Obs } \tau. \\ & \text{letref } u^* : \text{Ref Obs } \tau := f_0 \text{ in} \\ & (u^* := (\text{get } y \leftarrow u^* \text{ in } N(y))); u^* \uparrow \end{aligned}$$

The non-normalizing preterm we presented at the beginning of this discussion is a simplified version of the preterm $Y'(\lambda x.x)(\text{obs}(0))$. This is ill-typed only because $\text{Ref Obs } \tau$ is not a well-formed type. If observers were storable values, this would be well-typed and ILC would have non-normalizing, recursion-free terms. Note that this discussion suggests that the crux of the problem is the interpretation of the type $\text{Obs } T$ as $[\text{State} \rightarrow T]$. In Section 4.9, we shall suggest a type system that might provide us with both strong normalization and storable observers.

Proposition 7 (Strong Normalization) *Let $(\Gamma \vdash t : \omega)$ be a recursion-free term. Then there is no infinite reduction sequence $t \longrightarrow t_1 \longrightarrow t_2 \longrightarrow \dots$ of ILC terms.*

Proof: The proof of the above proposition is quite elaborate and uses twin induction on types and terms, along the classical lines of [82, 26]. We postpone its presentation to Chapter 3.

□

The Church-Rosser property for the reduction system may be established as follows.

Lemma 8 (Local Confluence) *If $(\Gamma \vdash r : \omega)$ is a term, and if $r \longrightarrow s_1$ and $r \longrightarrow s_2$, then there is a term $(\Gamma \vdash t : \omega)$ such that $s_1 \xrightarrow{*} t$ and $s_2 \xrightarrow{*} t$.*

Proof: Let R be a countably infinite set of reference variables. Treat the reduction rules (4) and (7) as schematic rules representing an infinite set of rules, one for each $v^* \in R$. Similarly,

the rules (5) and (8) may be treated as being schematic for an infinite set of rules, one for each distinct pair of $v^*, w^* \in R$. The resulting reduction system has no “critical overlaps”. A critical overlap occurs when a left hand side has a common instance with a subterm of another left hand side, where the subterm is not a metavariable. Further, the reduction system is left-linear, i.e., there are no repeated occurrences of metavariables on any left hand side. The result then follows from Huet [37], Lemma 3.3.

□

Theorem 9 (Confluence) *If $(\Gamma \vdash r:\omega)$ is a term, and if $r \xrightarrow{*} s_1$ and $r \xrightarrow{*} s_2$, then there is a term $(\Gamma \vdash t:\omega)$ such that $s_1 \xrightarrow{*} t$ and $s_2 \xrightarrow{*} t$.*

Proof: This follows from Proposition 7, Lemma 8, and Newman’s Lemma.

□

This result can be extended to the language with recursion as follows. Add the following reduction rule

$$(10) \quad \text{fix } e \longrightarrow e(\text{fix } e)$$

where fix is the least fixed point operator for each type ω . The resulting system still has no critical overlaps. Further, it is left-linear, i.e., there are no repeated occurrences of metavariables on any left hand side. Hence, by Huet [37], Lemma 3.3, we have

Proposition 10 *The reduction system (1-10) is confluent.*

This property, which is equivalent to the Church-Rosser property, gives further evidence of the side-effect-freedom of ILC. If there were side effects, then the evaluation of a subexpression would affect the meaning of its context, and normal forms would be dependent on the evaluation order.

The independence of results on the evaluation order means, in particular, that call-by-value and call-by-name evaluations produce the same results. This observation must be interpreted carefully. In the lambda calculus setting, the distinction between call-by-value and call-by-name refers to when the arguments to a function are *evaluated*. We are using these terms

in the same sense. However, in the imperative programming framework, the terms call-by-value and call-by-name are used to make a different distinction — the question of when the arguments to a function are *observed*. In the terminology of ILC, this involves a coercion from a type $\mathbf{Obs} \tau$ to a type τ . Since $\mathbf{Obs} \tau$ represents the function space $[\mathbf{State} \rightarrow D_\tau]$, such a coercion involves change of semantics. ILC permits no such coercion. Thus, in the imperative programming sense, ILC’s parameter passing is call-by-name. However, the linearity of observer constructions means that a function accepting an observer can use it to observe at most one state. This contrasts with Algol 60, where a call-by-name parameter can be used to observe many states with quite unpredictable effects.

2.6 An Abstract View of References

In ML, references are first-class values. They may be stored in other references, and may be passed as arguments and results of functions. A reference is allocated using a creation construct, e.g. `let` or `letrec`. A reference v is dereferenced by the expression `!v`. The content of a reference is modified by assignment, e.g. `(v := 3)`.

A reference has a dual role as a *producer* (or R-value) and an *acceptor* (or L-value), depending on whether its contents are being accessed or modified [66, 83]. A dereference operation `!v` coerces the reference v to a producer that takes the current state and returns a value (the reference’s value in the state) along with the argument state. An assignment operation `v := e` coerces the reference v to an acceptor that takes a value e to be assigned, and the current state, and returns a new state (after performing the assignment).

From this viewpoint, a reference type $\mathbf{Ref} \theta$ is an abbreviation of the type $(\mathbf{St} \rightarrow (\theta \times \mathbf{St})) \times (\theta \rightarrow \mathbf{St} \rightarrow \mathbf{St})$, where \mathbf{St} is the type of states. Let `fst` and `snd` be the first and second projection operators, and let σ be the current state. Then `!v` is notation for `fst(v)(σ)` while `v := e` is notation for `snd(v)(e)(σ)`. Note that the state is not explicit in the notation, but is implicitly applied by the language.

References in ILC may be viewed as the continuation-passing-style analog of the above references. Let τ be an arbitrary type. Then the ILC type $\mathbf{Ref} \theta$ may be viewed as an abbreviation of the type $(\mathbf{St} \rightarrow ((\theta \times \mathbf{St}) \rightarrow \tau) \rightarrow \tau) \times (\theta \rightarrow \mathbf{St} \rightarrow (\mathbf{St} \rightarrow \tau) \rightarrow \tau)$. As a producer, a reference takes a state and a continuation function, and returns the result of

applying the continuation function to the contents of the reference in the state (paired with the argument state). As an acceptor, a reference takes a value to be assigned, a state, and a continuation. It performs the assignment, evaluates the continuation in the modified state, and returns the result.

It is immaterial whether the producers and acceptors take the argument state before or after taking the argument continuation. Hence, we can rewrite the type of a reference as: $((\theta \rightarrow \mathbf{St} \rightarrow \tau) \rightarrow \mathbf{St} \rightarrow \tau) \times (\theta \rightarrow (\mathbf{St} \rightarrow \tau) \rightarrow \mathbf{St} \rightarrow \tau)$. The state is made implicit by abbreviating the type $(\mathbf{St} \rightarrow \tau)$ to the new ILC type $\mathbf{Obs} \tau$.

We thus have that $\mathbf{Ref} \theta$ may be viewed as an abbreviation of the type

$$((\theta \rightarrow \mathbf{Obs} \tau) \rightarrow \mathbf{Obs} \tau) \times (\theta \rightarrow \mathbf{Obs} \tau \rightarrow \mathbf{Obs} \tau)$$

With this interpretation, $(\mathbf{get} \ x \leftarrow l \ \mathbf{in} \ t)$ is notation for $\mathbf{fst}(1)(\lambda x. t)$, while $(l := e ; t)$ is notation for $\mathbf{snd}(1)(e)(t)$.

To summarize, ILC's references and observers are the continuation-passing-style analogs of references and state-dependent terms of semi-functional languages such as ML.

2.7 Example: Queues

We now present an example that performs *shared updates* on a data structure. The example is the traditional data structure example of queues.

$$\begin{aligned} \mathbf{type} \ \mathbf{QADT}(S) = \{ & \mathbf{insert} : S \rightarrow \mathbf{Obs} \ T \rightarrow \mathbf{Obs} \ T, \\ & \mathbf{isempty} : \mathbf{Obs} \ T \rightarrow \mathbf{Obs} \ T \rightarrow \mathbf{Obs} \ T, \\ & \mathbf{getfront} : (S \rightarrow \mathbf{Obs} \ T) \rightarrow \mathbf{Obs} \ T, \\ & \mathbf{delete} : \mathbf{Obs} \ T \rightarrow \mathbf{Obs} \ T \\ & \} \end{aligned}$$

$\mathbf{QADT}(S)$ is an abstract data type whose members are concrete imperative queue implementations, with the queues containing elements of type S . An implementation is a record structure containing four operations that manipulate the representation type of queues. A queue Q is used by writing a sequence of operations that refer to it. For example,

$$Q.\mathbf{insert}(3)(Q.\mathbf{insert}(4)(Q.\mathbf{getfront}(\lambda m. Q.\mathbf{delete}(Q.\mathbf{getfront}(\lambda n.n - m))))))$$

evaluates to 1.

```

Qcreate:  $S \rightarrow (\text{QADT} \rightarrow \text{Obs } T) \rightarrow \text{Obs } T$ 
=  $\lambda d.\lambda k.$  letref rnew: Ref Lnklist  $S := \text{Nil}$  in
  letref front: Ref Lnklist  $S := \text{Cons}(d, \text{rnew})$  in
  letref rear: Ref Lnklist  $S := \text{front} \uparrow$  in
  let QImpl: QADT
    = { insert =  $\lambda x.\lambda c.$  letref rnew: Ref Lnklist  $S := \text{Nil}$  in
      rear  $\uparrow$  .2 := Cons( $x$ , rnew);
      rear := rear  $\uparrow$  .2  $\uparrow$ ;  $c$ 
      isempty =  $\lambda c.\lambda c'.$  if front  $\uparrow$  = rear  $\uparrow$  then  $c$  else  $c'$ 
      getfront =  $\lambda k.$   $k(\text{front} \uparrow .2 \uparrow .1)$ 
      delete =  $\lambda c.$  front := front  $\uparrow$  .2  $\uparrow$ ;  $c$ 
    }
  in  $k(\text{QImpl})$ 

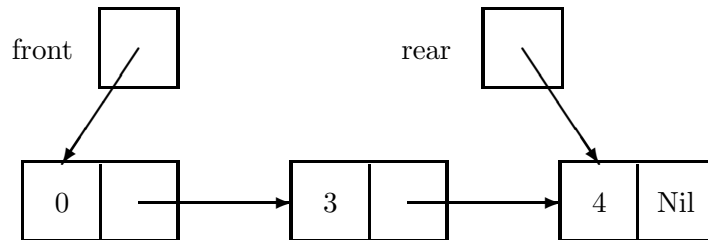
```

Figure 2.4: Queue implementation.

We implement a queue by a mutable linked list. A node in the linked list is either `Nil` (representing the end of the list), or is a pair comprising an element and a pointer (i.e., a reference) to another node.

```
datatype Lnklist  $S = \text{Nil} \mid \text{Cons of } (S \times \text{Ref Lnklist } S)$ 
```

A queue is represented by a pair of references `front` and `rear` pointing to the first and last nodes of a linked list respectively. The linked list is always nonempty. It contains a dummy node at the beginning. For example, the figure below depicts the linked list representation after the numbers 3 and 4 have been inserted into an empty queue, with 3 being inserted first, and with the dummy (header) node initialized to 0.



The queue implementation is defined by the function `Qcreate` in Figure 2.4. This function creates an empty queue and passes to its argument function a queue implementation that uses this new queue. The linked list for an empty queue consists of the single, dummy (header) node. The first argument to `Qcreate` is a dummy element that is stored in this header node. The initializations of `front` and `rear` correspond to this representation. The `insert` operation extends the queue and updates the rear pointer. The `isempty` operation checks if the linked list nodes referenced by `front` and `rear` are equal. The equality here is the component-wise equality on `Lnklist` pairs, with natural numbers compared in the usual fashion and references compared by identity. (Two references are equal if and only if they denote the same reference allocated in a particular use of `letref`). The `getfront` operation accesses the value in the second node of the linked-list (skipping over the dummy header), and the `delete` operation deletes the first node making the second node the dummy header.

The significant feature of this program is that when a linked list is extended by modifying `rear`↑.2, the modification is visible via other access paths to this reference originating at `front`. It is this information sharing that permits `Q.insert(3)` to affect the later evaluation of `Q.getfront` even though no values are *passed* between these program points. In contrast, in a functional language, whenever a value is computed, it needs to be explicitly passed to all the program points that need access to the value. In the case of the queue example, this means that whenever the last element of the list is modified, the modified element needs to be passed to all the previous elements so that they can be updated to refer to the new element. This involves $O(n)$ computation for the `insert` operation. In contrast, our implementation involves constant time computation for all its operations.

Constant time queue operations can be implemented in pure functional languages using a clever representation that keeps two lists instead of one. But, the principles we are referring to work for all kinds of data structure updates, such as graph traversal, graph reduction, graph unification, etc., where such clever tricks are not adequate to solve the problems efficiently.

To give a flavor of how the queue implementation discussed above may be used, we show a sample program for the level-order traversal of a binary tree in Figure 2.5.

```

datatype BTree = Niltree | Mktree of (nat × BTree × BTree)
breadthfirst: Btree → Obs T → Obs T
  = λtree.λc.
    Qcreate(Niltree)(λQ: QADT.
      Q.insert(tree)
      letrec bfirst: Obs T
        = Q.isempty(c)(Q.getfront(λt: Btree.
          Q.delete
          case t of Niltree ⇒ bfirst,
                  Mktree(n, t1, t2) ⇒ Q.insert t1
                                     (Q.insert t2
                                      (visitnode n
                                       bfirst))))
      in bfirst)

```

Figure 2.5: Level-order traversal of binary trees.

2.8 Example: Unification

In our final example, we implement unification by an algorithm that performs *shared updates* on a data structure. Unification [70] is a significant problem that finds applications in diverse areas including type inference, implementation of Prolog and theorem provers, and natural semantics. This example convincingly illustrates the expressive power and usability of ILC.

Figure 2.6 contains an ILC program that computes the most general common instance of two terms s and t . A *term* is either a variable or a pair denoting the application of a function symbol to a list of subterms. A variable is represented by a *reference*. The reference may contain either a term (if the variable is already bound), or the special value `Unbound`. This representation of terms illustrates the notion of *shared dynamic data* mentioned in the introduction; a function accepting a term has indirect access to all the references embedded in the term.

We assume the existence of a global reference called `sigma` that accumulates the list of references bound during an attempt at unification. If the unification is successful, this yields the most general unifier, while the representations of the terms s and t correspond to the most general common instance. On failure, this list is used to reset the values of the references to `Unbound`.

```

datatype term = Var of Ref var
              | Apply of (symbol × List term)

and var = Unbound | Bound of term

unify: term × term × Obs T × Obs T → Obs T
= λ(s, t, sc, fc). unify-aux(s, t, sc, undo(fc))

unify-aux: term × term × Obs T × Obs T → Obs T
= λ(s, t, sc, fc).
  case (s, t) of
    (Var(v), Var(w)) ⇒
      if (v = w) then sc else unify-vars(v, w, sc, fc)
  | (Var(v), Apply(−, −)) ⇒ bind(v, t, sc, fc)
  | (Apply(−, −), Var(v)) ⇒ bind(v, s, sc, fc)
  | (Apply(f, ls), Apply(g, lt)) ⇒
      if (f = g) then unify-lists(ls, lt, sc, fc) else fc

unify-lists: List term × List term × Obs T × Obs T → Obs T
= λ(ls, lt, sc, fc). case (ls, lt) of
  ([], []) ⇒ sc
  | (s :: ls2, t :: lt2) ⇒
      unify-aux(s, t, unify-lists(ls2, lt2, sc, fc), fc)
  | (−, −) ⇒ fc

unify-vars: var × var × Obs T × Obs T → Obs T
= λ(v, w, sc, fc).
  case (v ↑, w ↑) of
    (Unbound, Unbound) ⇒ bind(v, Var(w), sc, fc)
  | (Unbound, Bound(t)) ⇒ unify-aux(Var(v), t, sc, fc)
  | (Bound(t), −) ⇒ unify-aux(t, Var(w), sc, fc)

bind: Ref var × term × Obs T × Obs T → Obs T
= λ(v, t, sc, fc). case v ↑ of
  Unbound ⇒ occurs(v, t, fc, (v := Bound(t);
                                sigma := v :: sigma ↑;
                                sc))
  | Bound(s) ⇒ unify-aux(s, t, sc, fc)

```

Figure 2.6: Unification of first-order terms.

```

undo: Obs T → Obs T
= λfc. case sigma↑ of
    [] ⇒ fc
  | v :: lv ⇒ v := Unbound;
              sigma := lv;
              undo(fc)

occurs: Ref var × term × Obs T × Obs T → Obs T
= ...

```

Figure 2.6: Unification of first-order terms (continued).

The function `unify` attempts to compute the most general common instance of two terms `s` and `t`. If the unification is successful, it instantiates both `s` and `t` to their most general common instance (by updating the references embedded in them), and evaluates the success continuation `sc`. If the unification is unsuccessful, it leaves the terms unchanged and evaluates the failure continuation `fc`. Internally, it uses the auxiliary function `unify-aux`, which updates the terms in both cases. By providing the failure continuation (`undo fc`) to this function, terms are restored to their original values upon failure. The function `unify-lists` unifies two lists of terms (`ls`, `lt`), `unify-vars` unifies two variables (`v`, `w`), `bind` unifies a variable `v` with a term `t`, `occurs` checks whether a variable `v` occurs free in a term `t`, and `undo` resets the values of variables that have been bound during a failed attempt at unification. The definition of `occurs` is straightforward and has been omitted.

The significant aspect of this program is that when an unbound variable is unified with a term that does not contain any free occurrence of the variable, unification succeeds by *assigning* the term to the reference that represents the variable (see function `bind`). This modification is visible via other access paths to the reference. It is this information sharing that affects the unification of subsequent subterms, even though no values are *passed* between these program points. In contrast, in a pure functional language, every computed value needs to be passed explicitly to all program points that need it. In the unification example, this means that whenever a variable is modified, the modified value needs to be passed to all other subterms that are yet to be unified, an expensive proposition indeed.

Chapter 3

Strong Normalization for ILC

3.1 Introduction

In this chapter we prove that the reduction system of ILC is strongly normalizing for well-typed terms. The proof inducts over type derivations, and follows the structure of Girard's proof of strong normalization for the typed lambda calculus.

The proof uses an induction hypothesis that is stronger than strong normalization. The hypothesis is based on an abstract notion called *reducibility*. This notion has the property that if a term is reducible, then it is strongly normalizable. Thus strong normalization is proved by proving that all terms are reducible. However, reducibility satisfies three other conditions which are included in the induction hypothesis and which permit the proof to go through. The notion of reducibility was originally due to Tait [82]. We use Girard's formalization [26], including the technical improvements he made in the proof, though we have modified the details considerably.

The proof may be summarized as follows:

1. We define three families of relations on terms of ILC: τ -reducibility, θ -reducibility, and ω -reducibility. These relations are defined by a straightforward induction on the type structure of ILC. We shall view these relations as sets of terms of τ , θ , and ω types respectively. We call these sets *reducibility sets*.
2. We define four important properties ((**CR 1**), (**CR 2**), (**CR 3**), and (**CR 4**)) of the reducibility sets. The most important property, (**CR 1**), says that every term of every reducibility set is strongly normalizable.

3. We prove that members of the reducibility sets satisfy the four properties **(CR 1)** through **(CR 4)** . The proof is by induction on the structure of types.
4. The reducibility sets are defined to be closed under pair projection $(e.1, e.2)$, function application $(e_1 e_2)$, and reference creation (**letref** $v^* := e_1$ **in** e_2). We prove that they are also closed under pairing $((e_1, e_2))$, lambda abstraction $(\lambda x.e)$, dereference (**get** $x \leftarrow e_1$ **in** e_2), and assignment $(e_1 := e_2 ; e_3)$.
5. We use the above to prove that all well-typed terms are reducible. That is, we prove that all well-typed terms belong to one of the reducibility sets.
6. We know, from the property **(CR 1)** , that every term of every reducibility set is strongly normalizable. We conclude that all well-typed terms are strongly normalizable.

Our definition of reducibility modifies the Tait-Girard definition to include observation types. The complexities introduced by observation types make our proof longer and more complex than the proof for the simply typed lambda calculus. The proof presented in this chapter may be generalized to higher order theories; Gallier [22] presents an excellent discussion of various approaches to such a generalization.

3.2 Strong Normalization

This chapter is devoted to proving the following theorem.

Theorem 11 (Strong Normalization) *Let $\Gamma \vdash t : \omega$ be a recursion-free term. Then there is no infinite reduction sequence $t \longrightarrow t_1 \longrightarrow t_2 \longrightarrow \dots$ of well typed ILC terms.*

Notation:

Let x_1, \dots, x_n be variables and let e_1, \dots, e_n be terms such that x_1, \dots, x_{j-1} are free in e_j . We abbreviate the term $(t[e_n/x_n][e_{n-1}/x_{n-1}] \dots [e_i/x_i])$ to $(t[\bar{e}/\bar{x}]_i^n)$.

Let t be a strongly normalizable term. Then, by definition, every reduction sequence $t \longrightarrow t_1 \longrightarrow t_2 \longrightarrow \dots$ is finite. We write $\nu(t)$ to denote the length of the longest reduction sequence beginning with t . $\nu(t)$ is finite, and if $t \longrightarrow t'$, then $\nu(t) > \nu(t')$. We shall use these properties in the following proofs to induct over the number $\nu(t)$.

The following lemma is used in several places in the proof, and hence is presented separately.

Lemma 12 *Let $C[t[\bar{e}/\bar{x}]_1^n]$ be a term where $t[\bar{e}/\bar{x}]_1^n$ is embedded in context $C[_]$. If $C[t[\bar{e}/\bar{x}]_1^n]$ is strongly normalizable, then so is t .*

Proof: Let t be a term and let $C[t[\bar{e}/\bar{x}]_1^n]$ be a strongly normalizable term. For every reduction sequence $t \longrightarrow t_1 \longrightarrow t_2 \longrightarrow \dots$, we can construct a reduction sequence $C[t[\bar{e}/\bar{x}]_1^n] \longrightarrow C[t_1[\bar{e}/\bar{x}]_1^n] \longrightarrow C[t_2[\bar{e}/\bar{x}]_1^n] \longrightarrow \dots$. This sequence has to be finite as $C[t[\bar{e}/\bar{x}]_1^n]$ is strongly normalizable. Thus all reduction sequences from t are finite and so t is strongly normalizable.

3.3 Reducibility

We define a family of sets \mathbf{RED}_T^τ (reducible terms of τ -type T) by induction on the τ -type T .

1. $(\Gamma \vdash t : \beta)$, for atomic type β , is τ -reducible if t is strongly normalizable.
2. $(\Gamma \vdash t : \tau_1 \times \tau_2)$ is τ -reducible if $(\Gamma \vdash t.1 : \tau_1)$ and $(\Gamma \vdash t.2 : \tau_2)$ are τ -reducible.
3. $(\Gamma \vdash t : \tau_1 \rightarrow \tau_2)$ is τ -reducible if for all τ -reducible $(\Gamma \vdash e : \tau_1)$, $(\Gamma \vdash t(e) : \tau_2)$ is τ -reducible.

We define a family of sets \mathbf{RED}_T^θ (reducible terms of θ -type T) by induction on the θ -type T .

1. $(\Gamma \vdash t : \tau)$ is θ -reducible if t is strongly normalizable.
2. $(\Gamma \vdash t : \mathbf{Ref} \theta_1)$ is θ -reducible if t is strongly normalizable.
3. $(\Gamma \vdash t : \theta_1 \times \theta_2)$ is θ -reducible if $(\Gamma \vdash t.1 : \theta_1)$ and $(\Gamma \vdash t.2 : \theta_2)$ are θ -reducible.
4. $(\Gamma \vdash t : \theta_1 \rightarrow \theta_2)$ is θ -reducible if for all θ -reducible $(\Gamma \vdash e : \theta_1)$, $(\Gamma \vdash t(e) : \theta_2)$ is θ -reducible.

We define a family of sets \mathbf{RED}_T^ω (reducible terms of ω -type T) by induction on the ω -type T .

1. $(\Gamma \vdash t : \theta)$ is ω -reducible if t is strongly normalizable.
2. $(\Gamma \vdash t : \omega_1 \times \omega_2)$ is ω -reducible if $(\Gamma \vdash t.1 : \omega_1)$ and $(\Gamma \vdash t.2 : \omega_2)$ are ω -reducible.
3. $(\Gamma \vdash t : \omega_1 \rightarrow \omega_2)$ is ω -reducible if for all ω -reducible $(\Gamma \vdash e : \omega_1)$, $(\Gamma \vdash t(e) : \omega_2)$ is ω -reducible.
4. $(\Gamma \vdash t : \mathbf{Obs} \tau_1)$ is ω -reducible if

- (a) Γ has only τ types and $(\Gamma \vdash \mathbf{app}(t) : \tau_1)$ is τ -reducible; or
- (b) $\Gamma \equiv (\Gamma', v^* : \mathbf{Ref} \theta_1)$ and, for all θ -reducible $(\Gamma', v^* : \mathbf{Ref} \theta_1 \vdash e : \theta_1)$, $(\Gamma' \vdash (\mathbf{letref} v^* := e \mathbf{in} t) : \mathbf{Obs} \tau_1)$ is ω -reducible; or
- (c) $\Gamma \equiv (\Gamma', v : \omega)$ and for all ω -reducible $(\Gamma \vdash e : \omega)$, $(\Gamma' \vdash t[e/v] : \mathbf{Obs} \tau_1)$ is ω -reducible.

3.4 Neutrality

Definition 13 (Neutrality) *A term is called neutral if it is not of the form*

$$x \quad k \quad (\lambda x.e) \quad \langle e_1, e_2 \rangle \quad (\mathbf{get} x \Leftarrow l \mathbf{in} t)$$

In other words, neutral terms are those of the form:

$$f(e) \quad e.1 \quad e.2 \quad (\mathbf{letref} v := e \mathbf{in} t) \quad (l := e; t)$$

Intuitively, neutral terms are those which may have outermost redexes.

3.5 Properties of Reducibility

(CR 1) If $(\Gamma \vdash t : T)$ is reducible, then t is strongly normalizable.

(CR 2) If $(\Gamma \vdash t : T)$ is reducible, and $t \longrightarrow t'$, then $(\Gamma \vdash t' : T)$ is reducible.

(CR 3) Let $\Gamma = (x_1 : T_1, \dots, x_n : T_n)$. If $(\Gamma \vdash t : T)$ is a term such that t is neutral, and for all reducible $e_1 : T_1, \dots, e_n : T_n$, whenever we convert a redex of $t[\bar{e}/\bar{x}]_1^n$ we obtain a reducible term $(\vdash t' : T)$, then $(\Gamma \vdash t : T)$ is reducible.

(CR 4) Every well-typed variable $(\Gamma \vdash x : T)$ is reducible.

Lemma 14 \mathbf{RED}_T^τ satisfies **(CR 1-4)**.

Proof: Let $(\Gamma \vdash t : T)$ be a term. We prove the lemma by induction on the type T .

Atomic types β

A term of atomic type is reducible iff it is strongly normalizable. Thus \mathbf{RED}_β^τ is the set of strongly normalizable terms of type β . We show that this set satisfies **(CR 1-4)**.

(CR 1) is a tautology.

(CR 2) If t is strongly normalizable, then so is every term t' to which it reduces.

(CR 3) Let $e_1:T_1, \dots, e_n:T_n$ be reducible terms. Every reduction path leaving $t[\bar{e}/\bar{x}]_1^n$ must pass through one of the terms t' . Since these terms are strongly normalizable, the reduction paths must be finite. Hence $t[\bar{e}/\bar{x}]_1^n$ is strongly normalizable, and so t is strongly normalizable (by lemma 12).

(CR 4) A variable is in normal form and hence is strongly normalizable.

Product type

A term t of product type $(\tau_1 \times \tau_2)$ is reducible iff its projections $t.1$ and $t.2$ are reducible.

(CR 1) Suppose that $(\Gamma \vdash t : \tau_1 \times \tau_2)$ is τ -reducible. Then, by definition, $(\Gamma \vdash t.1 : \tau_1)$ is reducible, and by induction hypothesis **(CR 1)** for τ_1 , $t.1$ is strongly normalizable. Thus, by lemma 12, t is strongly normalizable.

(CR 2) Suppose that $(\Gamma \vdash t : \tau_1 \times \tau_2)$ is τ -reducible. If $t \longrightarrow t'$, then $t.1 \longrightarrow t'.1$ and $t.2 \longrightarrow t'.2$. As t is τ -reducible by hypothesis, so are $t.1$ and $t.2$ (by definition of reducibility). The induction hypothesis **(CR 2)** for τ_1 and τ_2 says that $t'.1$ and $t'.2$ are reducible, and so t' is reducible by definition.

(CR 3) Let $\Gamma = (x_1:\tau'_1, \dots, x_n:\tau'_n)$. Let $(\Gamma \vdash t : \tau_1 \times \tau_2)$ be a term such that t is neutral. Further, for all reducible $e_1:\tau'_1, \dots, e_n:\tau'_n[\bar{e}/\bar{x}]_1^{n-1}$, suppose that all the t' one step from $t[\bar{e}/\bar{x}]_1^n$ are reducible. Since t is neutral, it is not a variable or a pair, and hence $t[\bar{e}/\bar{x}]_1^n$ is not a pair. Thus the only redices in $t[\bar{e}/\bar{x}]_1^n.1$ are inside $t[\bar{e}/\bar{x}]_1^n$. Applying such a reduction, we get some $t'.1$ which is reducible since t' is (by assumption). But as $t.1$ is neutral, and as all the terms one step from $t.1[\bar{e}/\bar{x}]_1^n$ are reducible, the induction hypothesis **(CR 3)** for τ_1 ensures that $t.1$ is reducible. Similarly, $t.2$ is reducible, and so t is reducible (by definition).

(CR 4) Let $(\Gamma \vdash x : \tau_1 \times \tau_2)$ be a well-typed variable. Now $x.1$ is neutral. Let $\Gamma = (x_1:\tau'_1, \dots, x_n:\tau'_n)$. Let $e_1:\tau'_1, \dots, e_n:\tau'_n[\bar{e}/\bar{x}]_1^{n-1}$. Then $x_i.1[\bar{e}/\bar{x}]_1^n = (e_i[\bar{e}/\bar{x}]_{i+1}^n.1)$. Since

all the e_i 's are reducible, $(e_i[\bar{e}/\bar{x}]_{i+1}^n)$ is reducible and hence so is $e_i[\bar{e}/\bar{x}]_{i+1}^n.1$ (by definition). Thus, by induction hypothesis **(CR 3)** on τ_1 , $x.1$ is reducible. Likewise, $\Gamma \vdash x.2 : \tau_2$ is reducible, and hence so is $\Gamma \vdash x : \tau_1 \times \tau_2$ (by definition).

Function type

A term of function type is reducible iff all its applications to reducible terms are reducible.

(CR 1) Suppose that $(\Gamma \vdash t : \tau_1 \rightarrow \tau_2)$ is τ -reducible. Then, by definition, $(\Gamma \vdash t(e) : \tau_2)$ is reducible for all reducible $(\Gamma \vdash e : \tau_1)$. Now, by induction hypothesis **(CR 4)** for τ_1 , we have that $(\Gamma, x : \tau_1 \vdash x : \tau_1)$ is reducible. Thus $(\Gamma, x : \tau_1 \vdash t(x) : \tau_2)$ is reducible and so $t(x)$ is strongly normalizable. Thus, by lemma 12, t is strongly normalizable.

(CR 2) Suppose that $(\Gamma \vdash t : \tau_1 \rightarrow \tau_2)$ is τ -reducible. Then, by definition, $(\Gamma \vdash t(e) : \tau_2)$ is τ -reducible for all τ -reducible $(\Gamma \vdash e : \tau_1)$. If $t \rightarrow t'$, then $t(e) \rightarrow t'(e)$. The induction hypothesis **(CR 2)** for τ_2 gives that $t'(e)$ is reducible. So t' is reducible (by definition).

(CR 3) Let $\Gamma = (x_1 : T_1, \dots, x_n : T_n)$. Let $(\Gamma \vdash t : \tau_1 \rightarrow \tau_2)$ be a term such that t is neutral. Further, for all reducible $e_1 : T_1, \dots, e_n : T_n$, suppose that all the t' one step from $t[\bar{e}/\bar{x}]_1^n$ are reducible. Let $(\vdash u : \tau_1)$ be a τ -reducible term. By induction hypothesis **(CR 1)** for τ_1 , we know that u is strongly normalizable, so we can reason by induction on $\nu(u)$.

In one step, $t[\bar{e}/\bar{x}]_1^n(u)$ converts to

- $t'(u)$, with t' one step from $t[\bar{e}/\bar{x}]_1^n$; but t' is reducible (by assumption), and hence $t'(u)$ is reducible (by definition).
- $t[\bar{e}/\bar{x}]_1^n(u')$ with u' one step from u . u' is reducible by induction hypothesis **(CR 2)** for τ_1 , and $\nu(u') < \nu(u)$; so the induction hypothesis for u' tells us that $t[\bar{e}/\bar{x}]_1^n(u')$ is reducible.
- There is no other possibility, for $t[\bar{e}/\bar{x}]_1^n(u)$ cannot itself be a redex. (Since t is neutral, it is not a variable or a lambda abstraction, and so $t[\bar{e}/\bar{x}]_1^n$ is not a lambda abstraction).

In every case we have seen that the neutral term $t[\bar{e}/\bar{x}]_1^n(u)$ converts into reducible terms only. The induction hypothesis **(CR 3)** for τ_2 allows us to conclude that $t(u)$ is reducible, and so t is reducible.

(CR 4) Let $(\Gamma \vdash x : \tau_1 \rightarrow \tau_2)$ be a well-typed variable. By definition, it is reducible if for all reducible $(\Gamma \vdash e : \tau_1)$, $(\Gamma \vdash x(e) : \tau_2)$ is reducible. An identical argument to **(CR 3)** above proves that x is reducible.

This concludes the proof of lemma 14.

Lemma 15 \mathbf{RED}_T^θ satisfies **(CR 1-4)**.

Proof: The proof of lemma 15 parallels the proof of lemma 14.

Lemma 16 \mathbf{RED}_T^ω satisfies **(CR 1-4)**.

Proof: We reason by induction on the type T .

Atomic ω -types θ

The proof is identical to the proof for atomic types in lemma 14.

Product type $\omega_1 \times \omega_2$

The proof is identical to the proof for product types in lemma 14.

Function type $\omega_1 \rightarrow \omega_2$

The proof is identical to the proof for function types in lemma 14.

Observer type $\mathbf{Obs} \tau$

A term of observer type $\mathbf{Obs} \tau$ is ω -reducible iff it is ω -reducible under all possible assignments of θ -reducible terms to its free references. This is defined by inducting over the length of the type assumption Γ . Hence our proofs will also involve inducting over the length of Γ .

1. Γ has only τ types.

By definition, $(\Gamma \vdash t : \mathbf{Obs} \tau)$ is ω -reducible iff $(\Gamma \vdash t : \tau)$ is τ -reducible. But \mathbf{RED}_τ^τ satisfies **(CR 1-4)** (by lemma 14). Hence, so does $\mathbf{RED}_{\mathbf{Obs} \tau}^\omega$.

2. Γ is of the form $(\Gamma, v^* : \mathbf{Ref} \theta)$.

The following proofs for **(CR 1-4)** closely mimic the corresponding proofs for function types.

(CR 1) If $(\Gamma, v^* : \mathbf{Ref} \theta \vdash t : \mathbf{Obs} \tau)$ is ω -reducible, then t is strongly normalizable.

Proof: Let $(\Gamma, v^* : \mathbf{Ref} \theta \vdash t : \mathbf{Obs} \tau)$ be ω -reducible. Let x be a variable of type θ . Then, $(\Gamma, v^* : \mathbf{Ref} \theta \vdash x : \theta)$ is θ -reducible, since $\mathbf{RED}_\theta^\theta$ satisfies **(CR 4)** by lemma 15. Hence $(\Gamma \vdash (\mathbf{letref} v^* := x \mathbf{in} t) : \mathbf{Obs} \tau)$ is ω -reducible (by definition of reducibility). By induction hypothesis **(CR 1)** for Γ , we have that $(\mathbf{letref} v^* := x \mathbf{in} t)$ is strongly normalizable, and hence so is t (by lemma 12).

(CR 2) If $(\Gamma, v^* : \mathbf{Ref} \theta \vdash t : \mathbf{Obs} \tau)$ is ω -reducible, and $t \longrightarrow t'$, then $(\Gamma, v^* : \mathbf{Ref} \theta \vdash t' : \mathbf{Obs} \tau)$ is ω -reducible.

Proof: Let $(\Gamma, v^* : \mathbf{Ref} \theta \vdash t : \mathbf{Obs} \tau)$ be ω -reducible, and let $t \longrightarrow t'$. Let $(\Gamma, v^* : \mathbf{Ref} \theta \vdash e : \theta)$ be θ -reducible. Then, $(\Gamma \vdash (\mathbf{letref} v^* := e \mathbf{in} t) : \mathbf{Obs} \tau)$ is ω -reducible (by definition of reducibility). Since $t \longrightarrow t'$, $(\mathbf{letref} v^* := e \mathbf{in} t) \longrightarrow (\mathbf{letref} v^* := e \mathbf{in} t')$. The induction hypothesis **(CR 2)** on Γ gives that $(\Gamma \vdash \mathbf{letref} v^* := e \mathbf{in} t' : \mathbf{Obs} \tau)$ is ω -reducible. Thus, $(\Gamma, v^* : \mathbf{Ref} \theta \vdash t' : \mathbf{Obs} \tau)$ is ω -reducible (by definition of reducibility).

(CR 3) If t is neutral, and whenever we convert a redex of $t[\bar{e}/\bar{x}]_1^n$ we obtain a ω -reducible term $(\Gamma, v^* : \mathbf{Ref} \theta \vdash t' : \mathbf{Obs} \tau)$, then $(\Gamma, v^* : \mathbf{Ref} \theta \vdash t : \mathbf{Obs} \tau)$ is ω -reducible.

Proof: Let t be neutral. Assume that whenever we convert a redex of $t[\bar{e}/\bar{x}]_1^n$ we obtain a ω -reducible term $(\Gamma, v^* : \mathbf{Ref} \theta \vdash t' : \mathbf{Obs} \tau)$. Let $(\Gamma, v^* : \mathbf{Ref} \theta \vdash e : \theta)$ be θ -reducible; we want to show that $(\Gamma \vdash \mathbf{letref} v^* := e \mathbf{in} t : \mathbf{Obs} \tau)$ is ω -reducible. By lemma 15 **(CR 1)**, we know that e is strongly normalizable; so we can reason by induction on $\nu(e)$.

In one step, $(\Gamma \vdash \mathbf{letref} v^* := e \mathbf{in} t[\bar{e}/\bar{x}]_1^n : \mathbf{Obs} \tau)$ converts to:

- $(\Gamma \vdash \mathbf{letref} v^* := e \mathbf{in} t' : \mathbf{Obs} \tau)$ with t' one step from $t[\bar{e}/\bar{x}]_1^n$; but $(\Gamma, v^* : \mathbf{Ref} \theta \vdash t' : \mathbf{Obs} \tau)$ is ω -reducible (by assumption and $t[\bar{e}/\bar{x}]_1^n \longrightarrow t'$, and hence $(\Gamma \vdash \mathbf{letref} v^* := e \mathbf{in} t' : \mathbf{Obs} \tau)$ is ω -reducible (by definition of reducibility).
- $(\Gamma \vdash \mathbf{letref} v^* := e' \mathbf{in} t[\bar{e}/\bar{x}]_1^n : \mathbf{Obs} \tau)$ with e' one step from e . $(\Gamma, v^* : \mathbf{Ref} \theta \vdash e' : \theta)$ is θ -reducible by assumption, and hence so is $(\Gamma, v^* : \mathbf{Ref} \theta \vdash e' : \theta)$ (by lemma 15 **(CR 2)**). Since $\nu(e') < \nu(e)$, the induction hypothesis for e' tells us that $(\Gamma \vdash \mathbf{letref} v^* := e' \mathbf{in} t[\bar{e}/\bar{x}]_1^n : \mathbf{Obs} \tau)$ is ω -reducible.

- There is no other possibility, for $(\mathbf{letref} \ v^* := e' \ \mathbf{in} \ t[\bar{e}/\bar{x}]_1^n)$ cannot itself be a redex. (Since t is neutral and is not a variable, $t[\bar{e}/\bar{x}]_1^n$ is neutral and hence is not of the form $k, (\lambda x.e), \langle e_1, e_2 \rangle$ or $\mathbf{get} \ x \Leftarrow l \ \mathbf{in} \ s$).

In every case we have seen that the neutral term $\mathbf{letref} \ v^* := e \ \mathbf{in} \ t[\bar{e}/\bar{x}]_1^n$ converts into ω -reducible terms only. The induction hypothesis **(CR 3)** for Γ allows us to conclude that $\mathbf{letref} \ v^* := e \ \mathbf{in} \ t$ is reducible, and so t is reducible.

(CR 4) Let $(\Gamma, v^* : \mathbf{Ref} \ \theta \vdash x : \mathbf{Obs} \ \tau)$ be a well-typed variable. Let $(\Gamma, v^* : \mathbf{Ref} \ \theta \vdash e : \theta)$ be θ -reducible; we want to show that $(\Gamma \vdash \mathbf{letref} \ v^* := e \ \mathbf{in} \ x : \mathbf{Obs} \ \tau)$ is ω -reducible. This is proved by an argument similar to the proof of **(CR 3)** .

3. Γ is of the form $(\Gamma, v : \omega)$.

(CR 1) Let $(\Gamma, v : \omega_1 \vdash t : \mathbf{Obs} \ \tau)$ be an ω -reducible term. Let $(\Gamma \vdash e : \omega_1)$ also be ω -reducible. Then, $(\Gamma \vdash t[e/v] : \mathbf{Obs} \ \tau)$ is ω -reducible (by definition of reducibility). By induction hypothesis **(CR 1)** on Γ , $t[e/v]$ is strongly normalizable, and hence so is t (by lemma 12).

(CR 2) Let $(\Gamma, v : \omega_1 \vdash t : \mathbf{Obs} \ \tau)$ be an ω -reducible term, and let $t \longrightarrow t'$. Let $(\Gamma \vdash e : \omega_1)$ also be ω -reducible. Then, $(\Gamma \vdash t[e/v] : \mathbf{Obs} \ \tau)$ is ω -reducible (by definition of reducibility). Since $t \longrightarrow t'$, $t[e/v] \longrightarrow t'[e/v]$. By induction hypothesis **(CR 2)** on Γ , $(\Gamma \vdash t'[e/v] : \mathbf{Obs} \ \tau)$ is ω -reducible, and hence so is $(\Gamma, v : \omega_1 \vdash t' : \mathbf{Obs} \ \tau)$.

(CR 3) Let $\Gamma = (x_1 : T_1, \dots, x_n : T_n)$. Let $(\Gamma, v : \omega_1 \vdash t : \mathbf{Obs} \ \tau)$ be a term such that t is neutral. Further, for all reducible $e_1 : T_1, \dots, e_n : T_n, e : \omega_1$, suppose that all the t' one step from $t[e/v][\bar{e}/\bar{x}]_1^n$ are reducible. Let $(\Gamma \vdash e : \omega_1)$ be an ω -reducible term. we want to show that $(\Gamma \vdash t[e/v] : \mathbf{Obs} \ \tau)$ is ω -reducible. We know (by assumption) that all the t' one step from $t[e/v][\bar{e}/\bar{x}]_1^n$ are reducible. Further, since t is neutral and is not a variable, $t[e/v][\bar{e}/\bar{x}]_1^n$ is neutral. Thus, by induction hypothesis **(CR 3)** on Γ , $(\Gamma \vdash t[e/v] : \mathbf{Obs} \ \tau)$ is ω -reducible.

(CR 4) Let $(\Gamma, v : \omega_1 \vdash x : \mathbf{Obs} \ \tau)$ be a well-typed term. By definition, this is ω -reducible iff for all ω -reducible $(\Gamma \vdash e : \omega_1)$, $(\Gamma \vdash x[e/v] : \mathbf{Obs} \ \tau)$ is ω -reducible. If v is x , then this is immediate. Otherwise, $x[e/v] = x$ and $(\Gamma \vdash x : \mathbf{Obs} \ \tau)$ is ω -reducible by induction hypothesis **(CR 4)** on Γ .

3.6 Admissibility

3.6.1 Pairing

Lemma 17 *If $(\Gamma \vdash e_1 : \omega_1)$ and $(\Gamma \vdash e_2 : \omega_2)$ are reducible, then so is $(\Gamma \vdash \langle e_1, e_2 \rangle : \omega_1 \times \omega_2)$.*

Proof: See Girard [26], Section 6.3.1.

3.6.2 Abstraction

Lemma 18 *If $(\Gamma \vdash e_2[e_1/x] : \omega_2)$ is reducible for all reducible $(\Gamma \vdash e_1 : \omega_1)$, then so is $(\Gamma \vdash \lambda x. e_2 : \omega_1 \rightarrow \omega_2)$.*

Proof: See Girard [26], Section 6.3.2.

3.6.3 Dereference

Lemma 19 *If $(\Gamma \vdash l : \mathbf{Ref} \theta)$ is θ -reducible, and if $(\Gamma \vdash t[e/x] : \mathbf{Obs} \tau)$ is ω -reducible for all θ -reducible $(\Gamma \vdash e : \theta)$, then $(\Gamma \vdash \mathbf{get} x \Leftarrow l \mathbf{in} t : \mathbf{Obs} \tau)$ is ω -reducible.*

Proof: Assume that $(\Gamma \vdash l : \mathbf{Ref} \theta)$ is θ -reducible, and that $(\Gamma \vdash t[e/x] : \mathbf{Obs} \tau)$ is ω -reducible for all θ -reducible $(\Gamma \vdash e : \theta)$. We prove that $(\Gamma \vdash \mathbf{get} x \Leftarrow l \mathbf{in} t : \mathbf{Obs} \tau)$ is ω -reducible. We reason by induction on the number of free reference variables in Γ .

1. Γ has only τ types.

This is not possible since the language has no reference constants and since l is a reference valued expression under Γ .

2. $\Gamma \equiv (\Gamma', v^* : \mathbf{Ref} \theta)$.

By definition of reducibility, $(\Gamma \vdash \mathbf{get} x \Leftarrow l \mathbf{in} t : \mathbf{Obs} \tau)$ is ω -reducible iff $(\Gamma' \vdash \mathbf{letref} v^* := e \mathbf{in} \mathbf{get} x \Leftarrow l \mathbf{in} t : \mathbf{Obs} \tau)$ is ω -reducible for all θ -reducible $(\Gamma' \vdash e : \theta)$.

Let $(\Gamma' \vdash e : \theta)$ be θ -reducible. Earlier assumptions give us that $(\Gamma \vdash l : \mathbf{Ref} \theta)$ is θ -reducible, and $(\Gamma \vdash t[e/x] : \mathbf{Obs} \tau)$ is ω -reducible. **(CR 1)** permits us to reason by induction on $\nu(e) + \nu(l) + \nu(t)$ to show that $(\Gamma' \vdash \mathbf{letref} v^* := e \mathbf{in} \mathbf{get} x \Leftarrow l \mathbf{in} t : \mathbf{Obs} \tau)$ is ω -reducible. This term converts to:

- $(\Gamma' \vdash \mathbf{letref} v^* := e \mathbf{in} t[e/x] : \mathbf{Obs} \tau)$, which is ω -reducible by definition of reducibility (using our assumptions).

- $(\Gamma' \vdash (\text{get } x' \Leftarrow l \text{ in } \text{letref } v^* := e \text{ in } t[x'/x]) : \text{Obs } \tau)$, if l is $w^* \neq v^*$ and. $x' \notin V(e)$. This follows from the induction hypothesis on Γ . However, for the induction hypothesis to be applicable, $(\Gamma \vdash (\text{letref } v^* := e \text{ in } t[x'/x])[e'/x'] : \omega)$ should be ω -reducible for all θ -reducible $(\Gamma \vdash e' : \theta)$. But $(\text{letref } v^* := e \text{ in } t[x'/x])[e'/x'] \equiv \text{letref } v^* := e \text{ in } t[e'/x]$, which is ω -reducible by definition of reducibility and our first assumption.
- $(\Gamma' \vdash \text{letref } v^* := e' \text{ in } \text{get } x \Leftarrow l \text{ in } t : \text{Obs } \tau)$ with e' one step from e . $(\Gamma \vdash e' : \theta)$ is θ -reducible by **(CR 2)**, and we have $\nu(e') < \nu(e)$; so the induction hypothesis tells us that this term is ω -reducible.
- $(\Gamma' \vdash \text{letref } v^* := e \text{ in } \text{get } x \Leftarrow l' \text{ in } t : \text{Obs } \tau)$ with l' one step from l . This term is ω -reducible for similar reasons.
- $(\Gamma' \vdash \text{letref } v^* := e \text{ in } \text{get } x \Leftarrow l \text{ in } t' : \text{Obs } \tau)$ with t' one step from t . This term is ω -reducible for similar reasons.

In every case, the neutral term $(\Gamma' \vdash \text{letref } v^* := e \text{ in } \text{get } x \Leftarrow l \text{ in } t : \text{Obs } \tau)$ converts to ω -reducible terms only, and by **(CR 3)** it is ω -reducible. Hence, by definition, so is $(\Gamma \vdash \text{get } x \Leftarrow l \text{ in } t : \text{Obs } \tau)$.

3. $\Gamma \equiv \Gamma', v : \omega$

Let $(\Gamma \vdash e' : \omega)$ be ω -reducible. We have, by assumption, that $(\Gamma \vdash l : \text{Ref } \theta)$ is θ -reducible. Thus, by definition, $(\Gamma' \vdash l[e'/v] : \text{Ref } \theta)$ is θ -reducible. Similarly, by assumption and definition, $(\Gamma \vdash t[e/x][e'/v] : \text{Obs } \tau)$ is ω -reducible for all θ -reducible $(\Gamma \vdash e : \theta)$. Thus, by induction hypothesis on Γ , $(\Gamma' \vdash (\text{get } x \Leftarrow l[e'/v] \text{ in } t[e'/v]) : \text{Obs } \tau)$ is ω -reducible. By definition of reducibility, $(\Gamma \vdash \text{get } x \Leftarrow l \text{ in } t : \text{Obs } \tau)$ is ω -reducible.

3.6.4 Assignment

Let $(\Gamma \vdash t : T)$ be a reducible term. Then by **(CR 1)**, t is strongly normalizable and hence all its reduction sequences are finite. Let $t \longrightarrow t_1 \longrightarrow \dots \longrightarrow t_n$ be an arbitrary reduction sequence ψ of t with t_n in normal form. Let $\text{SD}_\psi(t)$ be the number of times the reduction rule $(\text{letref } v^* : \theta := e \text{ in } \text{get } x \Leftarrow v^* \text{ in } t \longrightarrow \text{letref } v^* : \theta := e \text{ in } t[e/x])$ is used in ψ .

Definition 20 *The state dependence rank $\text{SDR}(\Gamma \vdash t : T)$ of a reducible term $(\Gamma \vdash t : T)$ is defined as follows:*

- If Γ is free of reference variables, then $\text{SDR}(\Gamma \vdash t : T)$ is the minimum of $\text{SD}_{\psi}(t)$ over all reduction sequences ψ that reduce t to normal form.
- If $\Gamma = (\Gamma', v^* : \text{Ref } \theta)$, then $\text{SDR}(\Gamma \vdash t : T) \equiv \text{SDR}(\Gamma', x_{\theta} : \theta \vdash \text{letref } v^* := x_{\theta} \text{ in } t : T)$

Lemma 21 *If $(\Gamma \vdash l : \text{Ref } \theta)$, $(\Gamma \vdash e : \theta)$ and $(\Gamma \vdash t : \text{Obs } \tau)$ are reducible, then so is $(\Gamma \vdash l := e ; t : \text{Obs } \tau)$.*

Proof: Let $(\Gamma \vdash l : \text{Ref } \theta)$, $(\Gamma \vdash e : \theta)$ and $(\Gamma \vdash t : \text{Obs } \tau)$ be reducible, We show that $(\Gamma \vdash l := e ; t : \text{Obs } \tau)$ is reducible by induction on $\nu(l) + \nu(e) + \nu(t) + \text{SDR}(\Gamma \vdash t : \text{Obs } \tau)$. $(\Gamma \vdash l := e ; t : \text{Obs } \tau)$ converts to:

- $(\Gamma \vdash l' := e ; t : \text{Obs } \tau)$, with l' one step from l . $(\Gamma \vdash l' : \text{Ref } \theta)$ is θ -reducible by **(CR 2)**, and we have $\nu(l') < \nu(l)$; so the induction hypothesis tells us that this term is ω -reducible.
- $(\Gamma \vdash l := e' ; t : \text{Obs } \tau)$, with e' one step from e . This term is ω -reducible for similar reasons.
- $(\Gamma \vdash l := e ; t' : \text{Obs } \tau)$, with t' one step from t . This term is ω -reducible for similar reasons.
- $(\Gamma \vdash t : \text{Obs } \tau)$, if l is v^* and if t is of the form $k, \lambda x.e_1$, or $\langle e_1, e_2 \rangle$. This term is reducible by assumption.
- $(\Gamma \vdash (v^* := e ; s[e/x]) : \text{Obs } \tau)$, if l is v^* and if t is of the form $(\text{get } x \Leftarrow v^* \text{ in } s)$. Since v^* has type $\text{Ref } \theta$, we know that for all $a \in \theta$, $\text{SDR}(a) = 0$. Since $(\Gamma \vdash e : \theta)$ is a term, we have that $\text{SDR}(e) = 0$. Thus, $\text{SDR}(s[e/x]) < \text{SDR}(\text{get } x \Leftarrow v^* \text{ in } s)$, and hence by inductive hypothesis, $(\Gamma \vdash (v^* := e ; s[e/x]) : \text{Obs } \tau)$ is ω -reducible.
- $(\Gamma \vdash (\text{get } x \Leftarrow w^* \text{ in } (v^* := e ; s)) : \text{Obs } \tau)$, if l is v^* and if t is of the form $(\text{get } x \Leftarrow w^* \text{ in } s)$, for $w^* \neq v^*$. Now, $\text{SDR}(s) < \text{SDR}(\text{get } x \Leftarrow v^* \text{ in } s)$ (by definition) and hence by inductive hypothesis, $(\Gamma \vdash (v^* := e ; s[e/x]) : \text{Obs } \tau)$ is ω -reducible. By lemma 19 we have that $(\Gamma \vdash (\text{get } x \Leftarrow w^* \text{ in } (v^* := e ; s)) : \text{Obs } \tau)$ is ω -reducible.

In every case, the neutral term $(\Gamma \vdash l := e ; t : \text{Obs } \tau)$ converts to ω -reducible terms only, so by **(CR 3)** it is ω -reducible.

3.7 Reducibility Theorem

Proposition 22 *If $(x_1 : \omega_1, \dots, x_n : \omega_n \vdash t : \omega)$ is a term, and if $(\phi \vdash r_1 : \omega_1), \dots, (\phi \vdash r_n : \omega_n)$ are reducible terms, then $(\phi \vdash t[r_1/x_1, \dots, r_n/x_n] : \omega)$ is reducible.*

Proof: By induction on terms. We write $t[\bar{r}/\bar{x}]$ for $t[r_1/x_1, \dots, r_n/x_n]$.

1. t is one of $x_i, e.1, e.2, \langle e_1, e_2 \rangle, f(e)$ or $\lambda x.e$; the proof is as in Girard [26], Section 6.3.3.
2. t is $(\text{letref } v : \text{Ref } W := e \text{ in } s)$:
By induction hypothesis, $e[l/v, \bar{r}/\bar{x}]$ and $s[l/v, \bar{r}/\bar{x}]$ are reducible for all reducible l in $\text{Ref } W$. and so (by definition) is $(\text{letref } v := e[\bar{r}/\bar{x}] \text{ in } s[\bar{r}/\bar{x}])$; but this term is nothing other than $t[\bar{r}/\bar{x}]$.
3. t is $(\text{get } y : W \Leftarrow l \text{ in } s)$: By induction hypothesis, $l[\bar{r}/\bar{x}]$ and $s[e/y, \bar{r}/\bar{x}]$ are reducible for all reducible e of type W . Admissibility lemma 19 tells us that $(\text{get } y : W \Leftarrow l[\bar{r}/\bar{x}] \text{ in } s[y/y, \bar{r}/\bar{x}])$ is reducible; this term is nothing other than $t[\bar{r}/\bar{x}]$.
4. t is $(l := e ; s)$ of type $\text{Obs } \tau$: By induction hypothesis, $l[\bar{r}/\bar{x}]$, $e[\bar{r}/\bar{x}]$ and $s[\bar{r}/\bar{x}]$ are reducible. Admissibility lemma 21 tells us that $(l[\bar{r}/\bar{x}] := e[\bar{r}/\bar{x}] ; s[\bar{r}/\bar{x}])$ is reducible; this term is nothing other than $t[\bar{r}/\bar{x}]$.

Theorem 23 *All terms are reducible.*

Proof: This follows from proposition 22 by putting $r_i = x_i$, since $(\phi \vdash x_i : \omega_i)$ is reducible by **(CR 1)** .

Corollary 24 *All terms are strongly normalizable.*

Proof: This follows from theorem 23 and **(CR 1)** .

Chapter 4

Observation Type Theory (OTT)

4.1 Introduction

In this chapter, we present a programming logic for the language ILC. We choose constructive type theory as our framework. However, since a full constructive type theory is quite complex both in terms of presentation and understanding, we work with a barebone theory. We start with the simply typed lambda calculus and moderately enrich its type system. We then show that this enriched type system has an interpretation as a constructive logic, and permits us to reason about the equality of typed terms of the enriched lambda calculus. Hence, it is a programming logic for the language. An analogous enrichment and interpretation process can be applied to ILC to yield a programming logic for enriched ILC. We call the resulting theory *observation type theory* (OTT).

4.2 Simply Typed Lambda Calculus

The simply typed lambda calculus is defined as follows:

Terms	$e ::= k \mid x \mid \lambda x.e \mid ee \mid \langle e, e \rangle \mid e.1 \mid e.2$
Types	$\tau ::= \beta \mid \tau \rightarrow \tau \mid \tau \times \tau$
Reductions	$(\lambda x.e_1)e_2 \longrightarrow e_1[e_2/x]$ $\langle e_1, e_2 \rangle.1 \longrightarrow e_1$ $\langle e_1, e_2 \rangle.2 \longrightarrow e_2$

$\frac{\text{\(\times\)-introduction}}{\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2}}$	$\frac{\text{\(\times\)-elimination}}{\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e.1 : \tau_1} \quad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e.2 : \tau_2}}$
$\frac{\text{\(\rightarrow\)-introduction}}{\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash (\lambda x.e) : \tau_1 \rightarrow \tau_2}}$	$\frac{\text{\(\rightarrow\)-elimination}}{\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2}}$

Figure 4.1: Inference rules for the simply typed lambda calculus.

The terms of the language consist of constants (k), variables (x), function abstraction ($\lambda x.e$), function application ($e_1 e_2$), pair formation ($\langle e_1, e_2 \rangle$), and pair projection ($e.1$ and $e.2$). The types of the language are primitive types β , function spaces ($\tau_1 \rightarrow \tau_2$), and cartesian products ($\tau_1 \times \tau_2$). Figure 4.1 presents the inference rules for the type system. The introduction rules prescribe how we may construct members of a type, while the elimination rules prescribe how we may use those members in computations.

4.3 Simple Constructive Type Theory

We can enrich the above type system to obtain a theory that has an interpretation as a logic. The enriched type system is defined as follows:

Terms	$e ::= k \mid x \mid \lambda x.e \mid ee \mid \langle e, e \rangle \mid e.1 \mid e.2 \mid \mathbf{abort}(e) \mid \mathbf{fact}$
Types	$\tau ::= \beta \mid \mathbf{void} \mid (\Pi x : \tau)\tau \mid (\Sigma x : \tau)\tau \mid e = e \mathbf{in} \tau$
Reductions	$(\lambda x.e_1)e_2 \longrightarrow e_1[e_2/x]$ $\langle e_1, e_2 \rangle.1 \longrightarrow e_1$ $\langle e_1, e_2 \rangle.2 \longrightarrow e_2$

The terms of this new language include the terms of the simply typed lambda calculus. In addition, they contain terms of the form $\mathbf{abort}(e)$ that correspond to errors with justification e , and a single constant called \mathbf{fact} (whose significance will be explained later).

$\frac{\beta\text{-formation}}{\Gamma \vdash \beta \text{ Type}}$	$\frac{\text{void-formation}}{\Gamma \vdash \text{void Type}}$
$\frac{\Pi\text{-formation}}{\frac{\Gamma \vdash \tau_1 \text{ Type} \quad \Gamma, x: \tau_1 \vdash \tau_2 \text{ Type}}{\Gamma \vdash (\Pi x: \tau_1) \tau_2 \text{ Type}}}$	$\frac{\Sigma\text{-formation}}{\frac{\Gamma \vdash \tau_1 \text{ Type} \quad \Gamma, x: \tau_1 \vdash \tau_2 \text{ Type}}{\Gamma \vdash (\Sigma x: \tau_1) \tau_2 \text{ Type}}}$
$\frac{\text{equality-formation}}{\frac{\Gamma \vdash e_1: \tau \quad \Gamma \vdash e_2: \tau \quad \Gamma \vdash \tau \text{ Type}}{\Gamma \vdash (e_1 = e_2 \text{ in } \tau) \text{ Type}}}$	

Figure 4.2: Type formation rules for simple type theory.

4.3.1 Types

The primitive types of the system include the types β , and in addition include the type `void` (which represents the empty type). Types are closed under the dependent function, dependent product and equality type constructors; we describe these constructors below. The context-free grammar presented earlier only prescribes the structure of terms and types and does not prescribe when they are well-formed. We use the type system to dictate well-formedness. Well-formedness of terms is prescribed as usual by type inference rules that use assertions of the form $(\Gamma \vdash e: \tau)$. We assert type well-formedness by means of a new form of assertion, namely $(\Gamma \vdash \tau \text{ Type})$. This asserts that if Γ is a well-formed type assignment, then τ is a well-formed type. The type system includes a set of type formation rules that prescribe when a type is well-formed (see Figure 4.2).

4.3.2 Type Inhabitation

Figure 4.3 presents the introduction, elimination, and equality rules for the theory. The introduction rules prescribe how we may construct members of a type, the elimination rules prescribe how we may use those members in computations, and the equality rules prescribe when members of a type are equal. Types are considered equal only if they are identical, and hence we do

<p><i>hypothesis</i></p> $\frac{\Gamma, x: \tau_1, \Gamma' \vdash x: \tau_2}{\text{if } \tau_1 \text{ is } \alpha\text{-convertible to } \tau_2}$	<p><i>weakening</i></p> $\frac{\Gamma \vdash e: \tau_1}{\Gamma, x: \tau_2 \vdash e: \tau_1}$
<p><i>reflexivity</i></p> $\frac{\Gamma \vdash e: \tau_1}{\Gamma \vdash \mathbf{fact} : (e = e \text{ in } \tau_1)}$	<p><i>computation</i></p> $\frac{\Gamma \vdash e: \tau_1 \quad \Gamma \vdash e': \tau_1}{\Gamma \vdash \mathbf{fact} : (e = e' \text{ in } \tau_1)}$ <p style="text-align: center;">if e computes to e'</p>
<p><i>equality</i></p> $\Gamma \vdash \mathbf{fact} : (e = e' \text{ in } \tau_1)$ <p>if the equality can be deduced from Γ using only symmetry, transitivity and congruence</p>	<p><i>substitutivity</i></p> $\frac{\Gamma \vdash e_1 : \tau_1[e_2/x] \quad \Gamma \vdash \mathbf{fact} : (e_2 = e_3 \text{ in } \tau_2)}{\Gamma \vdash e_1 : \tau_1[e_3/x]}$
<p><i>void-elimination</i></p> $\frac{\Gamma \vdash e : \mathbf{void}}{\Gamma \vdash \mathbf{abort}(e) : \tau_1}$	
<p>Σ-introduction</p> $\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2[e_1/x]}{\Gamma \vdash \langle e_1, e_2 \rangle : (\Sigma x: \tau_1) \tau_2}$	<p>Π-introduction</p> $\frac{\Gamma, x: \tau_1 \vdash e: \tau_2}{\Gamma \vdash (\lambda x. e) : (\Pi x: \tau_1) \tau_2}$
<p>Σ-elimination</p> $\frac{\Gamma \vdash e : (\Sigma x: \tau_1) \tau_2}{\Gamma \vdash e.1 : \tau_1} \quad \frac{\Gamma \vdash e : (\Sigma x: \tau_1) \tau_2}{\Gamma \vdash e.2 : \tau_2[e.1/x]}$	<p>Π-elimination</p> $\frac{\Gamma \vdash e_1 : (\Pi x: \tau_1) \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2[e_2/x]}$
<p>Σ-equality</p> $\frac{\Gamma \vdash \mathbf{fact} : (e_1.1 = e_2.1 \text{ in } \tau_1) \quad \Gamma, x: \tau_1 \vdash \mathbf{fact} : (e_1.2 = e_2.2 \text{ in } \tau_2)}{\Gamma \vdash \mathbf{fact} : (e_1 = e_2 \text{ in } (\Sigma x: \tau_1) \tau_2)}$	<p>Π-equality</p> $\frac{\Gamma, x: \tau_1 \vdash \mathbf{fact} : (e_1 x = e_2 x \text{ in } \tau_2)}{\Gamma \vdash \mathbf{fact} : (e_1 = e_2 \text{ in } (\Pi x: \tau_1) \tau_2)}$

Figure 4.3: Inference rules for simple type theory.

not present any rules for the equality of types. We describe the type system in the following paragraphs.

Empty type (void)

`void` has no members, and hence there is no rule `void-intro`. Rule `void-elim` indicates that if we can ever show `void` to have a member e , then every type can be presumed to have a member. Such a (meaningless or erroneous) member is denoted by `abort(e)`.

Dependent Function Space $((\Pi x: \tau_1)\tau_2)$

Members of the dependent function type $(\Pi x: \tau_1)\tau_2$ are function abstractions $\lambda x.e$ that represent total functions with domain type τ_1 and range type $\tau_2[x]$ (rule `Π -intro`). The range type $\tau_2[x]$ depends on the argument x to the function. For example, in a language that includes the natural numbers, strings, and conditional expressions,

$$(\Pi m: \text{nat})(\Pi n: \text{nat})(\text{if } m \geq n \text{ then nat else string})$$

is the type of functions that take as arguments two numbers m and n . If $(m \geq n)$, then the functions return a number; otherwise, they return a string. Subtraction on the non-negative integers is an example of such a function:

$$\lambda m.\lambda n.\text{if } m \geq n \text{ then } m - n \text{ else "error"}$$

Functions are used in computations by applying them to arguments (rule `Π -elim`).

The dependent function space type is a generalization of the function space type. If x does not occur free in τ_2 , then the dependent type $(\Pi x: \tau_1)\tau_2$ is the same as the regular function space type $(\tau_1 \rightarrow \tau_2)$.

Dependent Product $((\Sigma x: \tau_1)\tau_2)$

Members of the dependent product type $(\Sigma x: \tau_1)\tau_2$ are ordered pairs $\langle e_1, e_2 \rangle$, where e_1 is a term of type τ_1 , and e_2 is a term of type $\tau_2[e_1/x]$ (rule `Σ -intro`). The pair's second component has a type $\tau_2[e_1/x]$ that depends on the first component e_1 of the pair. For example,

$$(\Sigma n: \text{nat})(\text{if even}(n) \text{ then nat else void})$$

is the type of pairs of natural numbers with even first components. Pairs are used in computations by projecting them to their first and second components and then using these components (rule Σ -**elim**).

The dependent product type is a generalization of the cartesian product type. If x does not occur free in τ_2 , then the dependent type $(\Sigma x: \tau_1)\tau_2$ is the same as the regular cartesian product type $(\tau_1 \times \tau_2)$.

Equality type $(e_1 = e_2 \text{ in } \tau)$

If e_1 and e_2 are equal members of type τ , then the equality type $(e_1 = e_2 \text{ in } \tau)$ has exactly one member, namely **fact**. Otherwise, the equality type is empty. The constant **fact** is a proxy for all proofs of the fact that e_1 and e_2 are equal members of type τ . It does not have any computational significance, and hence no computational operations are defined for it.

4.3.3 Propositional Interpretation

In a logic, the term $(e_1 = e_2)$ is a formula that may be true or false. The equality type $(e_1 = e_2 \text{ in } \tau)$ may be viewed as encoding such a formula for members of type τ . If the equality type is inhabited, then the formula $(e_1 =_{\tau} e_2)$ is true. If the type is empty, then the formula is false. This view equates types with formulas, type inhabitation with truth, and type emptiness with falsehood. It turns out that this view can be extended to the entire type system, yielding an isomorphism between types and formulas. This is known as the *formulas-as-types* principle [33, 10] or alternatively, the *propositions-as-types* principle, or the Curry-Howard isomorphism.

This isomorphism implies that every formula has a translation to a type; conversely, every type can be viewed as encoding a formula. Let \bar{P} be the type corresponding to formula P . We present the translation from (first-order) formulas to types in Figure 4.4. The meaning of each formula now depends on the meaning of the type it translates to. These meanings are summarized in Figure 4.5, and correspond to the *intuitionistic* (or *constructive*) meanings of the formulas. For example, the type

$$(\Pi x: \text{nat})(\Pi y: \text{nat})(\Pi z: \text{nat})((x = y \text{ in nat}) \times (y = z \text{ in nat}) \rightarrow (x = z \text{ in nat}))$$

Name of formula	Formula	Type	Name of type
absurdity	false	void	empty
truth	true	nat	any inhabited type
conjunction	$P \wedge Q$	$\overline{P} \times \overline{Q}$	cartesian product
implication	$P \Rightarrow Q$	$\overline{P} \rightarrow \overline{Q}$	function space
negation	$\neg P$	$\overline{P} \rightarrow \mathbf{void}$	
existential quantification	$\exists x: \tau. P_x$	$(\Sigma x: \tau) \overline{P}_x$	dependent product
universal quantification	$\forall x: \tau. P_x$	$(\Pi x: \tau) \overline{P}_x$	dependent function space

Figure 4.4: Propositions as types.

Connective	Notation	Intuitionistic meaning
conjunction	$P \wedge Q$	there is evidence for both P and Q.
negation	$\neg P$	there is no evidence for P; that is, P can never be proved.
implication	$P \Rightarrow Q$	there is a method for transforming any evidence for P into evidence for Q.
existential quantification	$\exists x: \tau. P_x$	we can find an element v of type τ such that there is evidence for $P[v/x]$.
universal quantification	$\forall x: \tau. P_x$	given any element v of type τ , we can find evidence for $P[v/x]$.

Figure 4.5: Intuitionistic meaning of logical connectives.

corresponds to the formula

$$\forall x.\forall y.\forall z.[(x =_{\text{nat}} y) \wedge (y =_{\text{nat}} z) \Rightarrow (x =_{\text{nat}} z)]$$

It asserts that equality on the natural numbers is transitive. As another example, the type

$$(\Pi x:\text{nat})(\Pi y:\text{nat})\neg(y = 0 \text{ in nat}) \rightarrow (\Sigma z:\text{nat} \times \text{nat})(x = (y * z.1 + z.2) \text{ in nat})$$

corresponds to the formula

$$\forall x.\forall y.(y \neq_{\text{nat}} 0) \Rightarrow \exists z.(x =_{\text{nat}} (y * z.1 + z.2))$$

This formula is the specification of integer division, with z being the pair of the quotient and remainder of the division of x by y . This is not a complete specification since we have no means of asserting that the remainder $z.2$ is less than the divisor y . In Chapter 5, we shall examine a full-fledged type theory that contains types of the form $(e_1 <_{\text{nat}} e_2)$, thus permitting the complete specification of integer division. Note that the simple type theory does not include disjunction. We have omitted disjunction from this discussion for pedagogical reasons. The full-fledged type theory of Chapter 5 includes disjunction.

The type system enables us to prove assertions of the form $(\Gamma \vdash t:\tau)$, namely that under the type assignment Γ , the term t has type τ . This asserts that the type τ is inhabited, or, under the propositions-as-types principle, that the formula encoded by type τ is true. The type system is thus also a formal logical system that permits us to prove formulas. Since the theory has atomic formulas representing equality of terms, the logic permits us to reason about the equality of terms in the language. It is thus a programming logic for the language. An assertion that a term t has type τ , when viewed logically, corresponds to the assertion that the program t meets its specification τ . Type checking is now isomorphic to program verification which we know is undecidable for any interesting language.

It is important to note that equality is *not* a single global relation. Rather, each type has its own equality relation. That is, the type $(e_1 = e_2 \text{ in } \tau)$ is well-formed only if e_1 and e_2 are both members of type τ (rule **=-formation**). The equality type can be used to check for equality of two terms only if it is known that the two terms are members of τ . In particular,

$(e = e \text{ in } \tau)$ is well-formed only if e is a member of type τ . Thus the equality relation is not globally reflexive, but is reflexive only at well-typed terms of types. This is expressed by the rule **reflexivity** in Figure 4.3. The relation *is* symmetric and transitive, as expressed by the rule **equality**.

Rule **computation** indicates that equality respects computation. Computation takes place by reducing terms using the reduction rules we presented earlier. Thus if a well-typed term t reduces (*computes*) to a well-typed term t' , then t is equal to t' . Rule Σ -**equality** indicates that two pairs are equal if their respective components are equal. Rule Π -**equality** indicates that two functions are equal if they map equal arguments to equal results (this is known as *extensional equality*).

The type-specificity of the type $(e_1 = e_2 \text{ in } \tau)$ implies that type formation is not context free. For example, $(x = y \text{ in nat})$ is well-formed only in contexts where x and y are variables of type **nat**. This is why we need to axiomatize type formation within the logic by means of the assertions $(\Gamma \vdash \tau \text{ Type})$ and the type formation rules.

The simple type theory presented in this section is a first-order theory since it does not have a type of types, and hence does not permit quantification over types (formulas). Types are distinct from terms and are not values of the language. Full-fledged type theories treat types as values and include higher-order types. Although such theories are considerably more powerful than the simple theory of this chapter, they are also considerably more complex. We postpone a discussion of such theories to Chapter 5.

4.4 Observation Type Theory

In the previous section, we enriched the simply typed lambda calculus to obtain a simple constructive type theory. Part of this enrichment involved our generalizing the function and product type constructors to dependent type constructors. We then presented a propositional interpretation of the theory that permitted us to reason about the equality of terms. We now present a similar enrichment of the type system of ILC. Part of this enrichment involves our generalizing the observation constructor **Obs** τ to a dependent constructor (**Get** $x:\theta \Leftarrow e \text{ in } \tau$). We call the resulting theory *observation type theory* (OTT). The language of OTT is defined as follows:

(1)	$(\lambda x:\omega.e_1)(e_2)$	\longrightarrow	$e_1[e_2/x]$	
(2)	$\langle e_1, e_2 \rangle.i$	\longrightarrow	e_i	for $i = 1, 2$
(3)	$\mathbf{letref} \ v^*:\mathbf{Ref} \ \theta := e_1 \ \mathbf{in} \ \mathbf{obs}(e_2)$	\longrightarrow	$\mathbf{obs}(e_2)$	if $v^* \notin V(e_2)$
(4)	$\mathbf{letref} \ v^*:\mathbf{Ref} \ \theta := e_1 \ \mathbf{in} \ \mathbf{get} \ x:\theta \leftarrow v^* \ \mathbf{in} \ e_2$	\longrightarrow	$\mathbf{letref} \ v^*:\mathbf{Ref} \ \theta := e_1 \ \mathbf{in} \ e_2[e_1/x]$	
(5)	$\mathbf{letref} \ v^*:\mathbf{Ref} \ \theta_1 := e_1 \ \mathbf{in} \ \mathbf{get} \ x:\theta_2 \leftarrow w^* \ \mathbf{in} \ e_2$	\longrightarrow	$\mathbf{get} \ x':\theta_2 \leftarrow w^* \ \mathbf{in} \ \mathbf{letref} \ v^*:\mathbf{Ref} \ \theta_1 := e_1 \ \mathbf{in} \ e_2[x'/x]$	if $v^* \neq w^*$ and $x' \notin V(e_1) \cup V(e_2)$
(6)	$v^* := e_1 ; \mathbf{obs}(e_2)$	\longrightarrow	$\mathbf{obs}(e_2)$	
(7)	$v^* := e_1 ; \mathbf{get} \ x:\theta \leftarrow v^* \ \mathbf{in} \ e_2$	\longrightarrow	$v^* := e_1 ; e_2[e_1/x]$	
(8)	$v^* := e_1 ; \mathbf{get} \ x:\theta \leftarrow w^* \ \mathbf{in} \ e_2$	\longrightarrow	$\mathbf{get} \ x':\theta \leftarrow w^* \ \mathbf{in} \ v^* := e_1 ; e_2[x'/x]$	if $v^* \neq w^*$ and $x' \notin V(e_1) \cup V(e_2)$
(9)	$\mathbf{app}(\mathbf{obs}(e))$	\longrightarrow	e	

Figure 4.6: Reduction rules.

Terms	$e ::= k \mid x \mid v^* \mid \lambda x.e \mid ee \mid \langle e, e \rangle \mid e.1 \mid e.2 \mid \mathbf{abort}(e) \mid \mathbf{fact} \mid$ $\mathbf{letref} \ v^* := e \ \mathbf{in} \ e \mid \mathbf{get} \ x \leftarrow e \ \mathbf{in} \ e \mid e := e ; e$
Applicative types	$\tau ::= \beta \mid \mathbf{void} \mid (\Pi x:\tau)\tau \mid (\Sigma x:\tau)\tau \mid e = e \ \mathbf{in} \ \tau$
Storage types	$\theta ::= \tau \mid \mathbf{Ref} \ \theta \mid (\Pi x:\theta)\theta \mid (\Sigma x:\theta)\theta \mid e = e \ \mathbf{in} \ \theta \mid (e =_\theta e)$
Assertion types	$\alpha ::= \tau \mid \mathbf{Obs} \ \alpha \mid \mathbf{Get} \ x:\theta \leftarrow e \ \mathbf{in} \ \alpha \mid (\mathbf{All} \ x:\alpha)\alpha \mid (\mathbf{Some} \ x:\alpha)\alpha \mid$ $e = e \ \mathbf{in} \ \alpha \mid (e =_\theta e) \mid (e \sim \alpha)$
Observation types	$\omega ::= \theta \mid \alpha \mid (\Pi x:\omega)\omega \mid (\Sigma x:\omega)\omega \mid e = e \ \mathbf{in} \ \omega$

The terms of OTT extend the terms of ILC with the constant **fact** and the term **abort**(e). These terms were explained in the previous section. The **letref**, **get**, and $:=$ terms have the same meaning as in ILC. The reduction rules are exactly the same as ILC's rules and are reproduced in Figure 4.6. In the sequel, we use l -like variables for reference valued terms (i.e., terms of some type **Ref** θ), and t -like variables for observer terms (i.e., terms of some type **Obs** τ).

In the following sections, we first discuss the logical concepts behind OTT, then we present a formal system based on these concepts.

4.5 Logical Concepts of OTT

Observation Type Theory (OTT) is a constructive logic for which ILC forms the language of constructions. Central to the theory is a certain correspondence between variables and references. Various aspects of this similarity have been noted by numerous researchers, and techniques applicable to variables have often been applied to references. For instance, in Hoare logic, references are treated just as variables of logic, and in common imperative programming parlance, references are simply called variables.

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), function abstraction (parameter passing), and observer abstraction (storing). Variables cannot participate in such semantic constructions (although some effects of references can be obtained in the setting of Girard’s Linear Logic [25], which exploits the duality between inputs (variables) and outputs (values)). Second, references are *reusable*. This permits data structures to be updated without copying.

4.5.1 Quantification of References

In simple constructive type theory, a proof of a universally quantified proposition $(\forall x:\tau)P$ is a *method* (function or rule) that, 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 $P[e/x]$. A 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 since 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 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 we use alternative terminology since l is not a variable but an expression). If ϕ is a proof of $(\mathbf{Get} \ x: \theta \Leftarrow l \ \mathbf{in} \ P)$, then we can obtain $P[e/x]$ with proof $(l := e ; \phi)$. Note that the reference l is not discharged, but 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 (discharged) after a specialization.

The second kind 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 $(\mathbf{letref} \ v^* := e \ \mathbf{in} \ \phi)$.

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

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 $\text{Ref nat} \times \text{Ref nat}$. Given a construction t of

$$(\text{Get } x:\text{nat} \Leftarrow l.1 \text{ in Get } y:\text{nat} \Leftarrow l.2 \text{ in } Q[x, y]),$$

we can form a construction for

$$(\text{Get } x:\text{nat} \Leftarrow l.1 \text{ in Get } y:\text{nat} \Leftarrow l.2 \text{ in } Q[x + 1, y])$$

as $(\text{get } x \Leftarrow l.1 \text{ in } (l.1 := x + 1 ; t))$. 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 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 u 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 these functions can be applied to values. 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.

4.5.3 State-assertions and State-judgements

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 for 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 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 . Thus the quantification over states is contained in the proposition Q rather than the judgement, just as in the case of $(\forall x: \mathbf{nat})P$ above.

This suggests the need for judgements that involve quantification over states. Propositional expressions appearing in such judgements refer to propositions in specific states. This closely corresponds to the notion of “assertions” in Hoare logic [31]. So, we use notation reminiscent of the latter. We call such expressions *state-assertions* (*assertions* for short) and write them enclosed in braces as $\{Q\}$. 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 asserts 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 evidence for the hypothesis **Get**-assertion, then $(\mathbf{get\ } y \leftarrow v^* \mathbf{ in\ } \langle y, \phi \rangle)$ is evidence for the right hand side. 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 $(\forall y: \mathbf{nat}) y = t$, i.e., that all natural numbers are equal to 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 that 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 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

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

Assertion-introduction asserts that if Q holds in all states, then Q holds in states satisfying Γ . *Assertion-elimination* asserts that if Q holds in all states that satisfy Γ , and if Γ has no state-assertions (and hence is satisfied by all states), then Q holds in all states. 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. Thus, for every logical operator on propositions, there is a corresponding operator on state-assertions. The rules for the state-assertional operators are similar to those for propositional operators, the difference being that they deal with state-assertions. The copy of the standard logic used for state-assertions is called *assertion logic*.

4.5.4 Noninterference

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

$$\text{Get } x:\theta \leftarrow l \text{ in Get } y:\theta \leftarrow l \text{ in } Q[x, y]$$

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' that 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 that may denote the same reference — that allows the flexibilities mentioned in Section 4.5.2.

The ambiguity involved in **Get**-quantification 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 [67, 83]. 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 [66, 83].

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

4.5.5 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 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 \mathbf{Obs} type constructor in much the same fashion as \forall quantification generalizes the \rightarrow type constructor of functional programs. This analogy is not perfect since \mathbf{Obs} cannot be *defined* to be a special case of **Get** — \mathbf{Obs} is also needed in contexts where no references are available, but **Get** necessarily involves a reference.

4.6 Formal System for OTT

We now present a formal system for OTT. We first describe the types of OTT, and then present type inhabitation rules for the relevant core of OTT. The complete set of rules may be found in Appendix B.

4.6.1 Types

The syntax of types is as follows:

$$\begin{aligned}
\text{Applicative types } \tau & ::= \beta \mid \mathbf{void} \mid (\Pi x:\tau)\tau \mid (\Sigma x:\tau)\tau \mid e = e \text{ in } \tau \\
\text{Storage types } \theta & ::= \tau \mid \mathbf{Ref } \theta \mid (\Pi x:\theta)\theta \mid (\Sigma x:\theta)\theta \mid e = e \text{ in } \theta \mid (e =_{\theta} e) \\
\text{Assertion types } \alpha & ::= \tau \mid \mathbf{Obs } \alpha \mid \mathbf{Get } x:\theta \Leftarrow e \text{ in } \alpha \mid (\mathbf{All } x:\alpha)\alpha \mid (\mathbf{Some } x:\alpha)\alpha \mid \\
& \quad e = e \text{ in } \alpha \mid (e =_{\theta} e) \mid (e \sim \alpha) \\
\text{Observation types } \omega & ::= \theta \mid \alpha \mid (\Pi x:\omega)\omega \mid (\Sigma x:\omega)\omega \mid e = e \text{ in } \omega
\end{aligned}$$

The types of OTT are stratified into four layers: τ , θ , α and ω . The τ and θ layers enhance ILC's τ and θ layers in the same way that simple type theory enhances the simply typed lambda calculus. That is, they include empty (**void**) and equality ($e_1 = e_2 \text{ in } T$) types, and they generalize the product and function space constructions to dependent product and dependent function space constructions. The ω layer is an enhancement of ILC's ω layer; however, 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 4.5.1, 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:

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

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 $\{\alpha\}$.

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

Figure 4.7: Equivalence of α type-terms.

The reference equality type ($e_1 =_{\theta} e_2$) is well-formed only if e_1 and e_2 are of some reference types $\text{Ref } \theta$ and $\text{Ref } \theta'$. 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-terms of the α layer have a considerable degree of redundancy because they capture implicit dependence on the state. Figure 4.7 lists the equivalences imposed on these 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. The equivalences 9 and 10 state that **All** and **Some** operating on applicative types are equivalent to Π and Σ .

Since we have four layers, we have four kinds of judgements that assert well-formedness. We use $(\Gamma \vdash T \text{Type}_{\tau})$, $(\Gamma \vdash T \text{Type}_{\theta})$, $(\Gamma \vdash T \text{Type}_{\alpha})$, and $(\Gamma \vdash T \text{Type}_{\omega})$ to assert that T is a τ type, θ type, α type, and ω type respectively. Figure 4.8 presents the type formation rules for the new types introduced in these layers; miscellaneous rules and rules for β , **void**, $(\Pi x:S)T$,

<p>τ-to-θ</p> $\frac{\Gamma \vdash \tau \text{Type}_\tau}{\Gamma \vdash \tau \text{Type}_\theta}$	<p>τ-to-α</p> $\frac{\Gamma \vdash \tau \text{Type}_\tau}{\Gamma \vdash \tau \text{Type}_\alpha}$
<p>θ-to-ω</p> $\frac{\Gamma \vdash \theta \text{Type}_\theta}{\Gamma \vdash \theta \text{Type}_\omega}$	<p>α-to-ω</p> $\frac{\Gamma \vdash \alpha \text{Type}_\alpha}{\Gamma \vdash \alpha \text{Type}_\omega}$
<p><i>Ref-formation</i></p> $\frac{\Gamma \vdash \theta \text{Type}_\theta}{\Gamma \vdash (\text{Ref } \theta) \text{Type}_\theta}$	<p><i>typeless-equality-formation</i></p> $\frac{\Gamma \vdash e_1: \theta_1 \quad \Gamma \vdash e_2: \theta_2 \quad \Gamma \vdash \theta_1 \text{Type}_\theta \quad \Gamma \vdash \theta_2 \text{Type}_\theta}{\Gamma \vdash (e_1 =_\theta e_2) \text{Type}_\theta}$
<p><i>Obs-formation</i></p> $\frac{\Gamma \vdash \alpha \text{Type}_\alpha}{\Gamma \vdash \text{Obs } \alpha \text{Type}_\alpha}$	<p><i>Get-formation</i></p> $\frac{\Gamma \vdash l: \text{Ref } \theta \quad \Gamma, x: \theta \vdash \alpha \text{Type}_\alpha}{\Gamma \vdash (\text{Get } x: \theta \leftarrow l \text{ in } \alpha) \text{Type}_\alpha}$
<p><i>All -formation</i></p> $\frac{\Gamma \vdash \alpha_1 \text{Type}_\alpha \quad \Gamma, x: \alpha_1 \vdash \alpha_2 \text{Type}_\alpha}{\Gamma \vdash (\text{All } x: \alpha_1) \alpha_2 \text{Type}_\alpha}$	<p><i>Some -formation</i></p> $\frac{\Gamma \vdash \alpha_1 \text{Type}_\alpha \quad \Gamma, x: \alpha_1 \vdash \alpha_2 \text{Type}_\alpha}{\Gamma \vdash (\text{Some } x: \alpha_1) \alpha_2 \text{Type}_\alpha}$
<p><i>equality-formation</i></p> $\frac{\Gamma \vdash e_1: \alpha_1 \quad \Gamma \vdash e_2: \alpha_1 \quad \Gamma \vdash \alpha_1 \text{Type}_\alpha}{\Gamma \vdash (e_1 = e_2 \text{ in } \alpha_1) \text{Type}_\alpha}$	<p><i>typeless-equality-formation</i></p> $\frac{\Gamma \vdash (e_1 =_\theta e_2) \text{Type}_\theta}{\Gamma \vdash (e_1 =_\theta e_2) \text{Type}_\alpha}$
<p><i>noninterference-formation</i></p> $\frac{\Gamma \vdash l: \text{Ref } \theta \quad \Gamma \vdash \alpha \text{Type}_\alpha}{\Gamma \vdash (l \sim \alpha) \text{Type}_\alpha}$	

Figure 4.8: Type formation rules for observation type theory.

$(\Sigma x: S)T$ and $(e_1 = e_2 \text{ in } T)$ have been omitted since they were presented in Figure 4.2. Appendix B contains a complete list of rules of OTT.

4.6.2 Type Inhabitation

The inference rules of the constructive logic OTT 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 chapter. The full set of rules may be found in Appendix B.

The rules for `void`, Π , and Σ types (in all layers) are conventional. The rules for $=$ types are also conventional except that, for assertion types, equality is defined to be extensional over states. `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

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

Figure 4.9 presents the introduction and elimination rules for the α layer of OTT. 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.

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 $t: (\text{Get } x: \theta \Leftarrow l \text{ in } \alpha)$ means that t is an observer method that proves `Get` $x: \theta \Leftarrow l \text{ in } \alpha$. 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 is an applicative or storage type (which is not permitted to have an observer term as a member).

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

Dereference

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

Assignment

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

Creation

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

Figure 4.9: Inference rules for observation type theory.

$$\frac{\sim \text{-axiom-}\tau \quad \Gamma \vdash l : \text{Ref } \theta \quad \Gamma \vdash \tau_1 \text{ Type}_\tau}{\Gamma \vdash \text{fact} : (l \sim \tau_1)}$$

$$\frac{\sim \text{-intro-Obs} \quad \Gamma \vdash \text{fact} : (l \sim \alpha)}{\Gamma \vdash \text{fact} : (l \sim \text{Obs } \alpha)}$$

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

$$\frac{\sim \text{-intro-All} \quad \Gamma \vdash \text{fact} : (l \sim \alpha_1) \quad \Gamma, x : \alpha_1 \vdash \text{fact} : (l \sim \alpha_2)}{\Gamma \vdash \text{fact} : (l \sim (\text{All } x : \alpha_1) \alpha_2)}$$

$$\frac{\sim \text{-intro-Some} \quad \Gamma \vdash \text{fact} : (l \sim \alpha_1) \quad \Gamma, x : \alpha_1 \vdash \text{fact} : (l \sim \alpha_2)}{\Gamma \vdash \text{fact} : (l \sim (\text{Some } x : \alpha_1) \alpha_2)}$$

$$\frac{\sim \text{-intro-equality} \quad \Gamma \vdash \text{fact} : (l \sim (e_1 = e_2 \text{ in } \alpha))}{\Gamma \vdash \text{fact} : (l \sim (e_1 =_\theta e_2))}$$

$$\frac{\sim \text{-intro-typeless-equality} \quad \Gamma \vdash \text{fact} : (l \sim (e_1 =_\theta e_2))}{\Gamma \vdash \text{fact} : (l \sim (e_1 =_\theta e_2))}$$

Figure 4.10: Inference rules for observation type theory (continued).

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, namely $(\text{letref } v^* := e \text{ in } t)$, 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 $(\text{Get } x : \theta \Leftarrow v^* \text{ in } \alpha)$ 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 , 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 (see Figure 4.10). We only describe the most interesting rule, namely \sim -**intro-Get**. The meaning of a type of the form $(\mathbf{Get} \ x:\theta_2 \Leftarrow l' \ \mathbf{in} \ \alpha)$ may depend on the content of reference l' . Thus, if l is not to interfere with such a type, l should be distinct from the reference l' :

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

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.

4.6.3 Assertion Logic

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 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 4.11 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),

$$\begin{array}{c}
\textit{Assert-intro} \\
\frac{\Gamma \vdash t : \alpha_1}{\Gamma \vdash t : \{\alpha_1\}}
\end{array}
\qquad
\begin{array}{c}
\textit{Assert-elim} \\
\frac{\Gamma \vdash t : \{\alpha_1\}}{\Gamma \vdash t : \alpha_1} \quad (\text{if } \Gamma \text{ has no } \{\alpha\} \text{ types})
\end{array}$$

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

$$\begin{array}{c}
\textit{Dereference} \\
\frac{\Gamma \vdash l : \text{Ref } \theta \quad \Gamma, x : \theta, \{\text{Get } z : \theta \Leftarrow l \text{ in } x = z\} \vdash t : \{\text{Obs } \alpha\}}{\Gamma \vdash (\text{get } x \Leftarrow l \text{ in } t) : \{\text{Get } x : \theta \Leftarrow l \text{ in } \alpha\}}
\end{array}$$

$$\begin{array}{c}
\textit{Assignment} \\
\frac{\Gamma \vdash l : \text{Ref } \theta \quad \Gamma, x : \theta \vdash \{(l \sim \alpha)\} \quad \Gamma \vdash e : \theta \quad \Gamma \vdash t : (\text{Get } x : \theta \Leftarrow l \text{ in } \alpha)}{\Gamma \vdash (l := e ; t) : \{\text{Obs } \alpha[e/x]\}}
\end{array}$$

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

Figure 4.11: Inference rules for state-assertions in OTT.

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 assertion logic counterparts of the type inferences rules of Figure 4.9. 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 Figure 4.9, 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 that 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 $\neg(A(0) =_{\theta} A(x))$ does not hold for all states (since there are states 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 [6], can be used to suppress the noncomputational constructions.

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

Some -introduction

$$\frac{\Gamma \vdash s: \{S\} \quad \Gamma \vdash t: \{T[s/x]\}}{\Gamma \vdash \underline{\langle s, t \rangle}: \{(\mathbf{Some} \ x: S)T\}}$$

Some -elimination

$$\frac{\Gamma \vdash e: \{(\mathbf{Some} \ x: S)T\}}{\Gamma \vdash e_{\underline{1}}: \{S\}}$$

$$\frac{\Gamma \vdash e: \{(\mathbf{Some} \ x: S)T\}}{\Gamma \vdash e_{\underline{2}}: \{T[e_{\underline{1}}/x]\}}$$

All -introduction

$$\frac{\Gamma, x: \{S\} \vdash t: \{T\}}{\Gamma \vdash \underline{\lambda}x.t: \{(\mathbf{All} \ x: S)T\}}$$

All -elimination

$$\frac{\Gamma \vdash f: \{(\mathbf{All} \ x: S)T\} \quad \Gamma \vdash s: \{S\}}{\Gamma \vdash \underline{f}(s): \{T[s/x]\}}$$

The underscored constructors “ $\underline{\langle}$ ”, “ $\underline{\lambda}$ ”, “ $\underline{\lambda}$ ” and “ $\underline{()}$ ”, signify the fact that they are “state-indexical”, *i.e.*, all their components are interpreted in the same state.

4.7 Modeling Hoare Logic

Hoare logic and its extension, Specification logic, can be modeled in observation type theory. Commands, which are conventionally treated as transformers of state, are treated in OTT as observer transformers. A command \bar{s} is a function that maps an observer c to an enhanced observer $\bar{s}(c)$ which first carries out the actions of \bar{s} and then continues with the observer c . With this interpretation, 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 modeled 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 effect of the Hoare-logic rule

$$\frac{P \text{ implies } Q \quad \{Q\}s\{R\}}{\{P\}s\{R\}}$$

is achieved by the following inference using OTT:

$$\frac{\frac{(p : \{P\}) \quad \frac{f : (P \text{ implies } Q)}{f : \{P \text{ implies } Q\}}}{f(p) : \{Q\}} \quad \frac{(r : R \text{ implies } C) \quad \frac{s : (R \text{ implies } C) \rightarrow (Q \text{ implies } C)}{s(r) : (Q \text{ implies } C)}}{s(r) : \{Q \text{ implies } C\}}}{s(r)(f(p)) : \{C\}}}{\lambda p.s(r)(f(p)) : \{P \text{ implies } C\}}}{\lambda r.\lambda p.s(r)(f(p)) : (R \text{ implies } C) \rightarrow (P \text{ implies } C)}$$

The terms p and $f(p)$ correspond to proofs of the assertions $\{P\}$ and $\{Q\}$ respectively, and do not have any computational significance. Thus, the computational effect of $\lambda r.\lambda p.s(r)(f(p))$ is the same as that of s (since $\lambda r.s(r) = s$). Ignoring the non-computational constructions, we can summarize the above derivation as follows:

$$\frac{f : (P \text{ implies } Q) \quad s : (R \text{ implies } C) \rightarrow (Q \text{ implies } C)}{s : (R \text{ implies } C) \rightarrow (P \text{ implies } C)}$$

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 without complications involving the context. Moreover, since all state-dependencies need to be explicitly stated by means of dereferences, reasoning about aliasing of references is simplified.

4.8 Examples

In this section, we present examples of the use of OTT in reasoning about ILC programs. We first introduce notation that simplifies our presentation.

4.8.1 Notation

In Section 2.3.1, we introduced a notation that permits us to abbreviate the term $(\mathbf{get} \ x \Leftarrow l \ \mathbf{in} \ t)$ to $t[l \uparrow / x]$ under certain conditions. In a similar fashion, under certain conditions, we permit the type $(\mathbf{Get} \ x:\theta \Leftarrow l \ \mathbf{in} \ T)$ to be abbreviated to $T[l \uparrow / x]$.

A term $(\mathbf{Get} \ x:\theta \Leftarrow l \ \mathbf{in} \ \alpha)$ is sugared by first using the type equality rules of Figure 4.7 to propagate the \mathbf{Get} past all outermost \mathbf{Obs} , \mathbf{All} and \mathbf{Some} constructors in α , getting rid of redundant \mathbf{Gets} . Then, subterms of the form $(\mathbf{Get} \ x:\theta \Leftarrow l \ \mathbf{in} \ (e_1 = e_2 \ \mathbf{in} \ T))$ are abbreviated to $(e_1[l \uparrow / x] = e_2[l \uparrow / x] \ \mathbf{in} \ T)$ if x does not occur free in T .

A type $T[l \uparrow]$ is desugared by introducing \mathbf{Gets} at the innermost type expressions that contain the occurrences of $l \uparrow$. The type equalities of Figure 4.7 can then be used to propagate the \mathbf{Gets} outward, if desired. For example,

$$\begin{aligned} \mathbf{Get} \ x:\mathbf{nat} \Leftarrow v \ \mathbf{in} \ (x = 3 \ \mathbf{in} \ \mathbf{nat}) &\equiv (v \uparrow = 3 \ \mathbf{in} \ \mathbf{nat}) \\ \mathbf{Get} \ x:(a = b \ \mathbf{in} \ S) \Leftarrow v \ \mathbf{in} \ (x = \mathbf{fact} \ \mathbf{in} \ (\mathbf{Get} \ y:\mathbf{nat} \Leftarrow w \ \mathbf{in} \ (y = 6 \ \mathbf{in} \ \mathbf{nat}))) & \\ &\equiv v \uparrow = \mathbf{fact} \ \mathbf{in} \ (w \uparrow = 6 \ \mathbf{in} \ \mathbf{nat}) \end{aligned}$$

In Chapter 2, our examples assumed that ILC was enhanced with user-defined type constructors, named records and record types, universal polymorphism, and constructs such as \mathbf{let} , \mathbf{letrec} and $\mathbf{if-then-else}$. Our examples in this chapter involve verifying some examples of Chapter 2. We assume that OTT is enhanced with rules to handle these additional constructs. This enhancement is nontrivial and is postponed to subsequent chapters. Chapter 5 describes a full-fledged, higher-order, constructive type theory. Chapter 6 describes how we may use

well-founded orderings (based on inductive types) to reason about recursive programs within constructive type theory.

An unfortunate aspect of constructive type theory is that the constructions corresponding to logical proofs get intermingled with computational terms, and thus simple programs get encoded as long, unwieldy terms. For example, consider the type

$$(\Pi x : \mathbf{nat})((x > 0) \rightarrow (\Sigma y : \mathbf{nat})(x = y + 1 \text{ in } \mathbf{nat}))$$

that specifies the predecessor function on the natural numbers. Members of this type are functions of the form $\lambda x. \lambda p. \langle e, r \rangle$, where x is a natural number, p is a proof that $(x > 0)$, e is some expression, and r is a proof that $(x = e + 1)$. For example, the function $\lambda x. \lambda p. \langle x - 1, \mathbf{fact} \rangle$ is a member of the above type. Computationally, this has the effect of the function $\lambda x. (x - 1)$, with the terms p and \mathbf{fact} being useless clutter. The clutter gets worse with larger examples. In the interest of readability, we shall ignore subterms that do not have any computational significance. For the remainder of this thesis, when we say that a term t is a member of type T , we mean that t is the computationally significant part of some member of T . For example, we say that $\lambda x. (x - 1)$ is a member of

$$(\Pi x : \mathbf{nat})((x > 0) \rightarrow (\Sigma y : \mathbf{nat})(x = y + 1 \text{ in } \mathbf{nat}))$$

and mean that $\lambda x. (x - 1)$ describes the computational effect of some actual member of the above type.

Finally, we shall abbreviate the term $(e_1 = e_2 \text{ in } T)$ to $(e_1 = e_2)$ when the type of the terms is obvious from the context.

4.8.2 Example: Factorial

To illustrate the inference rules of OTT, we verify the factorial program that was presented in Section 2.3.2. The factorial program is written as:

```

factorial =  $\lambda m:\text{nat}.$  app(letref  $n:\text{Ref nat} := m$  in
  letref  $acc:\text{Ref nat} := 1$  in
  letrec  $fac:\text{Obs nat} =$  if ( $n \uparrow < 2$ ) then obs( $acc \uparrow$ )
    else  $acc := n \uparrow * acc \uparrow;$ 
       $n := n \uparrow - 1;$ 
       $fac$ 
  in  $fac$ )

```

This program achieves iteration by means of a tail-recursive use of the `letrec` construct. Although we have not yet described how to reason about such a construct within constructive type theory, the basic intuition behind the reasoning is well-known. We associate an invariant with the loop, and ensure that execution of the loop preserves the invariant. We use well-founded induction to ensure that the loop terminates. Consider the program fragment

```

letrec  $fac:\text{Obs nat} =$  if ( $n \uparrow < 2$ ) then obs( $acc \uparrow$ )
  else  $acc := n \uparrow * acc \uparrow;$ 
     $n := n \uparrow - 1;$ 
     $fac$ 
in  $fac$ 

```

Assume the following hypotheses:

$$\begin{aligned}
& n, acc : \text{Ref nat} \\
& \neg(n =_{\theta} acc) \\
& \text{WF}(\text{nat} \times \text{nat}, <_{pr1})
\end{aligned}$$

where $<_{pr1}$ is the comparison operator that compares two pairs by comparing their first components (i.e. $<_{pr1} = \lambda p.\lambda q.(p.1 <_{\text{nat}} q.1)$), and where $\text{WF}(\text{nat} \times \text{nat}, <_{pr1})$ asserts that the comparison operator $<_{pr1}$ is well-founded over pairs of natural numbers.

We wish to prove that fac returns a natural number whose value is $(n \uparrow! * acc \uparrow)$ in the current state. In the formalism of type theory, we wish to show that fac is a member of the type $(\text{Some } x : \text{nat})(x = (n \uparrow! * acc \uparrow) \text{ in } \text{nat})$. (Recall that we are assuming that noncomputational constructions are suppressed, and hence although this assertion contains pairs as members, we only focus on the first components of the pairs). We abbreviate this type to $\{n \uparrow! * acc \uparrow\}_{\text{nat}}$, and we prove that fac belongs to it. The proof proceeds by well-founded induction. Assume

$$\begin{array}{c}
\frac{\frac{\frac{(n' \neq 2)}{fac : \text{Obs} (\langle n \uparrow, acc \uparrow \rangle <_{pr1} \langle n', acc' \rangle)}{\text{implies } \{n \uparrow! * acc \uparrow\}_{nat}}}{(n := n' - 1 ; fac) : \text{Obs} (\langle n' - 1, acc \uparrow \rangle <_{pr1} \langle n', acc' \rangle)}{\text{implies } \{(n' - 1)! * acc \uparrow\}_{nat}}}{(acc := n' * acc' ; n := n' - 1 ; fac) : \text{Obs} (\langle n' - 1, n' * acc' \rangle <_{pr1} \langle n', acc' \rangle)}{\text{implies } \{(n' - 1)! * n' * acc'\}_{nat}}}{(acc := n' * acc' ; n := n' - 1 ; fac) : \text{Obs } \{n'! * acc'\}_{nat}}}{\frac{(n' < 2)}{acc' : \{acc'\}_{nat}}}{acc' : \text{Obs } \{n'! * acc'\}_{nat}}}{\frac{\text{if } n' < 2 \text{ then } acc' \text{ else } acc := n' * acc' ; n := n' - 1 ; fac) : \text{Obs } \{n'! * acc'\}_{nat}}{\text{get } n' \Leftarrow n \text{ in get } acc' \Leftarrow acc \text{ in } \text{if } n' < 2 \text{ then } acc' \text{ else } acc := n' * acc' ; n := n' - 1 ; fac) : \text{Obs } \{n \uparrow! * acc \uparrow\}_{nat}}}{fac : \text{Obs } \{n \uparrow! * acc \uparrow\}_{nat}}
\end{array}$$

Figure 4.12: Correctness proof of the factorial program.

that for some $n', acc' : \text{nat}$, the constant fac is a member of the type $(\text{Obs} (\langle n \uparrow, acc \uparrow \rangle <_{pr1} \langle n', acc' \rangle) \text{implies } \{n \uparrow! * acc \uparrow\}_{nat})$. The deduction in Figure 4.12 is then valid.

The first two inferences in the second column are by rule **Assignment**, with the first inference using the hypothesis $\neg(n =_{\theta} acc)$ to prove noninterference. The third inference is by **implies -elim** since the proposition $(n' - 1, n' * acc') <_{pr1} (n', acc')$ is true, and since $(n' - 1)! * n' * acc' = n'! * acc'$. The fourth inference is by case analysis on $(n < 2) \mid (n \neq 2)$, while the fifth inference is by folding the definition of fac . Finally, well-founded induction over $\text{WF}(\text{nat} \times \text{nat}, <_{pr1})$ lets us conclude that fac is a member of $\text{Obs } \{n \uparrow! * acc \uparrow\}_{nat}$.

4.8.3 A Derived Rule for Sugared Assignments

All the inference rules of Figures 4.9– 4.11 are valid for both proper terms and sugared terms of OTT, except for the rule **Assignment**. To see that **Assignment** is not valid for sugared terms, consider following Hoare triple

$$\{1 = 1\} \quad (\text{if } v \uparrow = 0 \text{ then } v \text{ else } w) := 1 \quad \{(\text{if } v \uparrow = 0 \text{ then } v \text{ else } w) \uparrow = 1\}$$

that corresponds to the following OTT judgement

$$(\lambda c. (\text{if } v \uparrow = 0 \text{ then } v \text{ else } w) := 1 ; c) : \\ (((\text{if } v \uparrow = 0 \text{ then } v \text{ else } w) \uparrow = 1) \text{ implies } C) \rightarrow ((1 = 1) \text{ implies } C)$$

This is clearly invalid, since if the reference v initially contains 0, then the assignment assigns 1 to reference v , while the postcondition asserts that reference w contains 1 (since $v \uparrow$ is no longer 0). Were we to use the rule **Assignment** directly, this invalid judgement would be derivable.

Reynolds rules out such assignments in his Specification Logic [67], declaring the reference expression $(\text{if } v \uparrow = 0 \text{ then } v \text{ else } w)$ to be a “bad” variable (i.e. a bad reference). A reference is considered “bad” if assigning it a value changes its own meaning. The assignment rule of specification logic is only valid for good variables.

In ILC, an expression such as $(\text{if } v \uparrow = 0 \text{ then } v \text{ else } w)$ is not a reference, but an *observation* of a reference (i.e., it is state-dependent). Such an observation has to be carried out in a specific state. For example, we might write

$$\text{get } x \leftarrow v \text{ in } (\text{if } x = 0 \text{ then } v \text{ else } w) := 1 ; c$$

thereby observing v in the state *before* the assignment. Similarly, in OTT, the type of c needs to explicate all state dependencies; for instance, we might write

$$c : (\text{Get } x : \text{nat} \leftarrow v \text{ in } (\text{Get } y : \text{nat} \leftarrow (\text{if } x = 0 \text{ then } v \text{ else } w) \text{ in } (y = 1 \text{ implies } C)))$$

The rule **Assignment** cannot be used any more, and there is no way to deduce the invalid judgement in OTT.

Since we explicate all state dependencies, we have no need for the good variable concept of specification logic. Not only does this make the logic simpler, it also buys us significant generality. Examples of useful “bad” variables include array subscripts and pointer dereferences. Such bad variables can be used in assignments in OTT as long as they do not interfere with the properties being proved.

However, when we consider sugared terms, state dependencies are no longer explicit and we do need to ensure that the good variable condition is met. In this section, we use OTT to derive

an inference rule for sugared assignments. As expected, this derived rule includes a condition corresponding to the good variable requirement of specification logic. Let l be a reference expression of type θ_1 , and let $e[l\uparrow]$ be a reference expression of type θ_2 . Let $(e[l\uparrow] := e_1[l\uparrow]; t)$ be a sugared term with $l\uparrow$ being the only instance of sugaring. By the definition of desugaring, this is equivalent to

$$\text{get } x \leftarrow l \text{ in } e[x] := e_1[x]; t$$

Theorem 25 *The following inference rule can be derived for a sugared assignment term:*

$$\frac{\begin{array}{c} (x : \theta_1) \\ \alpha_1[x] \text{ implies } (e[x] \sim (e[l\uparrow] = e[x])) \end{array} \quad \begin{array}{c} (x : \theta_1) \\ (y : \theta_2) \\ \alpha_1[x] \text{ implies } (e[x] \sim \alpha_2[y]) \end{array} \quad t : \alpha_2[e[l\uparrow]\uparrow]}{(e[l\uparrow] := e_1[l\uparrow]; t) : (\alpha_1[l\uparrow] \text{ implies } \alpha_2[e_1[l\uparrow]])}$$

Proof: From the first two antecedents, use assertion logic to derive:

$$\frac{(x : \theta_1) \quad (y : \theta_2)}{\alpha_1[x] \text{ implies } (e[x] \sim ((e[l\uparrow] = e[x]) \text{ implies } \alpha_2[y]))}$$

Now, note the following deduction:

$$\frac{\begin{array}{c} (x : \theta_1) \\ \{\alpha_1[x]\} \\ (y : \theta_2) \end{array} \quad \frac{(x : \theta_1) \quad t : \alpha_2[e[l\uparrow]\uparrow]}{t : (e[l\uparrow] = e[x] \text{ implies } \alpha_2[e[l\uparrow]\uparrow])}}{\{e[x] \sim (e[l\uparrow] = e[x] \text{ implies } \alpha_2[y])\} \quad t : (e[l\uparrow] = e[x] \text{ implies } \alpha_2[e[x]\uparrow])}$$

$$\frac{(e[x] := e_1[x]; t) : \{e[l\uparrow] = e[x] \text{ implies } \alpha_2[e_1[x]]\}}{(e[x] := e_1[x]; t) : \{\alpha_1[x] \text{ implies } e[l\uparrow] = e[x] \text{ implies } \alpha_2[e_1[x]]\}}$$

$$\frac{(e[x] := e_1[x]; t) : \alpha_1[x] \text{ implies } e[l\uparrow] = e[x] \text{ implies } \alpha_2[e_1[x]]}{(\text{get } x \leftarrow l \text{ in } e[x] := e_1[x]; t) : \alpha_1[l\uparrow] \text{ implies } e[l\uparrow] = e[l\uparrow] \text{ implies } \alpha_2[e_1[l\uparrow]])}$$

$$(\text{get } x \leftarrow l \text{ in } e[x] := e_1[x]; t) : \alpha_1[l\uparrow] \text{ implies } \alpha_2[e_1[l\uparrow]]$$

□

There are two interesting features of this derived rule. First, there are two noninterference conditions involved. While the second one is similar to that in the rule **Assignment**, the first one ensures that the reference being assigned remains invariant through the assignment.

This condition plays the role of the good variable condition of specification logic. The second interesting feature of the derived rule is that we allow the inference to accumulate a precondition $\alpha_1[l \uparrow]$ which is not present in the type of t . To motivate this, assume that A is an array (reference-valued function), and attempt

$$\frac{t : (A(j \uparrow) \uparrow > 0)}{(A(i \uparrow) := 0 ; t) : (A(j \uparrow) \uparrow > 0)}$$

Noninterference requires the condition $\neg(i \uparrow =_{\theta} j \uparrow)$ to be true for all states, which is clearly impossible. Our derived rule allows state dependent conditions to be used in proving noninterference, thereby establishing state dependent noninterference.

4.8.4 Example: Queues

One of the major benefits of imperative languages over purely applicative languages is the ability to modify the content of references and observe these modifications via different access paths. Reasoning about such *shared* modifications is one of our goals in the formulation of OTT. Naturally, programs that use sharing are harder to reason about than those which do not use sharing. So, we believe it is important to formulate useful abstractions to facilitate such reasoning.

If we are trying to show

$$(a := e ; c) : C[b \uparrow]$$

we have to know the relationship between a and b . If $a = b$ in all states, then the assignment is a shared modification; it is observable via both a and b . We can then replace $C[b \uparrow]$ by $C[a \uparrow]$ using the equality, and prove the latter type. If $\neg(a =_{\theta} b)$ in all states, then a is noninterfering with the occurrences of $b \uparrow$ in C . If all other references are similarly noninterfering, rule *Assignment* can be used. If, on the other hand, the truth of $a =_{\theta} b$ is state dependent, the *Assignment* rule is not applicable. It may be possible to prove $C[b \uparrow]$ by cases or by using the *Assignment* rule of assertion logic. However, it is preferable to formulate programs such that this situation does not arise.

We illustrate these issues by verifying the *insert* operation in the “linked list” representation of queues that was presented in Section 2.7. The representation involves two pointers *front* and

$rear$ pointing to the two ends of the list, so that modifications to $rear \uparrow$ are also observable by references accessible from $front$. The essence of our argument is that we are mainly interested in the values lying *between* the nodes $front \uparrow$ and $rear \uparrow$; these are not affected by modifications to $rear \uparrow$, even though both $front \uparrow$ and $rear \uparrow$ are affected by such modifications.

Figure 4.13 shows the representation fragment. **List** is the type of applicative lists of type S . **Nil** represents the empty list, $a :: q$ is the usual “cons” operation on lists, and $q @ q'$ is the usual “append” operation on lists. **Lnklist** is the type of *linked* lists, with the second component of every pair being a reference to the next pair. A queue is represented by two pointers $front$ and $rear$ that point to the front and rear of a linked list. The linked list always contains a dummy header at its front. $f \mapsto r$ asserts that the linked list node r is accessible from f by following links. If $f \mapsto r$ holds, then $q\text{-refs}(f, r)$ is the set of references in the nodes between f and r , while $q\text{-val}(f, r)$ is the applicative list of values in those nodes. Note that $q\text{-refs}(f, r)$ does not include the reference in node r and $q\text{-val}(f, r)$ does not include the value in node f . This is because if f and r are the first and last nodes in a queue’s representation, then the value in the first (header) node and the reference in the last node are not significant to the representation. Note that Type_θ denotes the type of all “small” θ types (see Sections 4.9 and 5.3.2 for discussions on such universes of types). $\text{noninterfere-qrefs}(front, rear, T)$ asserts that none of the references in the queue representation interfere with the type T . **DRT** (distinct reference table) is an abstraction representing the structure of the store. It asserts that various collections of references are disjoint. This information is used to prove noninterference obligations. In the figure, we present the definition of **DRT** for two arguments; this can be generalized to handle multiple arguments. $\text{GQ}(front, rear)$ captures all essential properties of the representation of queues, and certifies that $\langle front, rear \rangle$ is a good queue.

The implementation of queues was presented in Section 2.7. We reproduce the implementation of the insertion operation (we have labeled the subterms with labels [1], [2], and [3], and shall subsequently refer to these subterms by their labels):

$$\begin{aligned} \text{insert} = \lambda v. \lambda c. & \text{ [3] } \text{letref } rnew : \text{Ref Lnklist } S := \text{Nil in} \\ & \text{ [2] } rear \uparrow .2 := \text{Cons}(v, rnew); \\ & \text{ [1] } rear := rear \uparrow .2 \uparrow; c \end{aligned}$$

This operation takes a value v and an observation c , modifies $rear$ to achieve the effect of inserting v , and then evaluates the observation c . In the remainder of this section, we show

datatype List $S = \text{Nil} \mid :: \text{ of } (S \times \text{List } S)$

$@ : \text{List} \times \text{List} \rightarrow \text{List}$

$\text{Nil} @ q \stackrel{\text{def}}{=} q$

$(a :: q) @ q' \stackrel{\text{def}}{=} a :: (q @ q')$

datatype Lnklist $S = \text{Nil} \mid \text{Cons of } (S \times \text{Ref Lnklist } S)$

Let $q \in \text{Lnklist } S$ such that $q \neq \text{Nil}$. Then,

$q.1 \stackrel{\text{def}}{=} \text{case } q \text{ of Nil} \Rightarrow \text{abort}(0)$
 $\quad \quad \quad \mid \text{Cons}(a, q') \Rightarrow a$

$q.2 \stackrel{\text{def}}{=} \text{case } q \text{ of Nil} \Rightarrow \text{abort}(0)$
 $\quad \quad \quad \mid \text{Cons}(a, q') \Rightarrow q'$

datatype Queue $S = \text{Ref Lnklist } S \times \text{Ref Lnklist } S$

$\mapsto : \text{Lnklist } S \times \text{Lnklist } S \rightarrow \text{Obs Prop}$

$(f \mapsto r) \stackrel{\text{def}}{=} (f = r) \text{ or } ((f \neq \text{Nil}) \text{ and } (f.2 \uparrow \mapsto r))$

$\text{q-refs} : \text{Lnklist } S \times \text{Lnklist } S \rightarrow \text{Obs Type}_\theta$

$\text{q-refs}(f, r) \stackrel{\text{def}}{=} \{x : \text{Ref Lnklist } S \mid$
 $\quad (\exists y : \text{Lnklist } S)(f \mapsto y) \text{ and } (y \mapsto r) \text{ and } (y \neq r) \text{ and } (x = y.2)\}$

$\text{q-val} : \text{Lnklist } S \times \text{Lnklist } S \rightarrow \text{Obs List } S$

$\text{q-val}(f, r) \stackrel{\text{def}}{=} \text{if } (f = r) \text{ then Nil else } (f.2 \uparrow .1) :: \text{q-val}(f.2 \uparrow, r)$

$\text{noninterfere-qrefs} : \text{Ref Lnklist } S \times \text{Ref Lnklist } S \times \text{Type}_\omega \rightarrow \text{Obs Prop}$

$\text{noninterfere-qrefs}(fp, rp, T) \stackrel{\text{def}}{=} (fp \sim T) \text{ and } (rp \sim T) \text{ and}$
 $\quad (\forall l : \text{q-refs}(fp \uparrow, rp \uparrow))(l \sim T) \text{ and } (rp \uparrow .2 \sim T)$

$\text{DRT} : \text{Obs Type}_\theta \times \text{Obs Type}_\theta \rightarrow \text{Obs Prop}$

$\text{DRT}(T_1, T_2) = (\text{All } l : T_1)(\text{All } l' : T_2) \neg (l =_\theta l')$

$\text{GQ} : \text{Ref Lnklist } S \times \text{Ref Lnklist } S \rightarrow \text{Obs Prop}$

$\text{GQ}(fp, rp) \stackrel{\text{def}}{=} (fp \uparrow \mapsto rp \uparrow) \text{ and } (rp \uparrow \neq \text{Nil}) \text{ and } \text{DRT}(\{fp\}, \{rp\}, \text{q-refs}(fp \uparrow, rp \uparrow), \{rp \uparrow .2\})$

Figure 4.13: Properties of the queue representation.

that insert has the following type:

$$\begin{aligned} \text{insert} : (\Pi v: S) \quad & (\text{GQ}(\text{front}, \text{rear}) \text{ and noninterfere-qrefs}(\text{front}, \text{rear}, T) \\ & \text{and } (\text{q-val}(\text{front}\uparrow, \text{rear}\uparrow) = q @ (v :: \text{Nil})) \text{ implies Obs } T) \\ \rightarrow \quad & (\text{GQ}(\text{front}, \text{rear}) \text{ and noninterfere-qrefs}(\text{front}, \text{rear}, T) \\ & \text{and } (\text{q-val}(\text{front}\uparrow, \text{rear}\uparrow) = q) \text{ implies Obs } T) \end{aligned}$$

Let $x, y, z \in \text{Lnklist } S$. We assume the following lemmas.

1. $(x \neq \text{Nil}) \Rightarrow (x \mapsto x.2\uparrow)$
2. $(x \mapsto y) \text{ and } (y \mapsto z) \text{ implies } (x \mapsto z)$
3. $(x \neq \text{Nil}) \text{ and } (x.2\uparrow \neq \text{Nil}) \text{ and } (x \neq x.2\uparrow) \text{ implies } \text{q-val}(x, x.2\uparrow) = (x.2\uparrow .1 :: \text{Nil})$
4. $\text{q-val}(x, y) @ \text{q-val}(y, z) = \text{q-val}(x, z)$
5. $\text{q-refs}(x, y) \cup \text{q-refs}(y, z) = \text{q-refs}(x, z)$
6. $(x \neq \text{Nil}) \Rightarrow \text{q-refs}(x, x.2\uparrow) = \{x.2\}$
7. $(p \notin \text{q-refs}(x, y)) \text{ implies } (p \sim x \mapsto y)$
8. $(p \notin \text{q-refs}(x, y)) \text{ implies } (p \sim \text{q-refs}(x, y))$

Assume:

$$\begin{aligned} c : (\text{GQ}(\text{front}, \text{rear}) \text{ and noninterfere-qrefs}(\text{front}, \text{rear}, T) \\ \text{and } (\text{q-val}(\text{front}\uparrow, \text{rear}\uparrow) = q @ v :: \text{Nil}) \text{ implies Obs } T) \end{aligned}$$

The proof proceeds by applying rule *Assignment* twice, and then applying rule *Creation*. The assignments result in a host of noninterference obligations; these are proved using the distinct reference table abstraction. Expanding the definitions of GQ and noninterfere-qrefs, we have that:

$$\begin{aligned} c : (\text{front}\uparrow \mapsto \text{rear}\uparrow) \text{ and } (\text{rear}\uparrow \neq \text{Nil}) \\ \text{and DRT}(\{\text{front}\}, \{\text{rear}\}, \text{q-refs}(\text{front}\uparrow, \text{rear}\uparrow), \{\text{rear}\uparrow .2\}) \\ \text{and } (\text{front} \sim T) \text{ and } (\text{rear} \sim T) \\ \text{and } (\forall l : \text{q-refs}(\text{front}\uparrow, \text{rear}\uparrow))(l \sim T) \text{ and } (\text{rear}\uparrow .2 \sim T) \\ \text{and } (\text{q-val}(\text{front}\uparrow, \text{rear}\uparrow) = q @ v :: \text{Nil}) \\ \text{implies Obs } T \end{aligned}$$

Then, by rule *Assignment*:

$$\begin{aligned}
[1] : & (front \uparrow \mapsto rear \uparrow .2 \uparrow) \text{ and } (rear \uparrow .2 \uparrow \neq \text{Nil}) \\
& \text{and DRT}(\{front\}, \{rear\}, \text{q-refs}(front \uparrow, rear \uparrow .2 \uparrow), \{rear \uparrow .2 \uparrow .2\}) \\
& \text{and } (front \sim T) \text{ and } (rear \sim T) \\
& \text{and } (\forall l : \text{q-refs}(front \uparrow, rear \uparrow .2 \uparrow))(l \sim T) \text{ and } (rear \uparrow .2 \uparrow .2 \sim T) \\
& \text{and } (\text{q-val}(front \uparrow, rear \uparrow .2 \uparrow) = q @ v :: \text{Nil}) \\
& \text{implies Obs } T
\end{aligned}$$

This simplifies to:

$$\begin{aligned}
[1] : & (front \uparrow \mapsto rear \uparrow) \text{ and } (rear \uparrow \neq \text{Nil}) \text{ and } (rear \uparrow .2 \uparrow \neq \text{Nil}) \\
& \text{and DRT}(\{front\}, \{rear\}, \text{q-refs}(front \uparrow, rear \uparrow), \{rear \uparrow .2\}, \{rear \uparrow .2 \uparrow .2\}) \\
& \text{and } (front \sim T) \text{ and } (rear \sim T) \\
& \text{and } (\forall l : \text{q-refs}(front \uparrow, rear \uparrow))(l \sim T) \text{ and } (rear \uparrow .2 \sim T) \text{ and } (rear \uparrow .2 \uparrow .2 \sim T) \\
& \text{and } \text{q-val}(front \uparrow, rear \uparrow) = q \text{ and } rear \uparrow .2 \uparrow .1 = v \\
& \text{implies Obs } T
\end{aligned}$$

Again, by rule *Assignment*:

$$\begin{aligned}
[2] : & (front \uparrow \mapsto rear \uparrow) \text{ and } (rear \uparrow \neq \text{Nil}) \text{ and } (\text{Cons}(v, rnew) \neq \text{Nil}) \\
& \text{and DRT}(\{front\}, \{rear\}, \text{q-refs}(front \uparrow, rear \uparrow), \{rear \uparrow .2\}, \{\text{Cons}(v, rnew).2\}) \\
& \text{and } (front \sim T) \text{ and } (rear \sim T) \\
& \text{and } (\forall l : \text{q-refs}(front \uparrow, rear \uparrow))(l \sim T) \text{ and } (rear \uparrow .2 \sim T) \text{ and } (\text{Cons}(v, rnew).2 \sim T) \\
& \text{and } \text{q-val}(front \uparrow, rear \uparrow) = q \text{ and } \text{Cons}(v, rnew).1 = v \\
& \text{implies Obs } T
\end{aligned}$$

which simplifies to:

$$\begin{aligned}
[2] : & (front \uparrow \mapsto rear \uparrow) \text{ and } (rear \uparrow \neq \text{Nil}) \\
& \text{and DRT}(\{front\}, \{rear\}, \text{q-refs}(front \uparrow, rear \uparrow), \{rear \uparrow .2\}, \{rnew\}) \\
& \text{and } (front \sim T) \text{ and } (rear \sim T) \\
& \text{and } (\forall l : \text{q-refs}(front \uparrow, rear \uparrow))(l \sim T) \text{ and } (rear \uparrow .2 \sim T) \text{ and } (rnew \sim T) \\
& \text{and } \text{q-val}(front \uparrow, rear \uparrow) = q \\
& \text{implies Obs } T
\end{aligned}$$

We now apply the rule *Creation* to create the reference *rnew*. This permits us to assume that *rnew* is distinct from all other references in existence.

Thus, by rule *Creation*:

$$\begin{aligned}
[3] : & (front \uparrow \mapsto rear \uparrow) \text{ and } (rear \uparrow \neq \text{Nil}) \\
& \text{and DRT}(\{front\}, \{rear\}, \text{q-refs}(front \uparrow, rear \uparrow), \{rear \uparrow .2\}) \\
& \text{and } (front \sim T) \text{ and } (rear \sim T) \\
& \text{and } (\forall l : \text{q-refs}(front \uparrow, rear \uparrow))(l \sim T) \text{ and } (rear \uparrow .2 \sim T) \\
& \text{and } \text{q-val}(front \uparrow, rear \uparrow) = q \\
& \text{implies Obs } T
\end{aligned}$$

which simplifies to:

$$\begin{aligned}
[3] : & \text{GQ}(front, rear) \text{ and noninterfere-qrefs}(front, rear, T) \\
& \text{and } (\text{q-val}(front \uparrow, rear \uparrow) = q) \text{ implies Obs } T
\end{aligned}$$

as desired.

The noninterference obligations for the first application of rule *Assignment* are:

$$\begin{aligned}
& \neg(rear =_{\theta} front) \\
& (rear \sim \text{q-refs}(front \uparrow, x)) \\
& (rear \sim \text{q-val}(front \uparrow, x) = q @ v :: \text{Nil})
\end{aligned}$$

for all $x \in \text{Ref Lnklist } S$. Assertion logic permits us to use information about the current state to prove these. Specifically, we can use the distinct reference table (DRT) to prove that the reference $rear$ is distinct from all references that are dereferenced in the above types.

The noninterference obligations for the second application of rule *Assignment* are:

$$\begin{aligned}
& \neg(rear \uparrow .2 =_{\theta} front) \\
& \neg(rear \uparrow .2 =_{\theta} rear) \\
& (rear \uparrow .2 \sim \text{q-refs}(front \uparrow, rear \uparrow)) \\
& (rear \uparrow .2 \sim (\text{q-val}(front \uparrow, rear \uparrow) = q))
\end{aligned}$$

These again follow from the DRT assumptions.

This example demonstrates how the creation rule provides us with information about the distinctness of references, while the assignment rule creates noninterference obligations that are proved using this information. The distinct reference table (DRT) is a convenient abstraction that captures this information.

4.9 Predicativity

An *impredicative* definition constructs an object by combining a collection of objects that includes the object being constructed. Thus an impredicative notion depends on itself for its definition, leading to the possibility of a “vicious circularity”. For example, a type of all types is an impredicative notion: types are defined in terms of their membership and equality relations; a type of all types contains itself as a member, and hence its definition involves combining a collection of objects that includes itself. Recursive definitions are also impredicative, as they are clearly circular (although not viciously so).

It is well-known that impredicativity has the potential of leading to paradoxes. Russell’s paradox is famous for showing this in set theory. The paradox goes as follows: consider the set of all sets that do *not* include themselves; does this set include itself? Type theory is not immune from this, and indeed, Martin-Lof’s type theory becomes inconsistent when a type of all types is added to it. Hence, it is defined to be a predicative theory, i.e. no type is defined in terms of a collection of types that includes itself. The theory is organized as an infinite hierarchy of universes, each of which is a collection of types (see Section 5.3.2). The hierarchy is cumulative: every universe is a proper subset of the next higher universe. The hierarchy is reflective: every universe is also a type that is a member of the next higher universe. The hierarchy is predicative: whenever an object’s definition quantifies over the types in a universe, the object resides in a higher universe.

The Nuprl type theory is a variant of Martin-Lof’s type theory and has the same universe structure. However, Nuprl contains several new type constructors, including inductive type constructors. These constructors take the least (and greatest) fixpoints of recursive type equations. As usual, these are defined by modeling universes as lattices and taking the least upper bounds of the appropriate chains. These constructors quantify over collections of types that include the types being defined, and hence are impredicative. However, these constructors can be shown to be consistent with the rest of the theory [52].

ILC and OTT’s type systems contain a type constructor $\mathbf{Obs} T$. $\mathbf{Obs} T$ contains terms (called *observers*) that observe the state and return values of type T . $\mathbf{Obs} T$ may thus be viewed as an implicit, polymorphic, function-space type from the type of a state to the type T . That is, if \mathbf{State} is the type of all state types, then $\mathbf{Obs} T \equiv (\forall \text{St} : \mathbf{State})(\text{St} \rightarrow T)$. However,

a state type \mathbf{St} is an arbitrary product of reference types; if $\mathbf{Ref\ Obs\ } T$ were a well-formed type, then a state might include references of this type. Thus the definition of $\mathbf{Obs\ } T$ would depend on the definition of $\mathbf{Ref\ Obs\ } T$, which in turn would depend on the definition of $\mathbf{Obs\ } T$. We may conclude that $\mathbf{Obs\ } T$ is an impredicative type constructor, and has the possibility of making the theory inconsistent. This is borne out by our observation (see Section 2.5.2) that the type $\mathbf{Ref\ Obs\ } T$ results in the failure of strong normalization, a property that is the testbed of consistency in type theory.

There are two possible remedies to this situation. First, we might prohibit types like $\mathbf{Ref\ Obs\ } T$ from being well-formed types; this corresponds to saying that observers are not storable values. This is the approach we have adopted in this thesis, where we use a layered type system to syntactically eliminate types of the form $\mathbf{Ref\ Obs\ } T$, thus eliminating the circularity. Alternatively, we might prohibit $\mathbf{Obs\ } T$ from being a well-formed type, and require instead a predicative type constructor. With this approach, we would replace the impredicative type constructor $\mathbf{Obs\ } T$ with a family of predicative type constructors $\mathbf{Obs}_i\ T$. $\mathbf{Obs}_i\ T$ is the type of observers that can only observe references that are well-typed at universe level i . As before, the observers return a value of type T . Since the type $\mathbf{Obs}_i\ T$ quantifies over types in universe i , it is made to reside in a universe higher than i . It no longer depends on prior knowledge of itself, and is a predicative type. We do not pursue this idea further in this thesis.

Chapter 5

Constructive Type Theory (CTT)

5.1 Introduction

Type theory is a rapidly developing field that, although still in its infancy, has already had a considerable impact on programming language research. Type theory has been used to formalize intuitionistic logic [45, 46] and has influenced the design of numerous functional and procedural languages [4, 6, 16, 59]. It has been used to model polymorphism [65, 56] and data abstraction [5, 43, 57] and to reason about programs [6, 9]. It has even found use in the study of semantics [2, 51, 75].

This chapter is not intended to be a tutorial on type theory; for such expositions, we direct the reader to the literature [26, 3, 47, 6]. The purpose of this chapter is to provide an *overview* of the important concepts underlying a Martin-Lof-style type theory [46] and to describe the language of such a theory.

5.2 Type Systems and Type Theories

Type systems are formal systems for assigning types to expressions of a programming language. Their purpose is either to specify the context sensitive syntax of typed programming languages, or to impose constraints on untyped programming languages [56, 55]. Type systems may be interpreted either *mathematically* or *computationally*. A mathematical interpretation treats types as collections of *mathematical* objects; it does not distinguish between mathematically equivalent but computationally distinct terms. A computational interpretation treats types

as collections of untyped *computational* terms; this permits reasoning about computational properties of the terms. Type systems for programming languages always have a computational interpretation, but do not necessarily have a mathematical interpretation (eg. the Nuprl type theory) [74].

Type systems vary widely in their richness, spanning the range from the simply typed lambda calculus to extremely rich constructive type theories. Constructive type theories (hereafter abbreviated to type theories) are type systems rich enough to serve as programming logics for the underlying languages. These theories include equality types that axiomatize equality of terms, and structured types that correspond to formulas of constructive logic.

Some of the more prominent type theories are: de Bruijn’s AUTOMATH [12], Scott’s theory of constructive validity [72], Stenlund’s Theory of Species [76], Martin-Lof’s intuitionistic type theories [45, 46, 47], Constable’s Nuprl type theory [6], Coquand and Huet’s Calculus of Constructions [8], Feferman’s theory T_0 [18], Hayashi’s PX [28], and Henson’s TK5 [30, 29]. Of these, Martin-Lof’s intuitionistic type theory has been by far the most influential. In the remainder of this chapter, we review the basic concepts of Martin-Lof’s type theory (MLTT) and the similar Nuprl type theory. These theories are much more powerful than the simple constructive type theory described in Section 4.3.

5.3 Constructive Type Theory

Figure 5.1 presents the abstract syntax of terms of the language. Terms may be *canonical* or *noncanonical*. Canonical terms are the “values” of the language; noncanonical terms define computations and are the “programs” of the language. The abstract syntax of terms that are *canonical types* has been factored out into Figure 5.2. This separation is merely to enhance the presentation — otherwise, types are terms with the same status as all other terms. In the syntax, the metavariable x ranges over variables, n ranges over the natural numbers, the e_i range over terms, S, T range over terms that evaluate to types, and CT ranges over canonical types.

Let $V(e)$ be the set of free variables in term e . $V(e)$ is defined by induction on terms in Appendix A.

Canonical	
x	variable
n	natural number (constant)
CT	canonical type
$\text{inl}(e)$	inject left
$\text{inr}(e)$	inject right
$\langle e_1, e_2 \rangle$	pair
$\lambda x.e$	abstraction
fact	unit
Noncanonical	
abort (e)	error/abort
$e_1 \text{ op } e_2$	$+$, $-$, $*$, $/$ or mod
ind ($e_1; e_2; x, y.e_3$)	primitive induction
case e_1 of $\text{inl}(x) \Rightarrow e_2 \mid \text{inr}(x) \Rightarrow e_3$ end	union tag discrimination
let $\langle x, y \rangle = e_1$ in e_2	pair component selection
$e_1(e_2)$	function application
letrec $f = \lambda x.e_1$ in $f(e_2)$	recursion
if $e_1 = e_2$ then e_3 else e_4	equality conditional
if $e_1 < e_2$ then e_3 else e_4	less-than conditional

Figure 5.1: Abstract syntax of terms.

void	empty
nat	natural numbers
U_n	universe
$S \mid T$	disjoint union
$S \times T$	cartesian product
$S \rightarrow T$	function space
$(\Sigma x: S)T_x$	dependent product
$(\Pi x: S)T_x$	dependent function
$\{x: S \mid T_x\}$	subset
$(\mu x: U_n)T_x$	inductive type
$(e_1 = e_2 \text{ in } T)$	equality
$(e_1 < e_2)$	less-than

Figure 5.2: Abstract syntax of canonical types.

5.3.1 Typing and Computation Rules

Appendix A contains the type-inference and computation rules of the type system. For each type constructor, we include

1. a *formation* rule to describe when a type is well-formed;
2. *introduction* rules to construct members of the type;
3. *congruence* rules to define equality on members of the type;
4. *elimination* rules to compute with members of the type; and
5. *computation* rules to define the operational semantics of computation.

5.3.2 Informal Description

Empty type (void)

`void` has no members. If e is a member of `void`, then `abort(e)` is a term representing an error (since there *is* no member of `void`). Hence, `abort(e)` has the operational effect of aborting the entire computation with no final result.

Natural number (nat)

`nat`'s canonical members are the numbers $0, 1, 2, \dots$. The usual operations of addition, subtraction, multiplication, integer division and modulo (remainder of integer division) are provided. One computes with numbers via primitive recursion using the construct `induct`. This construct translates into ML's recursive "`letrec`" construct as follows:

$$\text{ind}(e_1; e_2; x, y.e_3) \equiv (\text{letrec } f = (\lambda x. \text{if } x = 0 \text{ then } e_2 \text{ else } e_3[f(x-1)/y]) \text{ in } f(e_1))$$

Thus, it first evaluates e_1 to a natural number, say x . If x is zero, it returns the value of e_2 . Otherwise, it returns the value of e_3 , with y bound to the recursive result of the construct at $(x-1)$. The `if-then-else` conditionals represent the fact that integer comparison is decidable.

Universe (U_i)

Types are ordinary terms of the language. Hence we need to indicate what the type of a type is. There are two prominent approaches to this:

1. Impredicative approach: We can define a type of all types. This approach is unsuited for Martin-Lof style type theories, as it makes the theories inconsistent [24].
2. Predicative approach: We can define a “cumulative hierarchy of universes”. That is, define a family U_1, U_2, \dots of universes. Each universe is a type (i.e. collection) of some types, and each universe U_i is strictly contained in all higher universes U_{i+j} . This is the approach taken by Martin-Lof.

Thus, U_i is the i th universe type. It contains, as canonical members, all types constructible from the universe types U_1 through U_{i-1} , the primitive types (`nat` and `void`), and the type constructors. U_1 is called the universe of small types, while U_2 includes first-order polymorphic types. No special computations are defined on canonical types.

Disjoint Union ($S \mid T$)

Members of $S \mid T$ are either members of S or members of T , the members being tagged to indicate which of S or T they come from. Thus if $s \in S$ and $t \in T$, then $\text{inl}(s), \text{inr}(t) \in S \mid T$. The terms `inl` and `inr` stand for “inject left” and “inject right” respectively. The case statement permits conditional computation based on the tag of a term. `case e_1 of $\text{inl}(x) \Rightarrow e_2 \mid \text{inr}(x) \Rightarrow e_3$ end` evaluates to either e_2 or e_3 depending on the tag of the value e_1 . If e_1 is tagged with `inl`, then it evaluates to e_2 with x bound to the value of e_1 ; otherwise it evaluates to e_3 with x bound to the value of e_1 .

Cartesian Product ($S \times T$)

Members of $S \times T$ are ordered pairs $\langle s, t \rangle$ of members s of S and t of T . Two ordered pairs are equal if their corresponding components are equal. The computational “let” construct permits a program to access the individual components of a pair. `(let $\langle x, y \rangle = e_1$ in e_2)` evaluates to e_2 with x bound to the first component of the pair e_1 , and y bound to the second component.

Function Space ($S \rightarrow T$)

Members of $S \rightarrow T$ are total functions $\lambda x.t$ from type S to type T . Functions are used in computations by applying them to values. $f(e)$ evaluates f to a function and returns the value of this function at e . Equality on functions is defined extensionally. That is, two functions are equal if they yield equal values when applied to equal arguments.

Dependent Product ($(\Sigma x:S)T$)

Members of $(\Sigma x:S)T$ are ordered pairs $\langle s, t \rangle$ of members of s of S and t of $T[s/x]$. For example, $(\Sigma n:\text{nat})(\text{if even}(n) \text{ then nat else void})$ is the type of pairs of natural numbers with even first components. The dependent product type is a generalization of the cartesian product type.

Dependent Function Space ($(\Pi x:S)T$)

Members of $(\Pi x:S)T$ are total functions $\lambda x.t$ from type S to type T . The range type T depends on the argument to the function. Thus $f \in (\Pi x:S)T$ iff for all $s \in S$, $f(s) \in T[s/x]$. For example, $(\Pi T:U_1)(T \rightarrow T)$ is the type of explicitly polymorphic functions $\lambda T.\lambda t.b$ that take as arguments a type T and a term t of type T , and return as result $b[T, t]$ of type T . The dependent function space type is a generalization of the function space type.

Subset ($\{x:S \mid T\}$)

Members of type $\{x:S \mid T\}$ are members s of type S such that the type $T[s/x]$ is inhabited. This corresponds to forming a subset of type S . Equality and computation is the same as for members of type S .

Inductive type ($(\mu x:U_i)T$)

Members of $(\mu x:U_i)T$ are members of the “unrolled” type $T[(\mu x:U_i)T/x]$. This corresponds to the least fixpoint of the type equation $x = T$. A recursive “**letrec**” construct is provided to compute recursively with these inductive objects. (**letrec** $f = \lambda x.e_1$ **in** $f(e_2)$) defines a recursive function $f = \lambda x.e_1$ and returns f 's value at e_2 .

Equality type ($(s = t \text{ in } T)$)

If s and t are equal members of type T , then the term $(s = t \text{ in } T)$ is a type that contains the single member “**fact**”; otherwise, this term is not a well-formed type. No computations are defined on the term **fact**, since this term does not have any computational significance.

Less-than type ($s < t$)

If s and t compute to natural numbers with s less than t , then $s < t$ is a type that contains the single member “**fact**”. Otherwise, this term is not a well-formed type. Again, no computations are defined on the term **fact**.

5.4 The Propositions-as-Types Principle

A fundamental axiom of classical logic is the “law of the excluded middle”. This says that $(\theta \text{ or } \neg\theta)$ holds for all propositions θ . Intuitionistic (constructive) logic ¹ rejects the law of the excluded middle as a valid axiom. It considers a mathematical object to exist only if a procedure for constructing and exhibiting the object is given. For example, to show a set to be inhabited, one must construct an element of the set. One cannot merely show that it is contradictory for the set to be empty – the proof-by-contradiction method of proof is not valid.

Martin-Lof’s type theory treats a type as the intuitionistic set of all its members, and it treats an intuitionistic proposition as the intuitionistic set of all its proofs. Thus, it identifies a proposition P with the type of all proofs of P . Conversely, it identifies a type T with the proposition that the type T is inhabited. This is not merely a simple analogy; rather, it is an isomorphism between propositions in constructive logic and types of a programming language. This principle was first observed by Curry [10], developed by Howard [33], and used by DeBruijn [11]. It is now known as the “propositions-as-types principle” or the “Curry-Howard isomorphism” [7].

This isomorphism is used in an essential way to embed a programming logic within type theory. Figure 5.3 describes the intuitionistic meaning of the standard logical connectives. Figure 5.4 describes how the types of the previous section may be interpreted as intuitionistic

¹We use the words *intuitionistic* and *constructive* interchangeably.

Connective	Notation	Intuitionistic meaning
conjunction	$P \wedge Q$	there is evidence for both P and Q.
disjunction	$P \vee Q$	there is evidence for either P or Q; (we should be able to tell from the evidence which of P or Q it supports).
negation	$\neg P$	there is no evidence for P; that is, P can never be proved.
implication	$P \Rightarrow Q$	there is a method for transforming any evidence for P into evidence for Q.
existential quantification	$\exists x:T.P_x$	we can find an element v of type T such that there is evidence for $P[v/x]$.
universal quantification	$\forall x:T.P_x$	given any element v of type T , we can find evidence for $P[v/x]$.

Figure 5.3: Intuitionistic meaning of logical connectives.

Name of formula	Formula	Type	Name of type
absurdity	false	void	empty
truth	true	nat	any inhabited type
set of propositions	Prop	U_1	first universe
conjunction	$P \wedge Q$	$P \times Q$	cartesian product
disjunction	$P \vee Q$	$P \mid Q$	disjoint union
implication	$P \Rightarrow Q$	$P \rightarrow Q$	function space
negation	$\neg P$	$P \rightarrow \mathbf{void}$	
existential quantification	$\exists x:T.P_x$	$(\Sigma x:T)P_x$	dependent product
universal quantification	$\forall x:T.P_x$	$(\Pi x:T)P_x$	dependent function space

Figure 5.4: Propositions as types.

formulas. This has two major consequences. First, it permits type theory to be a foundation for doing (constructive) mathematics [45]. Second, it permits type theory to be a foundation for doing computer science [46, 6].

To elaborate on the second consequence, the duality between type theory and constructive logic provides for an interesting style of program development: if the formula corresponding to a problem specification is proved within constructive logic, then an executable program can be mechanically extracted from the proof. Based on this idea, program development systems [61, 6, 79, 8, 28, 30] have been implemented for several type theories.

5.5 Notation

We have seen how the type system of Section 5.3 may be interpreted as a constructive logic. In subsequent chapters, we shall freely use the logical connectives of Figure 5.4, with the understanding that these are merely notational variants of the corresponding types.

In addition, we shall also use the following derived notation:

$$\begin{aligned}
 e_1 =_T e_2 &\equiv (e_1 = e_2 \text{ in } T) \\
 (\mathbf{a11} \ x: A \mid P_x) B_x &\equiv (\Pi x: \{x: A \mid P_x\}) B_x \\
 &\text{i.e., the dependent function space from a subtype of } A \text{ to } B, \\
 &\text{with } P_x \text{ describing the domain.} \\
 \{t_1, t_2, \dots, t_n\}_T &\equiv \{x: T \mid x =_T t_1 \vee \dots \vee x =_T t_n\} \\
 &\text{i.e., the subset of type } T \text{ with members } t_1, t_2, \dots, t_n \\
 \mathbf{Type} &\equiv U_1 \\
 &\text{i.e., the type of all small types}
 \end{aligned}$$

The type theory we described does not have any built-in data types other than the natural numbers. Other data types may be defined using the inductive type constructor. For example, we can define a data type of lists as follows. $(\mathbf{List} \ T)$ is the type of lists whose elements have type T . \mathbf{Nil} represents the empty list, $a :: q$ is the usual “cons” operation on lists, and $q @ q'$ is the usual “append” operation on lists. A case construct permits conditional execution based on whether the list is empty or not. A recursive form provides primitive recursion over lists. This is based on the rule μ -elim which performs structural induction over lists, yielding a primitive recursive instance of “letrec” as the computational form.

In previous chapters, we had defined and used recursive data types based on constructors (using ML notation). Those data types are just notational variants of types definable using the inductive type constructor. As an example, we present the definition of a list data type.

$$\begin{aligned}
\mathbf{List} &\stackrel{\text{def}}{=} \lambda T:U_i.(\mu x:U_i)(\{0\}_{nat} \mid (T \times x)) \\
\mathbf{Nil} &\stackrel{\text{def}}{=} \mathbf{inl}(0) \\
a :: b &\stackrel{\text{def}}{=} \mathbf{inr}(\langle a, b \rangle) \\
(\mathbf{case } l \text{ of Nil} \Rightarrow e_1 \mid x :: y \Rightarrow e_2) &\stackrel{\text{def}}{=} \\
&\quad \mathbf{case } l \text{ of inl}(z) \Rightarrow e_1 \mid \mathbf{inr}(z) \Rightarrow (\mathbf{let } \langle x, y \rangle = z \text{ in } e_2) \mathbf{end} \\
(f(l) = \mathbf{case } l \text{ of Nil} \Rightarrow e \mid x :: y \Rightarrow e'_{x,f(y)}) &\stackrel{\text{def}}{=} \\
&\quad \mathbf{letrec } f = (\lambda l. \mathbf{case } l \text{ of Nil} \Rightarrow e \mid x :: y \Rightarrow e') \text{ in } f(l) \\
l @ l' &\stackrel{\text{def}}{=} \mathbf{case } l \text{ of Nil} \Rightarrow l' \\
&\quad \mid x :: y \Rightarrow x :: (y @ l')
\end{aligned}$$

Chapter 6

Well-founded Orderings

6.1 Introduction

We have seen that constructive type theory defines both a programming language and its logic. Viewed computationally, type theory defines a pure functional language with a very rich type system. Viewed mathematically, type theory defines a constructive logic. This duality can be used to form the basis of a program development system. Martin-Lof's type theory has been a particularly influential type theory having spawned numerous derivative theories and program development systems.

Martin-Lof's type theory has several drawbacks as a programming language, a major drawback being that it does not provide general recursion. It only provides the restrictive form of primitive recursion for every datatype. A general recursive function is defined by the ML-like construct (`1etrec $f = \lambda x.b_{f,x}$ in e`), where $b_{f,x}$ indicates that the term b may have f and x free in it. In contrast, a primitive recursive (arithmetic) definition has the form (`1etrec $f = \lambda x.b_{f(x-1),x}$ in e`). Thus a primitive recursive b may only use $f(x - 1)$ and x to compute the value $f(x)$.

Mathematically, this is not much of a limitation since a primitive recursive definition may involve functions of higher type. Any recursive function whose termination is provable in first-order arithmetic is definable in Martin-Lof's type theory [73]. For example, Ackermann's

function [44] is a standard example of a recursive function that is not primitive recursive over first-order functions. Ackermann's function is defined as $\lambda x.\text{ack}(x, x)$ where

$$\text{ack}(m, n) \equiv \text{if } m = 0 \text{ then } n + 1 \text{ else if } n = 0 \text{ then } \text{ack}(m - 1, 1) \\ \text{else } \text{ack}(m - 1, \text{ack}(m, n - 1))$$

Although not primitive recursive, Ackermann's function can be expressed using primitive recursion over higher-order functions [68]. Let `succ` be the usual successor function and let `aug` be an auxiliary function such that $\text{aug}(f)(n) = f^{n+1}(1)$.

$$\begin{aligned} \text{succ} &\equiv \lambda x.x + 1 \\ \text{aug} &\equiv \lambda f:\text{nat} \rightarrow \text{nat}.\lambda x:\text{nat}.\text{if } x = 0 \text{ then } f(1) \text{ else } f(\text{aug}(f)(x - 1)) \\ \text{ack} &\equiv \lambda x:\text{nat}.\text{if } x = 0 \text{ then } \text{succ} \text{ else } \text{aug}(\text{ack}(x - 1)) \end{aligned}$$

Now since

$$\text{ack}(m + 1)(n) = (\text{if } m + 1 = 0 \text{ then } \text{succ} \text{ else } \text{aug}(\text{ack}(m)))(n) = \text{aug}(\text{ack}(m))(n)$$

we have

$$\begin{aligned} \text{ack}(0)(n) &= \text{succ}(n) = n + 1 \\ \text{ack}(m + 1)(0) &= \text{aug}(\text{ack}(m))(0) \\ &= \text{ack}(m)(1) \\ \text{ack}(m + 1)(n + 1) &= (\text{if } m + 1 = 0 \text{ then } \text{succ} \text{ else } \text{aug}(\text{ack}(m)))(n + 1) \\ &= \text{aug}(\text{ack}(m))(n + 1) \\ &= \text{if } n + 1 = 0 \text{ then } \text{ack}(m)(1) \text{ else } \text{ack}(m)(\text{aug}(\text{ack}(m))(n)) \\ &= \text{ack}(m)(\text{aug}(\text{ack}(m))(n)) \\ &= \text{ack}(m)(\text{ack}(m + 1)(n)) \end{aligned}$$

That is, $\lambda x.\text{ack}(x)(x)$ is Ackermann's function.

Although not a significant mathematical limitation, the restriction to primitive recursive definitions is a substantial computational and linguistic limitation that forces functions to be implemented with inefficient, inelegant, complex algorithms. For example, the above coding of Ackermann's function is far more complex than its general recursive definition. This defect is not

isolated to pathological cases, but extends to common recursive algorithms such as the quicksort algorithm [58, 59, 73]. The inefficiency and lack of expressiveness of Martin-Lof’s language have been the subject of considerable research, with numerous extensions and alternatives having been proposed [52, 75, 30].

In the previous chapter, we saw two such extensions (namely subsets and inductive types) made by Nuprl’s type theory over Martin-Lof’s type theory. In Nuprl, it is possible to construct inductive data structures and to compute recursively over their (finite) structure. In this chapter, we demonstrate how such inductive values may be used to give general-recursive definitions to some functions. The results of this chapter may be used to permit well-founded recursion over the contents of references in observation type theory (as we did, for example, in Section 4.8.2).

The remainder of this chapter proceeds as follows: we define *orderings* on types and give a type-theoretic definition of *well-foundedness* of orderings. Induction over the structure of these orderings gives us computational induction. This permits efficient, recursive programs to be extracted from constructive proofs of problem specifications. We detail constructive proofs of the well-foundedness of several classes of orderings, and demonstrate their significance through examples.

6.2 Well-founded Orderings

Let S be a type and let $<_S \in (S \rightarrow S \rightarrow \text{Prop})$ be a binary relation on S . We use infix notation for $<_S$, writing $(x <_S y)$ in place of $<_S(x)(y)$. We also use the notation $y >_S x$ interchangeably with $x <_S y$. We drop the subscript S on $<_S$ when the type S is obvious from the context.

Definition 1 An *ordering* $(S, <)$ is a binary relation $<$ on type S .

Classically, an ordering $(S, <)$ is said to be a well-ordering if either of the following conditions holds:

1. every non-empty subset of S has a least element; That is,

$$(\forall P \subset S)((P \neq \{\}) \Rightarrow (\exists a \in P)(\forall b \in P) a \leq b)$$

2. there are no infinite descending sequences of elements of S . That is,

$$\neg(\exists P \subset S)((P \neq \{\}) \wedge (\forall a \in P)(\exists b \in P) a > b)$$

The classical equivalence of these two conditions follows from De Morgan's laws.

Within intuitionistic logic, these conditions are no longer equivalent (since De Morgan's laws are not valid). More importantly, these conditions are intuitionistically inappropriate. Condition (1) is inappropriate since it implies the law of the excluded middle. To see this, consider the type $S = \{1, 2\}$ with $1 < 2$ and $\neg(2 \leq 1)$. Let S be well-founded by condition (1). Then, for an arbitrary proposition θ , the set $T \equiv \{x: \{1, 2\} \mid x = 2 \vee (x = 1 \wedge \theta)\}$ has a minimal element (by condition (1)). Let $a \in T$ be the minimal element of T ; i.e. $\forall b: T.(a \leq b)$. Since $a \in T$, we have that $a = 2 \vee (a = 1 \wedge \theta)$. We show by cases that $(\theta \vee \neg\theta)$ holds:

Case $a = 2$: Assume θ holds. Then $1 \in T$, and so $2 \leq 1$ (since $a = 2$ is the minimal element of T). But by definition of $<$, $\neg(2 \leq 1)$. We have a contradiction, and so can conclude that our assumption is false. Thus $\neg\theta$ holds (since the intuitionistic meaning of $\neg\theta$ is “the absurdity is derivable from θ ”.)

Case $(a = 1) \wedge \theta$: The case condition says that θ holds.

Now, intuitionistic logic is classical logic minus the law of the excluded middle. We have just shown that the law of the excluded middle follows intuitionistically from the assumption that the two element set S satisfies condition (1), and hence this assumption is inappropriate. Since we clearly wish S to be well-founded, condition (1) is unfit as a definition of well-foundedness.

Condition (2) is intuitionistically inappropriate since it is defined in terms of infinite objects. Further, since it is defined using negation, it does not have any meaningful computational content. Consequently, we rephrase condition (2) as an affirmative statement. Instead of saying that “there are no infinite descending sequences of elements of s ”, we say that “all descending sequences of elements of S are finite”. Since our main interest in well-orderings is to use their structure to perform induction, we use an inductive definition of finiteness of a descending sequence. Let $(S, <)$ be an ordering, and let $s, s', s_1, s_2, s_3 \dots \in S$.

Definition 2 A *descending sequence* from s_1 is a sequence s_1, s_2, s_3, \dots of elements of S such that $s_1 > s_2 > s_3 > \dots$

$$\begin{aligned}
\text{WS}(S, <, s) &\equiv (\text{all } s' : S \mid s' < s) \text{WS}(S, <, s') \\
\text{WF}(S, <) &\equiv (\forall s : S) \text{WS}(S, <, s) \\
\text{WFT} &\equiv (\exists S : \text{Type})(\exists < : (S \rightarrow S \rightarrow \text{Prop})) \text{WF}(S, <)
\end{aligned}$$

Figure 6.1: Definition of well-founded orderings.

WS-introduction

$$\frac{\Gamma, x : S \vdash e : (\text{all } y : S \mid y < x) \text{WS}(S, <, y)}{\Gamma, x : S \vdash e : \text{WS}(S, <, x)}$$

WS-elimination

$$\frac{\Gamma \vdash e_2 : S \quad \Gamma, x : S, f : ((\text{all } y : S \mid y < x) A[y]) \vdash e_1 : A[x] \quad \Gamma \vdash e_3 : \text{WS}(S, <, e_2)}{\Gamma \vdash \text{letrec } f = \lambda x. e_1 \text{ in } f(e_2) : A[e_2]}$$

Figure 6.2: Inference rules for well-founded orderings.

Definition 3 All descending sequences from s are *finite* iff all descending sequences from s' are finite for all $s' < s$.

Definition 4 An ordering $(S, <)$ is said to be *well-structured* at s (written $\text{WS}(S, <, s)$) if $s \in S$ and if all descending sequences from s are finite.

Definition 5 An ordering $(S, <)$ is said to be *well-founded* (written $\text{WF}(S, <)$) if it is well-structured at all s in S .

We represent well-foundedness within type theory using inductive types (see Figure 6.1). $\text{WS}(S, <, s)$ says that all descending sequences from s are finite if and only if all descending sequences from a smaller s' are finite. $\text{WF}(S, <)$ asserts that all descending sequences of elements of S are finite. WFT is the type of all well-founded orderings. An element of this type is an ordering $(S, <)$ together with a proof of the well-foundedness of the ordering.

The inference rules for well-founded orderings are presented in Figure 6.2. WS-introduction *constructs* a well-founded ordering by unfolding its structure (see the earlier definition of WS).

WS-elimination *uses* a well-founded ordering by inducting over its structure. This rule says that if $A[x]$ holds assuming $A[y]$ for all $y < x$, and if $(S, <)$ is a well-founded ordering, then $A[e_2]$ holds for all $e_2 \in S$.

6.3 Proving Orderings to be Well-founded

We now define several interesting classes of orderings [13, 14] and give proofs of their well-foundedness. The main significance of these proofs is that they are *constructive* proofs. We first prove the well-foundedness of the ordering $<_{nat}$ on natural numbers, since the proof method used in this simple proof carries over to other proofs. We then successively define more complicated orderings, and prove them to be well-founded based on the well-foundedness of the simpler orderings.

6.3.1 Natural Numbers

The following theorem provides the means for course-of-values induction on natural numbers.

Theorem 26 *The ordering $<_{nat} \equiv \lambda x.\lambda y.x < y$ on natural numbers is well-founded. That is, $WF(\mathbf{nat}, <_{nat})$ holds.*

Proof:

By definition of well-foundedness, we need to prove (inductively) that for every $m \in \mathbf{nat}$, all descending sequences from m are finite. We prove this using primitive induction on the natural number m . Primitive induction says that if $T[0]$ holds, and if $T[n]$ holds assuming $T[n - 1]$, then $T[m]$ holds for all natural numbers m .

There is only one descending sequence from 0, and it is finite. Assume that all descending sequences from $n - 1$ are finite. We prove that all descending sequences from n are finite. By WS-introduction, we prove instead that for all $k <_{nat} n$, all descending sequences from k are finite. We do this by cases:

Case $k = (n - 1)$: The result follows trivially from the induction hypothesis.

Case $k < (n - 1)$: By definition of WS, the induction hypothesis is equivalent to the proposition that all descending sequences from j are finite for all $j < (n - 1)$. Since $k < (n - 1)$, the result follows.

A course-of-values induction proof is obtained by specializing the above theorem $WF(\mathbf{nat}, <_{nat})$ at some $n \in \mathbf{nat}$ obtaining $WS(\mathbf{nat}, <_{nat}, n)$, and then using the rule WS-elimination to induct over this structure.

6.3.2 Cartesian Product

Let $(A, <_A)$ and $(B, <_B)$ be well-founded orderings, and let p and q be proofs of the well-foundedness of the respective orderings. Define the lexicographic relation on pairs $\langle a, b \rangle \in A \times B$ as follows: $<_{lex} \equiv \lambda x. \lambda y. (x.1 <_A y.1 \vee (x.1 =_A y.1 \wedge x.2 <_B y.2))$

Theorem 27 *The lexicographic ordering $(A \times B, <_{lex})$ is well-founded:*

$$(\forall \langle A, <_A, p \rangle : WFT) (\forall \langle B, <_B, q \rangle : WFT) WF(A \times B, <_{lex})$$

Proof:

Let $\langle a, b \rangle \in A \times B$. We prove that all descending sequences from $\langle a, b \rangle$ are finite by inducting over the structure of the well-founded ordering $(A, <_A)$ at the element $a \in A$. Assume as inductive hypothesis I that for all $a' <_A a$, all descending sequences from $\langle a', b \rangle$ are finite for all $b \in B$. We now induct over the structure of the well-founded ordering $(B, <_B)$ at $b \in B$. Assume as inductive hypothesis II that for all $b' <_B b$, all descending sequences from $\langle a, b' \rangle$ are finite. We need to prove that all descending sequences from $\langle a, b \rangle$ are finite. By WS-introduction, we prove instead that for all $\langle a', b' \rangle <_{lex} \langle a, b \rangle$, all descending sequences from $\langle a', b' \rangle$ are finite. We do this by cases.

Case $a' <_A a$: The result follows from specializing induction hypothesis I at $b' \in B$.

Case $a' =_A a \wedge b' <_B b$: The result follows from induction hypothesis II.

Corollary 28 *The first and second projection orderings $<_{pr1} \equiv \lambda a. \lambda b. (a.1 <_A b.1)$ and $<_{pr2} \equiv \lambda a. \lambda b. (a.2 <_B b.2)$ are well-founded.*

$$(\forall \langle A, <_A, p \rangle : WFT) (\forall \langle B, <_B, q \rangle : WFT) WF(A \times B, <_{pr_i}) \text{ for } i = 1, 2$$

6.3.3 Example: Greatest Common Divisor

We present a simple example of how well-founded orderings may be used to develop recursive algorithms. If we prove a problem specification by induction on the structure of well-founded orderings, we obtain a program with well-founded recursive structure. The computational structure of the algorithm corresponds to the structure of the ordering used.

The example problem is to find the greatest common divisor n of any two natural numbers i and j . Let $n \mid i$ read as “ n divides i ” and $GCD(n, \langle i, j \rangle)$ read as “ n is the greatest common divisor of i and j ”. That is,

$$n \mid i \equiv ((i \bmod n) = 0)$$

$$\text{and } GCD(n, \langle i, j \rangle) \equiv n \mid i \wedge n \mid j \wedge (\forall k: \mathbf{nat}) (k \mid i \wedge k \mid j \Rightarrow k \leq n)$$

We can then specify the problem as follows:

$$(\forall \langle i, j \rangle : \mathbf{nat} \times \mathbf{nat}) (\exists n: \mathbf{nat}) GCD(n, \langle i, j \rangle)$$

A primitive recursive solution of this problem would require iterating down from the larger of i, j until a divisor is found. By using a simple well-founded ordering, we obtain a general recursive solution for this problem.

Let $(\mathbf{nat} \times \mathbf{nat}, <_{pr2})$ be the second-projection combination of the well-founded ordering $(\mathbf{nat}, <_{nat})$ with itself. We induct over the structure of this ordering by specializing it at $\langle i, j \rangle$, and then applying rule WS-elimination. Assume as inductive hypothesis:

$$(\mathbf{all} \langle i', j' \rangle : \mathbf{nat} \times \mathbf{nat} \mid \langle i', j' \rangle <_{pr2} \langle i, j \rangle) (\exists n: \mathbf{nat}) GCD(n, \langle i', j' \rangle)$$

If $j = 0$, we take i as the solution to the problem. $GCD(i, \langle i, 0 \rangle)$ is easily proved true. Otherwise, by arithmetic reasoning, we know that $\langle j, i \bmod j \rangle <_{pr2} \langle i, j \rangle$. Thus by specializing the inductive hypothesis at this smaller pair, we obtain a natural number m such that $GCD(m, \langle j, i \bmod j \rangle)$. By arithmetic reasoning again, we can show that

$$GCD(m, \langle j, i \bmod j \rangle) =_{nat} GCD(m, \langle i, j \rangle)$$

Thus m is the desired solution to the problem. The program constructed by this proof is:

$$\lambda\langle i, j \rangle. \text{letrec gcd} = \lambda\langle i, j \rangle. \text{if } j = 0 \text{ then } i \text{ else gcd}(\langle j, i \bmod j \rangle) \\ \text{in gcd}(\langle i, j \rangle)$$

6.3.4 Example: Ackermann's Function

Ackermann's function is defined as $\lambda x. \text{ack}(x, x)$ where

$$\text{ack}(m, n) \equiv \text{if } m = 0 \text{ then } n + 1 \text{ else if } n = 0 \text{ then ack}(m - 1, 1) \\ \text{else ack}(m - 1, \text{ack}(m, n - 1))$$

As we discussed in the introduction to this chapter, this function is not primitive recursive over first-order functions. However we can easily derive the general recursive solution by inducting over the lexicographic ordering. The proof involves the same principles used in the gcd proof, and is easily reconstructed.

6.3.5 Disjoint Union

Let $(A, <_A)$ and $(B, <_B)$ be well-founded orderings, and let p and q be proofs of the well-foundedness of the respective orderings.

Let $<_{\text{union}} \equiv \lambda u. \lambda v. \text{case } \langle u, v \rangle \text{ of}$

$$\langle \text{inl}(x), \text{inl}(y) \rangle \Rightarrow (x <_A y)$$

$$\langle \text{inl}(x), \text{inr}(y) \rangle \Rightarrow \text{false}$$

$$\langle \text{inr}(x), \text{inl}(y) \rangle \Rightarrow \text{false}$$

$$\langle \text{inr}(x), \text{inr}(y) \rangle \Rightarrow (x <_B y)$$

Theorem 29 *If $(A, <_A)$ and $(B, <_B)$ are well-founded orderings, then so is their disjoint union:*

$$(\forall \langle A, <_A, p \rangle : WFT) (\forall \langle B, <_B, q \rangle : WFT) WF(A \mid B, <_{\text{union}})$$

Proof: The proof of this is straightforward.

Note that the above ordering can be strengthened by making all elements of A less than all elements of B or viceversa.

6.3.6 Subset

Theorem 30 *If $(A, <)$ is a well-founded ordering, then $<$ is well-founded on any subset of A .*

$$(\forall \langle A, <_A, p \rangle : WFT) (\forall P : A \rightarrow \text{Prop}) WF(\{a : A \mid P(a)\}, <_A)$$

Proof: The proof of this is straightforward.

6.3.7 Mapping

Theorem 31 *If A is a type, $(B, <_B)$ is a well-founded ordering, and f is a function from A to B , then $(A, \lambda x. \lambda y. f(x) <_B f(y))$ is a well-founded ordering.*

Proof: The proof of this is straightforward.

This is also known as the *inverse image construction*.

6.3.8 Example: Quicksort

Quicksort is an efficient algorithm for sorting lists. It first selects a pivot element from list l . It then partitions l into two lists, one of which is smaller than the pivot and the other larger. The original list is sorted by recursively sorting the two lists and then joining them together. Quicksort cannot be defined cleanly by primitive recursion; it requires a general recursive solution. We can easily obtain such a solution by inducting over an inverse image ordering. We use an ordering that compares two lists by comparing their lengths.

To simplify the presentation, we shall work with lists of natural numbers. The proof is easily generalized to obtain a polymorphic quicksort. We shall also assume that the lists are to be sorted in ascending order. This can be easily generalized by parameterizing the comparison operator $<$. We begin by defining several predicates and auxiliary functions. These are all primitive recursive on lists, and can be easily constructed from their specifications.

- $\text{lesslist}(a, l)$ is a predicate that is true if a is less than every element in list l .
- $\text{geqlist}(a, l)$ is a predicate that is true if a is greater than or equal to every element in l .

- $\text{filter}(<, l, a)$ is a function that returns the list of all elements of list l that bear the relation “ $<$ ” with a . Thus $\text{filter}(\leq, l, a)$ returns the list of elements of l less than or equal to a , while $\text{filter}(>, l, a)$ returns the list of elements of l greater than a .
- $(l - a)$ is a function that returns the list l after removing exactly one occurrence of a from it; if a does not occur in l , then l is returned unchanged.
- $\text{member}(a, l)$ is a predicate that is true if a occurs in list l .
- $\text{sorted}(l)$ is a predicate that is true if list l is sorted. This is defined inductively by saying that a list l is sorted if its tail is sorted, and if l 's head is less than all elements in l 's tail.
- $\text{permut}(l, l')$ is a predicate that is true if lists l and l' are permutations of each other.

These auxiliary functions and predicates are defined as follows:

$$\begin{aligned} \text{lesslist}(a, l) &= \text{case } l \text{ of Nil} \Rightarrow \text{true} \\ &\quad | \quad x :: y \Rightarrow (a <_{\text{nat}} x) \wedge \text{lesslist}(a, y) \\ \text{geqlist}(a, l) &= \text{case } l \text{ of Nil} \Rightarrow \text{true} \\ &\quad | \quad x :: y \Rightarrow (x \leq_{\text{nat}} a) \wedge \text{geqlist}(a, y) \\ \text{filter}(<, l, a) &= \text{case } l \text{ of Nil} \Rightarrow \text{Nil} \\ &\quad | \quad x :: y \Rightarrow \text{if } x < a \text{ then } x :: \text{filter}(<, y, a) \text{ else } \text{filter}(<, y, a) \\ l - a &= \text{case } l \text{ of Nil} \Rightarrow \text{Nil} \\ &\quad | \quad x :: y \Rightarrow \text{if } (x = a) \text{ then } y \text{ else } x :: (y - a) \\ \text{member}(a, l) &= \text{case } l \text{ of Nil} \Rightarrow \text{false} \\ &\quad | \quad x :: y \Rightarrow \text{if } (x = a) \text{ then true else } \text{member}(a, y) \\ \text{sorted}(l) &= \text{case } l \text{ of Nil} \Rightarrow \text{true} \\ &\quad | \quad x :: y \Rightarrow \text{lesslist}(x, y) \wedge \text{sorted}(y) \\ \text{permut}(l, l') &= \text{case } l \text{ of Nil} \Rightarrow (l' = \text{Nil}) \\ &\quad | \quad x :: y \Rightarrow \text{member}(x, l') \wedge \text{permut}(y, l' - x) \end{aligned}$$

We can now specify the sorting problem as follows:

$$\text{Sort} = (\forall l : \text{List nat})(\exists l' : \text{List nat}) \text{sorted}(l') \wedge \text{permut}(l, l')$$

We prove this by induction on the length of the list l . We first define a function to compute the length of a list, and a comparison operator that compares two lists by comparing their lengths.

$$\begin{aligned} \text{length} &: \text{List nat} \rightarrow \text{nat} \\ \text{length}(l) &= \text{case } l \text{ of Nil} \Rightarrow 0 \\ &\quad | \quad x :: y \Rightarrow (1 + \text{length}(y)) \\ <_{len} &: \text{List nat} \rightarrow \text{List nat} \rightarrow \text{Prop} \\ <_{len} &= \lambda l. \lambda l'. \text{length}(l) <_{nat} \text{length}(l') \end{aligned}$$

$<_{nat}$ is a well-founded ordering by Theorem 26. Thus, by Theorem 31, $(\text{List nat}, <_{len})$ is a well-founded ordering.

Let l be a list of type List nat . We show, by induction on the length of list l , that there exists a list l' that is a sorted permutation of l . Assume as inductive hypothesis that $(\forall l' <_{len} l)(\exists m : \text{List nat}) \text{sorted}(m) \wedge \text{permut}(m, l')$.

Case $l = \text{Nil}$: $l' = \text{Nil}$ is a sorted permutation of list l .

Case $l = x :: y$: We can show that $\text{filter}(\leq, y, x) \leq_{len} y$ and $\text{filter}(>, y, x) \leq_{len} y$ (these are easily proved by induction). Then, since $y <_{len} l$, we can use the inductive hypothesis to obtain lists p and q such that $\text{sorted}(p) \wedge \text{permut}(p, \text{filter}(\leq, y, x))$ and $\text{sorted}(q) \wedge \text{permut}(q, \text{filter}(>, y, x))$.

Now, we can prove by induction that $\text{geqlist}(x, \text{filter}(\leq, y, x))$ and $\text{lesslist}(x, \text{filter}(>, y, x))$. However, if lists l_1 and l_2 are permutations, then any property which holds for all elements of l_1 also holds for all elements of l_2 (this is again provable by induction). We thus have that $\text{geqlist}(x, p)$ and $\text{lesslist}(x, q)$ hold.

Combining this with the earlier result that p and q are sorted, we get that $p @ x :: q$ is sorted. We thus have

$$\text{sorted}(p @ x :: q) \wedge \text{permut}(p @ x :: q, \text{filter}(\leq, y, x) @ x :: \text{filter}(>, y, x))$$

which simplifies to

$$\text{sorted}(p @ x :: q) \wedge \text{permut}(p @ x :: q, l)$$

Thus, $p @ x :: q$ is the desired witness.

The program constructed from the proof is:

$$\begin{aligned} \text{quicksort} &= \lambda l. \text{case } l \text{ of Nil} \Rightarrow \text{Nil} \\ &| \quad x :: y \Rightarrow \text{quicksort}(\text{filter}(\leq, y, x)) @ x :: \text{quicksort}(\text{filter}(>, y, x)) \end{aligned}$$

6.3.9 Lists

Let $(A, <_A)$ be a well-founded ordering, and let p be a proof of its well-foundedness. Let $\text{SortedList } A = \{l : \text{List } A \mid \text{sorted}(l)\}$ be the type of lists l of type A where the lists are sorted in strict descending order. The predicate `sorted` is defined as follows:

$$\begin{aligned} \text{gtlist}(a, l) &= \text{case } l \text{ of Nil} \Rightarrow \text{true} \\ &| \quad x :: y \Rightarrow (x <_A a) \wedge \text{gtlist}(a, y) \\ \text{sorted}(l) &= \text{case } l \text{ of Nil} \Rightarrow \text{true} \\ &| \quad x :: y \Rightarrow \text{gtlist}(x, y) \wedge \text{sorted}(y) \end{aligned}$$

Define the dictionary relation on sorted lists $l \in \text{SortedList } A$ as follows:

$$\begin{aligned} <_{\text{dict}} &\equiv \lambda l_1. \lambda l_2. \text{case } l_2 \text{ of Nil} \Rightarrow \text{false} \\ &| \quad x_2 :: y_2 \Rightarrow \text{case } l_1 \text{ of Nil} \Rightarrow \text{true} \\ &| \quad x_1 :: y_1 \Rightarrow ((x_1 <_A x_2) \vee \\ &| \quad \quad \quad ((x_1 =_A x_2) \wedge (y_1 <_{\text{dict}} y_2))) \end{aligned}$$

Theorem 32 *The dictionary ordering $(\text{SortedList } A, <_{\text{dict}})$ is well-founded:*

$$(\text{all } \langle A, <_A, p \rangle : \text{WFT}) \text{WF}(\text{SortedList } A, <_{\text{dict}})$$

Proof:

Let $l \in \text{SortedList } A$. Thus l is a list of elements of type A that are sorted in strict descending order. We prove that all descending sequences from l are finite. If l is the empty list Nil , then this holds trivially. Otherwise, let l be $a :: p$. We induct over the structure of the well-founded ordering $(A, <_A)$ at the element $a \in A$. Assume as inductive hypothesis I that for all $a' <_A a$, all descending sequences from $a' :: p'$ are finite for all p' such that $a' :: p' \in \text{SortedList } A$. We prove that all descending sequences from $a :: p'$ are finite for all p' such that $a :: p' \in \text{SortedList } A$. Then, by specializing this at $p \in \text{SortedList } A$, we will obtain the desired result.

Let $p' \in \text{SortedList } A$ such that $a :: p' \in \text{SortedList } A$. By definition, p' is a list of elements strictly smaller than a . Thus all descending sequences from p' are finite (this is trivial if p is Nil ; otherwise, it follows from inductive hypothesis I applied to the first element of list p). We can thus induct over this structure. Assume as inductive hypothesis II that for all $q' <_{\text{dict}} q$, all descending sequences from $a :: q'$ are finite. We prove that all descending sequences from $a :: q$ are finite.

By WS-introduction, we prove instead that for all $l' <_{\text{dict}} a :: q$, all descending sequences from l' are finite. If l' is the empty list Nil , this is trivial. Otherwise let $l' = (s :: t)$. We proceed by cases.

Case $s < a$: The result follows from specializing induction hypothesis I at $t \in \text{SortedList } A$.

Case $s =_A a$ and $t <_{\text{dict}} q$: The result follows from induction hypothesis II.

Corollary 33 *The ordering $\lambda l.\lambda l'.\text{sort}(l) <_{\text{dict}} \text{sort}(l')$ on arbitrary lists (without duplicate elements) is well-founded.*

This ordering sorts the argument lists in strict descending order and then compares them using the dictionary ordering. The ordering is well-founded since it is obtained by applying the inverse image construction to the dictionary ordering.

6.4 Comments

We have presented techniques for developing practical recursive programs in a Martin-Lof-style type theory. This work clearly shows the power of inductive types and their role in supporting general recursion. The notions of well-foundedness and the various classes of well-founded orderings are well-known. We have formulated these notions in a constructive framework, and have presented constructive proofs of well-foundedness of orderings. These proofs make clear the computational structure of the orderings. They also permit this potentially complex structure to be used to easily derive total recursive programs with a similar computational structure.

Chapter 7

Conclusion

We have addressed the problem of adding assignments and dynamic data to functional languages without violating their semantic properties. We have presented both a programming language (ILC) and a programming logic (OTT). In this chapter, we review the results of this thesis and discuss research directions.

7.1 Results

We have presented a formal basis for adding mutable references and assignments to applicative languages without violating their semantic properties. This is achieved through a rich type system that distinguishes between state-dependent and state-independent expressions and sequentializes modifications to the state. The language, called ILC, possesses the desired properties of applicative languages such as strong normalization and confluence. At the same time, it allows the efficient encoding of state-oriented algorithms and linked data structures.

We have also presented a programming logic for ILC called *observation type theory* (OTT). OTT is a strict extension of a Martin-Löf-style constructive type theory. It embodies the fundamental logical principles that underlie our formulation of references and assignments; these are remarkably similar to the principles that underlie variables and functions. OTT's language of constructions is ILC. As mentioned above, 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 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 might be inevitable. We continue to investigate further refinement of the present theory.

Finally, we have presented techniques for developing practical recursive programs in a Martin-Lof-style type theory. These techniques can also be used to develop recursive programs in OTT.

7.2 Future Directions

We hope this work forms the beginning of a systematic and disciplined integration of functional and imperative programming paradigms. Their differing strengths are orthogonal, but not conflicting. Much further work remains to be done regarding the approach presented here. The issues of polymorphism over mutable and observer types must be investigated. A complete equational calculus must be found for supporting formal reasoning. This, in turn, requires a formalization of the models of ILC and the development of proof methods like logical relations. The incorporation of an effect system and use of the monad comprehension notation would make the language more flexible and convenient to use. Finally, the issues of implementation (including language syntax, type reconstruction, and program optimization) need to be addressed.

Much work remains to be done regarding the practical application of OTT for carrying out proofs and program derivations. Some examples were presented in this thesis, but much experience needs to be obtained; 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 to the nature of references, observer methods, and assignment as a form of method application. It would be worthwhile to study programming paradigms weaker than references to obtain a better understanding of their nature, *e.g.*, “logic variables” of logic programming [41, 88] and the negative types of linear logic [1, 25].

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.

Appendix A

Rules for Constructive Type Theory

We present a collection of rules that describe the type theory used in the examples of this thesis. Our description of what the rules mean is informal. A formal account of the judgements and rules may be found in writings of Martin-Lof [46] and the Nuprl group [6]. Our judgements should be interpreted as Nuprl-style judgements, even though we adopt Martin-Lof's notation for this presentation.

A type assignment Γ is a *sequential* assignment of types to variables. Γ has the form $\Gamma = x_1 : T_1, \dots, x_k : T_k$, with no x_i occurring twice. The units of assertion of the theory are called *judgements*. For example, the judgement $\Gamma \vdash e : T$ means (intuitively) that if variables x_1, \dots, x_k have types $T_1, T_2[x_1] \dots, T_k[x_1 \dots x_{(k-1)}]$ respectively, then e is a well-formed term of type T . We read it as saying “in context Γ , the term e has type T .” A typing rule has the form

$$\frac{J_1, \dots, J_k}{J}$$

It says that if each judgement J_i is valid, then so is judgement J .

The theory is described as follows:

Syntax: We present the syntax of the language (this was presented earlier in Section 5.3).

Terms are either *canonical* or *non-canonical*. Canonical terms are the normal-forms of the language. Non-canonical terms can be reduced by the computation rules of the system.

Free variables: We inductively define a function $V(e)$ that determines the set of free variables of term e .

Computation rules: We present a collection of rewrite rules that determine the operational semantics of the language. A term is reduced by applying these rules in an outermost, leftmost evaluation order. This is called *head-reduction*. If a term is non-canonical and none of the rules apply, reduction terminates with an error. Reduction of well-typed terms is normalizing – it reduces every well-typed term to head normal form.

Formation rules: We present rules that determine when a type is well-formed. An informal reading of the judgement $\Gamma \vdash T : U_i$ is that the term T is a well-formed type under the type assignment Γ .

Congruence rules for types Congruence rules are used to specify when two types are equal. The judgement $\Gamma \vdash S = T$ may be read as asserting that terms S and T are equal well-formed types under the type assignment Γ . This relation is taken to be intensional equality. That is, two types are equal if their component types are equal. Consequently, we do not present congruence rules for types.

Introduction and elimination rules: Introduction rules specify the well-formed members of a type, while elimination rules indicate how to compute with these members. Both classes of rules use the judgement $\Gamma \vdash t : T$ to assert that term t has type T under the type assignment Γ .

Congruence rules: These rules specify when two terms are equal, well-formed members of a type. The judgement $\Gamma \vdash s = t \text{ in } T$ asserts that s and t are equal members of type T . Note that $S = T \text{ in } U_i$ is the same as the type congruence $(S = T)$; as mentioned above, this is taken to be intensional equality, and such rules are not presented.

Canonical	
x	variable
n	natural number (constant)
CT	canonical type
$\text{inl}(e)$	inject left
$\text{inr}(e)$	inject right
$\langle e_1, e_2 \rangle$	pair
$\lambda x.e$	abstraction
fact	unit
Noncanonical	
abort (e)	error/abort
$e_1 \text{ op } e_2$	$+$, $-$, $*$, $/$ or mod
ind ($e_1; e_2; x, y.e_3$)	primitive induction
case e_1 of $\text{inl}(x) \Rightarrow e_2 \mid \text{inr}(x) \Rightarrow e_3$ end	union tag discrimination
let $\langle x, y \rangle = e_1$ in e_2	pair component selection
$e_1(e_2)$	function application
letrec $f = \lambda x.e_1$ in $f(e_2)$	recursion
if $e_1 = e_2$ then e_3 else e_4	equality conditional
if $e_1 < e_2$ then e_3 else e_4	less-than conditional

Figure A.1: Abstract syntax of terms.

void	empty
nat	natural numbers
U_n	universe
$S \mid T$	disjoint union
$S \times T$	cartesian product
$S \rightarrow T$	function space
$(\Sigma x: S)T_x$	dependent product
$(\Pi x: S)T_x$	dependent function
$\{x: S \mid T_x\}$	subset
$(\mu x: U_n)T_x$	inductive type
$(e_1 = e_2 \text{ in } T)$	equality
$(e_1 < e_2)$	less-than

Figure A.2: Abstract syntax of canonical types.

$V(x)$	$= x$
$V(n)$	$= \phi$
$V(\mathbf{inl}(e))$	$= V(e)$
$V(\mathbf{inr}(e))$	$= V(e)$
$V((e_1, e_2))$	$= V(e_1) \cup V(e_2)$
$V(\lambda x.e)$	$= V(e) - \{x\}$
$V(\mathbf{fact})$	$= \phi$
$V(\mathbf{abort}(e))$	$= V(e)$
$V(e_1 \text{ op } e_2)$	$= V(e_1) \cup V(e_2)$
$V(\mathbf{ind}(e_1; e_2; r, y.e_3))$	$= V(e_1) \cup V(e_2) \cup (V(e_3) - \{r, y\})$
$V(\mathbf{case } e_1 \text{ of } \mathbf{inl}(x) \Rightarrow e_2 \mid \mathbf{inr}(y) \Rightarrow e_3 \text{ end})$	$= V(e_1) \cup (V(e_2) - \{x\}) \cup (V(e_3) - \{y\})$
$V(\mathbf{let } \langle x, y \rangle = e_1 \text{ in } e_2)$	$= V(e_1) \cup (V(e_2) - \{x, y\})$
$V(e_1(e_2))$	$= V(e_1) \cup V(e_2)$
$V(\mathbf{letrec } f = \lambda x.e_1 \text{ in } f(e_2))$	$= (V(e_1) - \{x, f\}) \cup (V(e_2) - \{f\})$
$V(\mathbf{if } e_1 = e_2 \text{ then } e_3 \text{ else } e_4)$	$= V(e_1) \cup V(e_2) \cup V(e_3) \cup V(e_4)$
$V(\mathbf{if } e_1 < e_2 \text{ then } e_3 \text{ else } e_4)$	$= V(e_1) \cup V(e_2) \cup V(e_3) \cup V(e_4)$
$V(\mathbf{void})$	$= \phi$
$V(\mathbf{nat})$	$= \phi$
$V(U_i)$	$= \phi$
$V(S \mid T)$	$= V(S) \cup V(T)$
$V(S \times T)$	$= V(S) \cup V(T)$
$V(S \rightarrow T)$	$= V(S) \cup V(T)$
$V((\Sigma x: S)T_x)$	$= V(S) \cup (V(T) - \{x\})$
$V((\Pi x: S)T_x)$	$= V(S) \cup (V(T) - \{x\})$
$V(\{x: S \mid T_x\})$	$= V(S) \cup (V(T) - \{x\})$
$V((\mu x: U_i)T_x)$	$= V(T) - \{x\}$
$V((s = t \text{ in } T))$	$= V(s) \cup V(t) \cup V(T)$
$V((s < t))$	$= V(s) \cup V(t)$

Figure A.3: Free variables of terms.

<p><i>void-formation</i></p> $\Gamma \vdash \mathbf{void} : U_i$	<p><i>nat-formation</i></p> $\Gamma \vdash \mathbf{nat} : U_i$
<p><i>×-formation</i></p> $\frac{\Gamma \vdash S : U_i \quad \Gamma \vdash T : U_i}{\Gamma \vdash (S \times T) : U_i}$	<p><i>→-formation</i></p> $\frac{\Gamma \vdash S : U_i \quad \Gamma \vdash T : U_i}{\Gamma \vdash (S \rightarrow T) : U_i}$
<p><i>Σ-formation</i></p> $\frac{\Gamma \vdash S : U_i \quad \Gamma, x : S \vdash T : U_i}{\Gamma \vdash (\Sigma x : S) T : U_i}$	<p><i>Π-formation</i></p> $\frac{\Gamma \vdash S : U_i \quad \Gamma, x : S \vdash T : U_i}{\Gamma \vdash (\Pi x : S) T : U_i}$
<p><i> -formation</i></p> $\frac{\Gamma \vdash S : U_i \quad \Gamma \vdash T : U_i}{\Gamma \vdash (S T) : U_i}$	<p><i>subset-formation</i></p> $\frac{\Gamma \vdash S : U_i \quad \Gamma, x : S \vdash T : U_i}{\Gamma \vdash \{x : S T\} : U_i}$
<p><i>μ-formation</i></p> $\frac{\Gamma, x : U_i \vdash T : U_i \quad \Gamma, x : U_i, y : U_i, x \subseteq y \vdash T \subseteq T[y/x]}{\Gamma \vdash (\mu x : U_i) T : U_i}$	<p><i><-formation</i></p> $\frac{\Gamma \vdash e_1 : \mathbf{nat} \quad \Gamma \vdash e_2 : \mathbf{nat}}{\Gamma \vdash (e_1 < e_2) : U_i}$
<p><i>=-formation</i></p> $\frac{\Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T \quad \Gamma \vdash T : U_i}{\Gamma \vdash (e_1 = e_2 \mathbf{in} T) : U_i}$	

Figure A.4: Type formation rules.

void-introduction

void-elimination

$$\frac{\Gamma \vdash e : \mathbf{void}}{\Gamma \vdash \mathbf{abort}(e) : T}$$

nat-introduction

$$\Gamma \vdash i : \mathbf{nat} \quad (\text{for } i = 0, 1, 2, \dots)$$

nat-elimination-1

$$\frac{\Gamma \vdash e_1 : \mathbf{nat} \quad \Gamma \vdash e_2 : \mathbf{nat}}{\Gamma \vdash e_1 \text{ op } e_2 : \mathbf{nat}}$$

nat-elimination-2

$$\frac{\Gamma \vdash e_1 : \mathbf{nat} \quad \Gamma \vdash s : T[0/z] \quad \Gamma, x : \mathbf{nat}, x > 0, y : T[x-1/z] \vdash e_2 : T[x/z]}{\Gamma \vdash \mathbf{ind}(e_1; s; x, y.e_2) : T[e_1/z]}$$

\times -introduction

$$\frac{\Gamma \vdash s : S \quad \Gamma \vdash t : T}{\Gamma \vdash \langle s, t \rangle : S \times T}$$

\times -elimination

$$\frac{\Gamma \vdash e_1 : S \times T \quad \Gamma, x : S, y : T \vdash e_2 : A[\langle x, y \rangle/z]}{\Gamma \vdash \mathbf{let} \langle x, y \rangle = e_1 \text{ in } e_2 : A[e_1/z]}$$

\rightarrow -introduction

$$\frac{\Gamma, x : S \vdash t : T}{\Gamma \vdash \lambda x. t : S \rightarrow T}$$

\rightarrow -elimination

$$\frac{\Gamma \vdash e : S \quad \Gamma \vdash f : S \rightarrow T}{\Gamma \vdash f(e) : T}$$

Σ -introduction

$$\frac{\Gamma \vdash s : S \quad \Gamma \vdash t : T[s/x]}{\Gamma \vdash \langle s, t \rangle : (\Sigma x : S)T}$$

Σ -elimination

$$\frac{\Gamma \vdash e_1 : (\Sigma x : S)T \quad \Gamma, x : S, y : T \vdash e_2 : A[\langle x, y \rangle/z]}{\Gamma \vdash \mathbf{let} \langle x, y \rangle = e_1 \text{ in } e_2 : A[e_1/z]}$$

Π -introduction

$$\frac{\Gamma, x : S \vdash t : T}{\Gamma \vdash \lambda x. t : (\Pi x : S)T}$$

Π -elimination

$$\frac{\Gamma \vdash e : S \quad \Gamma \vdash f : (\Pi x : S)T}{\Gamma \vdash f(e) : T[e/x]}$$

Figure A.5: Introduction and elimination rules.

<p> <i>-introduction-1</i></p> $\frac{\Gamma \vdash s : S}{\Gamma \vdash \mathbf{inl}(s) : S \mid T}$	<p> <i>-elimination</i></p> $\frac{\Gamma, x : S \vdash e_2 : A[\mathbf{inl}(x)/z] \quad \Gamma \vdash e_1 : S \mid T \quad \Gamma, y : T \vdash e_3 : A[\mathbf{inr}(y)/z]}{\Gamma \vdash \mathbf{case } e_1 \mathbf{ of } \mathbf{inl}(x) \Rightarrow e_2 \mid \mathbf{inr}(y) \Rightarrow e_3 \mathbf{ end} : A[e_1/z]}$
<p> <i>-introduction-2</i></p> $\frac{\Gamma \vdash t : T}{\Gamma \vdash \mathbf{inr}(t) : S \mid T}$	
<p>subset-<i>introduction</i></p> $\frac{\Gamma \vdash s : S \quad \Gamma \vdash t : T[s/x]}{\Gamma \vdash s : \{x : S \mid T\}}$	<p>subset-<i>elimination</i> if $y \notin V(e_2)$</p> $\frac{\Gamma \vdash e_1 : \{x : S \mid T\} \quad \Gamma, x : S, [y : T] \vdash e_2 : A[x/z]}{\Gamma \vdash e_2[e_1/x] : A[e_1/z]}$
<p>μ-<i>introduction</i></p> $\frac{\Gamma \vdash t : T[(\mu x : U_i)T/x]}{\Gamma \vdash t : (\mu x : U_i)T}$	<p>μ-<i>elimination</i></p> $\frac{\Gamma \vdash e_2 : (\mu x : U_i)T \quad \Gamma, x : U_i, x \subseteq (\mu x : U_i)T, f : (\Pi z : x)A, z : T \vdash e_1 : A}{\Gamma \vdash (\mathbf{letrec } f = \lambda z. e_1 \mathbf{ in } f(e_2)) : A[e_2/z]}$
<p>= <i>-conditional</i></p> $\frac{\Gamma \vdash e_1 : \mathbf{nat} \quad \Gamma, e_1 = e_2 \mathbf{ in } \mathbf{nat} \vdash e_3 : T \quad \Gamma \vdash e_2 : \mathbf{nat} \quad \Gamma, \neg(e_1 = e_2 \mathbf{ in } \mathbf{nat}) \vdash e_4 : T}{\Gamma \vdash (\mathbf{if } e_1 = e_2 \mathbf{ then } e_3 \mathbf{ else } e_4) : T}$	<p>< <i>-conditional</i></p> $\frac{\Gamma \vdash e_1 : \mathbf{nat} \quad \Gamma, e_1 < e_2 \vdash e_3 : T \quad \Gamma \vdash e_2 : \mathbf{nat} \quad \Gamma, \neg(e_1 < e_2) \vdash e_4 : T}{\Gamma \vdash (\mathbf{if } e_1 < e_2 \mathbf{ then } e_3 \mathbf{ else } e_4) : T}$

Figure A.6: Introduction and elimination rules (continued).

equality

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash \mathbf{fact} : (t = t \text{ in } T)}$$

×-congruence

$$\frac{\Gamma \vdash \mathbf{fact} : (s = s' \text{ in } S) \quad \Gamma \vdash \mathbf{fact} : (t = t' \text{ in } T)}{\Gamma \vdash \mathbf{fact} : (\langle s, t \rangle = \langle s', t' \rangle \text{ in } S \times T)}$$

→-congruence

$$\frac{\Gamma \vdash f : S \rightarrow T \quad \Gamma \vdash g : S \rightarrow T \quad \Gamma, x : S \vdash \mathbf{fact} : (f(x) = g(x) \text{ in } T)}{\Gamma \vdash \mathbf{fact} : (f = g \text{ in } S \rightarrow T)}$$

Σ-congruence

$$\frac{\Gamma \vdash \mathbf{fact} : (s = s' \text{ in } S) \quad \Gamma \vdash \mathbf{fact} : (t = t' \text{ in } T[s/x])}{\Gamma \vdash \mathbf{fact} : (\langle s, t \rangle = \langle s', t' \rangle \text{ in } (\Sigma x : S)T)}$$

Π-congruence

$$\frac{\Gamma \vdash f : (\Pi x : S)T \quad \Gamma \vdash g : (\Pi x : S)T \quad \Gamma, x : S \vdash \mathbf{fact} : (f(x) = g(x) \text{ in } T)}{\Gamma \vdash \mathbf{fact} : (f = g \text{ in } (\Pi x : S)T)}$$

| -congruence-1

$$\frac{\Gamma \vdash \mathbf{fact} : (s = s' \text{ in } S)}{\Gamma \vdash \mathbf{fact} : (\text{inl}(s) = \text{inl}(s') \text{ in } S | T)}$$

| -congruence-2

$$\frac{\Gamma \vdash \mathbf{fact} : (t = t' \text{ in } T)}{\Gamma \vdash \mathbf{fact} : (\text{inr}(t) = \text{inr}(t') \text{ in } S | T)}$$

subset-congruence

$$\frac{\Gamma \vdash \mathbf{fact} : (s = s' \text{ in } S)}{\Gamma \vdash \mathbf{fact} : (s = s' \text{ in } \{x : S | T\})}$$

μ-congruence

$$\frac{\Gamma \vdash \mathbf{fact} : (t = t' \text{ in } T[(\mu x : U_i)T/x])}{\Gamma \vdash \mathbf{fact} : (t = t' \text{ in } (\mu x : U_i)T)}$$

Figure A.7: Congruence rules.

$-n$	\longrightarrow	the negation of n
$m \text{ op } n$	\longrightarrow	the result of the operation “op” on m and n
$\text{ind}(0; e_1; x, y.e_2)$	\longrightarrow	e_1
$\text{ind}(m; e_1; x, y.e_2)$	\longrightarrow	$e_2[m, \text{ind}(m - 1; e_1; x, y.e_2)/x, y]$ if $m > 0$
$\text{if } m = n \text{ then } e_1 \text{ else } e_2$	\longrightarrow	e_1 if $m = n$
$\text{if } m = n \text{ then } e_1 \text{ else } e_2$	\longrightarrow	e_2 if $m \neq n$
$\text{if } m < n \text{ then } e_1 \text{ else } e_2$	\longrightarrow	e_1 if $m < n$
$\text{if } m < n \text{ then } e_1 \text{ else } e_2$	\longrightarrow	e_2 if $m \not< n$
$\text{let } \langle x, y \rangle = \langle e_1, e_2 \rangle \text{ in } e_3$	\longrightarrow	$e_3[e_1, e_2/x, y]$
$(\lambda x.e_1)(e_2)$	\longrightarrow	$e_1[e_2/x]$
$\text{case inl}(e_1) \text{ of } \text{inl}(x) \Rightarrow e_2$	\longrightarrow	$e_2[e_1/x]$
$\text{inr}(y) \Rightarrow e_3$		
$\text{case inr}(e_1) \text{ of } \text{inl}(x) \Rightarrow e_2$	\longrightarrow	$e_3[e_1/y]$
$\text{inr}(y) \Rightarrow e_3$		
$\text{letrec } f = \lambda y.e_1 \text{ in } f(e_2)$	\longrightarrow	$e_1[e_2, (\lambda x.\text{letrec } f = \lambda y.e_1 \text{ in } f(x))/y, f]$

Figure A.8: Computation rules.

Appendix B

Rules for Observation Type Theory

In this appendix, we summarize the definition of simple observation type theory. We present the syntax, reduction rules and proof rules of the theory.

A type assignment Γ is a *sequential* assignment of types to variables. Γ has the form $\Gamma = x_1 : T_1, \dots, x_k : T_k$, with no x_i occurring twice. The units of assertion of the theory are called *judgements*. For example, the judgement $\Gamma \vdash e : T$ means (intuitively) that if variables x_1, \dots, x_k have types $T_1, T_2[x_1], \dots, T_k[x_1 \dots x_{(k-1)}]$ respectively, then e is a well-formed term of type T . We read it as saying “in context Γ , the term e has type T .” A typing rule has the form

$$\frac{J_1, \dots, J_k}{J}$$

It says that if each judgement J_i is valid, then so is judgement J .

Terms	$e ::= k \mid x \mid v^* \mid \lambda x.e \mid ee \mid \langle e, e \rangle \mid e.1 \mid e.2 \mid \mathbf{abort}(e) \mid \mathbf{fact} \mid$ $\mathbf{letref} \ v^* := e \ \mathbf{in} \ e \mid \mathbf{get} \ x \leftarrow e \ \mathbf{in} \ e \mid e := e ; e$
Applicative types	$\tau ::= \beta \mid \mathbf{void} \mid (\Pi x:\tau)\tau \mid (\Sigma x:\tau)\tau \mid e = e \ \mathbf{in} \ \tau$
Storage types	$\theta ::= \tau \mid \mathbf{Ref} \ \theta \mid (\Pi x:\theta)\theta \mid (\Sigma x:\theta)\theta \mid e = e \ \mathbf{in} \ \theta \mid (e =_{\theta} e)$
Assertion types	$\alpha ::= \tau \mid \mathbf{Obs} \ \alpha \mid \mathbf{Get} \ x:\theta \leftarrow e \ \mathbf{in} \ \alpha \mid (\mathbf{All} \ x:\alpha)\alpha \mid (\mathbf{Some} \ x:\alpha)\alpha \mid$ $e = e \ \mathbf{in} \ \alpha \mid (e =_{\theta} e) \mid (e \sim \alpha)$
Observation types	$\omega ::= \theta \mid \alpha \mid (\Pi x:\omega)\omega \mid (\Sigma x:\omega)\omega \mid e = e \ \mathbf{in} \ \omega$

Figure B.1: Abstract syntax.

(1)	$(\lambda x: \omega. e_1)(e_2)$	\longrightarrow	$e_1[e_2/x]$	
(2)	$\langle e_1, e_2 \rangle. i$	\longrightarrow	e_i	for $i = 1, 2$
(3)	letref $v^*: \text{Ref } \theta := e_1$ in obs (e_2)	\longrightarrow	obs (e_2)	if $v^* \notin V(e_2)$
(4)	letref $v^*: \text{Ref } \theta := e_1$ in get $x: \theta \leftarrow v^*$ in e_2	\longrightarrow	letref $v^*: \text{Ref } \theta := e_1$ in $e_2[e_1/x]$	
(5)	letref $v^*: \text{Ref } \theta_1 := e_1$ in get $x: \theta_2 \leftarrow w^*$ in e_2	\longrightarrow	get $x': \theta_2 \leftarrow w^*$ in letref $v^*: \text{Ref } \theta_1 := e_1$ in $e_2[x'/x]$	if $v^* \neq w^*$ and $x' \notin V(e_1) \cup V(e_2)$
(6)	$v^* := e_1$; obs (e_2)	\longrightarrow	obs (e_2)	
(7)	$v^* := e_1$; get $x: \theta \leftarrow v^*$ in e_2	\longrightarrow	$v^* := e_1$; $e_2[e_1/x]$	
(8)	$v^* := e_1$; get $x: \theta \leftarrow w^*$ in e_2	\longrightarrow	get $x': \theta \leftarrow w^*$ in $v^* := e_1$; $e_2[x'/x]$	if $v^* \neq w^*$ and $x' \notin V(e_1) \cup V(e_2)$
(9)	app (obs (e))	\longrightarrow	e	

Figure B.2: Reduction rules.

<p><i>β-formation</i></p> $\Gamma \vdash \beta \text{ Type}_\tau$	<p><i>void-formation</i></p> $\Gamma \vdash \text{void} \text{ Type}_\tau$
<p><i>Π-formation</i></p> $\frac{\Gamma \vdash \tau_1 \text{ Type}_\tau \quad \Gamma, x: \tau_1 \vdash \tau_2 \text{ Type}_\tau}{\Gamma \vdash (\Pi x: \tau_1) \tau_2 \text{ Type}_\tau}$	<p><i>Σ-formation</i></p> $\frac{\Gamma \vdash \tau_1 \text{ Type}_\tau \quad \Gamma, x: \tau_1 \vdash \tau_2 \text{ Type}_\tau}{\Gamma \vdash (\Sigma x: \tau_1) \tau_2 \text{ Type}_\tau}$
<p><i>equality-formation</i></p> $\frac{\Gamma \vdash e_1: \tau \quad \Gamma \vdash e_2: \tau \quad \Gamma \vdash \tau \text{ Type}_\tau}{\Gamma \vdash (e_1 = e_2 \text{ in } \tau) \text{ Type}_\tau}$	

Figure B.3: Type formation rules for τ types.

<p><i>τ-coercion</i></p> $\frac{\Gamma \vdash \tau \text{ Type}_\tau}{\Gamma \vdash \tau \text{ Type}_\theta}$	<p><i>Ref-formation</i></p> $\frac{\Gamma \vdash \theta \text{ Type}_\theta}{\Gamma \vdash (\text{Ref } \theta) \text{ Type}_\theta}$
<p><i>Π-formation</i></p> $\frac{\Gamma \vdash \theta_1 \text{ Type}_\theta \quad \Gamma, x: \theta_1 \vdash \theta_2 \text{ Type}_\theta}{\Gamma \vdash (\Pi x: \theta_1) \theta_2 \text{ Type}_\theta}$	<p><i>Σ-formation</i></p> $\frac{\Gamma \vdash \theta_1 \text{ Type}_\theta \quad \Gamma, x: \theta_1 \vdash \theta_2 \text{ Type}_\theta}{\Gamma \vdash (\Sigma x: \theta_1) \theta_2 \text{ Type}_\theta}$
<p><i>equality-formation</i></p> $\frac{\Gamma \vdash e_1: \theta \quad \Gamma \vdash e_2: \theta \quad \Gamma \vdash \theta \text{ Type}_\theta}{\Gamma \vdash (e_1 = e_2 \text{ in } \theta) \text{ Type}_\theta}$	<p><i>typeless-equality-formation</i></p> $\frac{\Gamma \vdash e_1: \theta_1 \quad \Gamma \vdash e_2: \theta_2 \quad \Gamma \vdash \theta_1 \text{ Type}_\theta \quad \Gamma \vdash \theta_2 \text{ Type}_\theta}{\Gamma \vdash (e_1 =_\theta e_2) \text{ Type}_\theta}$

Figure B.4: Type formation rules for θ types.

<p><i>Obs-formation</i></p> $\frac{\Gamma \vdash \alpha \text{ Type}_\alpha}{\Gamma \vdash \text{Obs } \alpha \text{ Type}_\alpha}$	<p><i>Get-formation</i></p> $\frac{\Gamma \vdash l : \text{Ref } \theta \quad \Gamma, x: \theta \vdash \alpha \text{ Type}_\alpha}{\Gamma \vdash (\text{Get } x: \theta \leftarrow l \text{ in } \alpha) \text{ Type}_\alpha}$
<p><i>All -formation</i></p> $\frac{\Gamma \vdash \alpha_1 \text{ Type}_\alpha \quad \Gamma, x: \alpha_1 \vdash \alpha_2 \text{ Type}_\alpha}{\Gamma \vdash (\text{All } x: \alpha_1) \alpha_2 \text{ Type}_\alpha}$	<p><i>Some -formation</i></p> $\frac{\Gamma \vdash \alpha_1 \text{ Type}_\alpha \quad \Gamma, x: \alpha_1 \vdash \alpha_2 \text{ Type}_\alpha}{\Gamma \vdash (\text{Some } x: \alpha_1) \alpha_2 \text{ Type}_\alpha}$
<p><i>equality-formation</i></p> $\frac{\Gamma \vdash e_1: \alpha_1 \quad \Gamma \vdash e_2: \alpha_1 \quad \Gamma \vdash \alpha_1 \text{ Type}_\alpha}{\Gamma \vdash (e_1 = e_2 \text{ in } \alpha_1) \text{ Type}_\alpha}$	<p><i>typeless-equality-formation</i></p> $\frac{\Gamma \vdash (e_1 =_\theta e_2) \text{ Type}_\theta}{\Gamma \vdash (e_1 =_\theta e_2) \text{ Type}_\alpha}$
<p><i>noninterference-formation</i></p> $\frac{\Gamma \vdash l : \text{Ref } \theta \quad \Gamma \vdash \alpha \text{ Type}_\alpha}{\Gamma \vdash (l \sim \alpha) \text{ Type}_\alpha}$	<p><i>τ-to-α</i></p> $\frac{\Gamma \vdash \tau \text{ Type}_\tau}{\Gamma \vdash \tau \text{ Type}_\alpha}$

Figure B.5: Type formation rules for α types.

<p><i>θ-coercion</i></p> $\frac{\Gamma \vdash \theta \text{Type}_\theta}{\Gamma \vdash \theta \text{Type}_\omega}$	<p><i>α-coercion</i></p> $\frac{\Gamma \vdash \alpha \text{Type}_\alpha}{\Gamma \vdash \alpha \text{Type}_\omega}$
<p><i>Π-formation</i></p> $\frac{\Gamma \vdash \omega_1 \text{Type}_\omega \quad \Gamma, x: \omega_1 \vdash \omega_2 \text{Type}_\omega}{\Gamma \vdash (\Pi x: \omega_1) \omega_2 \text{Type}_\omega}$	<p><i>Σ-formation</i></p> $\frac{\Gamma \vdash \omega_1 \text{Type}_\omega \quad \Gamma, x: \omega_1 \vdash \omega_2 \text{Type}_\omega}{\Gamma \vdash (\Sigma x: \omega_1) \omega_2 \text{Type}_\omega}$
<p><i>equality-formation</i></p> $\frac{\Gamma \vdash e_1: \omega \quad \Gamma \vdash e_2: \omega \quad \Gamma \vdash \omega \text{Type}_\omega}{\Gamma \vdash (e_1 = e_2 \text{ in } \omega) \text{Type}_\omega}$	

Figure B.6: Type formation rules for ω types.

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

Figure B.7: Equivalence of α type-terms.

hypothesis

$\Gamma, x: S, \Gamma' \vdash x: T$
if S is α -convertible to T

weakening

$\frac{\Gamma \vdash t: T}{\Gamma, x: S \vdash t: T}$

reflexivity

$\frac{\Gamma \vdash t: T}{\Gamma \vdash \mathbf{fact} : (t = t \text{ in } T)}$

computation

$\frac{\Gamma \vdash t: T \quad \Gamma \vdash t': T}{\Gamma \vdash \mathbf{fact} : (t = t' \text{ in } T)}$ if t converts to t'

equality

$\Gamma \vdash \mathbf{fact} : (t = t' \text{ in } T)$
if the equality can be deduced from Γ
using only symmetry, transitivity and
congruence

substitutivity

$\frac{\Gamma \vdash t : T[s/x] \quad \Gamma \vdash \mathbf{fact} : (s = s' \text{ in } S)}{\Gamma \vdash t : T[s'/x]}$

void-elimination

$\frac{\Gamma \vdash t: \mathbf{void}}{\Gamma \vdash \mathbf{abort}(t) : T}$

Figure B.8: Miscellaneous inference rules for all layers (τ , θ , α and ω).

Σ -introduction

$$\frac{\Gamma \vdash s : S \quad \Gamma \vdash t : T[s/x]}{\Gamma \vdash \langle s, t \rangle : (\Sigma x : S)T}$$

Π -introduction

$$\frac{\Gamma, x : S \vdash t : T}{\Gamma \vdash (\lambda x. t) : (\Pi x : S)T}$$

Σ -elimination

$$\frac{\Gamma \vdash e : (\Sigma x : S)T}{\Gamma \vdash e.1 : S} \quad \frac{\Gamma \vdash e : (\Sigma x : S)T}{\Gamma \vdash e.2 : T[e.1/x]}$$

Π -elimination

$$\frac{\Gamma \vdash f : (\Pi x : S)T \quad \Gamma \vdash s : S}{\Gamma \vdash f(s) : T[s/x]}$$

Σ -equality

$$\frac{\Gamma \vdash \mathbf{fact} : (e.1 = e'.1 \text{ in } S) \quad \Gamma, x : S \vdash \mathbf{fact} : (e.2 = e'.2 \text{ in } T)}{\Gamma \vdash \mathbf{fact} : (e = e' \text{ in } (\Sigma x : S)T)}$$

Π -equality

$$\frac{\Gamma, x : S \vdash \mathbf{fact} : (f(x) = g(x) \text{ in } T)}{\Gamma \vdash \mathbf{fact} : (f = g \text{ in } (\Pi x : S)T)}$$

typeless-equality

$$\frac{\Gamma \vdash t : (e_1 = e_2 \text{ in } \theta)}{\Gamma \vdash t : (e_1 =_{\theta} e_2)}$$

Figure B.9: Inference rules for type constructors (in layers τ , θ , and ω).

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

Dereference

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

Assignment

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

Creation

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

Figure B.10: Inference rules for the α layer.

$\sim \text{-axiom-}\tau$ $\frac{\Gamma \vdash l : \text{Ref } \theta \quad \Gamma \vdash \tau_1 \text{ Type}_\tau}{\Gamma \vdash \mathbf{fact} : (l \sim \tau_1)}$	
$\sim \text{-intro-Obs}$ $\frac{\Gamma \vdash \mathbf{fact} : (l \sim \alpha)}{\Gamma \vdash \mathbf{fact} : (l \sim \mathbf{Obs } \alpha)}$	$\sim \text{-intro-Get}$ $\frac{\Gamma \vdash l : \text{Ref } \theta_1 \quad \Gamma \vdash l' : \text{Ref } \theta_2 \quad \Gamma \vdash \mathbf{fact} : \neg(l =_\theta l') \quad \Gamma, x : \theta_2 \vdash \mathbf{fact} : (l \sim \alpha)}{\Gamma \vdash \mathbf{fact} : (l \sim (\mathbf{Get } x : \theta_2 \Leftarrow l' \text{ in } \alpha))}$
$\sim \text{-intro-All}$ $\frac{\Gamma \vdash \mathbf{fact} : (l \sim \alpha_1) \quad \Gamma, x : \alpha_1 \vdash \mathbf{fact} : (l \sim \alpha_2)}{\Gamma \vdash \mathbf{fact} : (l \sim (\mathbf{All } x : \alpha_1) \alpha_2)}$	$\sim \text{-intro-Some}$ $\frac{\Gamma \vdash \mathbf{fact} : (l \sim \alpha_1) \quad \Gamma, x : \alpha_1 \vdash \mathbf{fact} : (l \sim \alpha_2)}{\Gamma \vdash \mathbf{fact} : (l \sim (\mathbf{Some } x : \alpha_1) \alpha_2)}$
$\sim \text{-intro-equality}$ $\Gamma \vdash \mathbf{fact} : (l \sim (e_1 = e_2 \text{ in } \alpha))$	$\sim \text{-intro-typeless-equality}$ $\Gamma \vdash \mathbf{fact} : (l \sim (e_1 =_\theta e_2))$

Figure B.11: Inference rules for noninterference in the α -layer.

<p><i>Assert-intro</i></p> $\frac{\Gamma \vdash t : \alpha_1}{\Gamma \vdash t : \{\alpha_1\}}$	<p><i>Assert-elim</i></p> $\frac{\Gamma \vdash t : \{\alpha_1\}}{\Gamma \vdash t : \alpha_1} \quad (\text{if } \Gamma \text{ has no } \{\alpha\} \text{ types})$
<p><i>Obs-intro</i></p> $\frac{\Gamma \vdash t : \{\alpha\}}{\Gamma \vdash t : \{\mathbf{Obs} \alpha\}}$	<p><i>Obs-elim</i></p> $\frac{\Gamma \vdash t : \{\mathbf{Obs} \tau\}}{\Gamma \vdash t : \{\tau\}} \quad (\text{if } \Gamma \text{ has only } \tau \text{ types})$
<p><i>Dereference</i></p> $\frac{\Gamma \vdash l : \mathbf{Ref} \theta \quad \Gamma, x: \theta, \{\mathbf{Get} z: \theta \leftarrow l \text{ in } x = z\} \vdash t : \{\mathbf{Obs} \alpha\}}{\Gamma \vdash (\mathbf{get} x \leftarrow l \text{ in } t) : \{\mathbf{Get} x: \theta \leftarrow l \text{ in } \alpha\}}$	
<p><i>Assignment</i></p> $\frac{\Gamma \vdash l : \mathbf{Ref} \theta \quad \Gamma, x: \theta \vdash \{(l \sim \alpha)\} \quad \Gamma \vdash e : \theta \quad \Gamma \vdash t : (\mathbf{Get} x: \theta \leftarrow l \text{ in } \alpha)}{\Gamma \vdash (l := e ; t) : \{\mathbf{Obs} \alpha[e/x]\}}$	
<p><i>Creation</i></p> $\frac{\Gamma \vdash e : \theta \quad \Gamma \vdash l_1 : \mathbf{Ref} \theta_1 \quad \dots \quad \Gamma \vdash l_n : \mathbf{Ref} \theta_n \quad \Gamma, v^*: \mathbf{Ref} \theta, \neg(v^* =_{\theta} l_1), \dots, \neg(v^* =_{\theta} l_n) \vdash t : (\mathbf{Get} x: \theta \leftarrow v^* \text{ in } \alpha)}{\Gamma \vdash (\mathbf{letref} v^* := e \text{ in } t) : \{\mathbf{Obs} \alpha[e/x]\}}$	
<p><i>Some -introduction</i></p> $\frac{\Gamma \vdash s : \{S\} \quad \Gamma \vdash t : \{T[s/x]\}}{\Gamma \vdash \langle s, t \rangle : \{(\mathbf{Some} x: S)T\}}$	<p><i>Some -elimination</i></p> $\frac{\Gamma \vdash e : \{(\mathbf{Some} x: S)T\}}{\Gamma \vdash e_{\cdot 1} : \{S\}} \quad \frac{\Gamma \vdash e : \{(\mathbf{Some} x: S)T\}}{\Gamma \vdash e_{\cdot 2} : \{T[e.1/x]\}}$
<p><i>All -introduction</i></p> $\frac{\Gamma, x: \{S\} \vdash t : \{T\}}{\Gamma \vdash \underline{\lambda}x.t : \{(\mathbf{All} x: S)T\}}$	<p><i>All -elimination</i></p> $\frac{\Gamma \vdash f : \{(\mathbf{All} x: S)T\} \quad \Gamma \vdash s : \{S\}}{\Gamma \vdash f(\underline{s}) : \{T[s/x]\}}$

Figure B.12: Inference rules for state-assertions.

Bibliography

- [1] S. Abramsky. Computational interpretations of linear logic. Research Report DOC 90/20, Imperial College, London, Oct 1990.
- [2] S. Allen. A non-type-theoretic semantics of Martin-Löf's types. In *IEEE Symposium on Logic in Computer Science*, pages 215–224, June 1987.
- [3] R. Backhouse, P. Chisholm, G. Malcolm, and E. Saaman. Do-it-yourself type theory. *Formal Aspects of Computing*, 1(1):19–84, 1989.
- [4] R. M. Burstall and B. Lampson. A kernel language for abstract data types and modules. In G. Kahn, D. B. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *LNCS*. Springer-Verlag, 1984.
- [5] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1986.
- [6] R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, New York, 1986.
- [7] T. Coquand. On the analogy between propositions and types. In G. Huet, editor, *Logical Foundations of Functional Programming*, chapter 17, pages 399–417. Addison-Wesley, 1990.
- [8] T. Coquand and G. Huet. Constructions: A higher order proof system for mechanizing mathematics. volume 203 of *LNCS*, pages 151–184. Springer-Verlag, 1985.
- [9] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.

- [10] H. B. Curry and R. Feys. *Combinatory Logic*, volume 1. North-Holland, Amsterdam, 1958.
- [11] N. G. de Bruijn. The mathematical language AUTOMATH, its usage and some of its extensions. In *Proc. Symposium on Automatic Demonstration (1968)*, volume 125 of *Lecture Notes in Mathematics*, pages 29–61. Springer-Verlag, 1970.
- [12] N. G. de Bruijn. A survey of the project Automath. In J.R. Hindley and J.P. Seldin, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 579–606. Academic Press, 1980.
- [13] N. Dershowitz. Well-founded orderings. Technical Report ATR-83(8478)-3, Aerospace Corp., El Segundo, CA, May 1983.
- [14] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3:69–116, 1987. Corrigendum in Vol 4, pages 409–410, 1987.
- [15] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, 1976.
- [16] J.E. Donahue and A.J. Demers. Revised report on Russell. Technical Report TR 79-389, Cornell University, Computer Science Department, 1979.
- [17] 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).
- [18] S. Feferman. A language and axioms for explicit mathematics. In J.N. Crossley, editor, *Algebra and Logic*, volume 450 of *Lecture Notes in Mathematics*, pages 87–139. Springer-Verlag, 1975.
- [19] M. Felleisen. lambda-v-cs: An extended lambda-calculus for scheme. In *ACM Symposium on LISP and Functional Programming*, 1988.
- [20] M. Felleisen and D. P. Friedman. A calculus for assignments in higher-order languages. In *ACM Symposium on Principles of Programming Languages*, pages 314–325, 1987.
- [21] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. Technical Report COMP TR89-100, Rice University, 1989.

- [22] J. H. Gallier. On Girard's candidats de reductibilite. In *Logic and Computer Science*, pages 123–203. Academic Press Ltd., 1990.
- [23] D.K. Gifford and J.M. Lucassen. Integrating functional and imperative programming. In *ACM Symposium on LISP and Functional Programming*, pages 28–38, 1986.
- [24] J-Y. Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, University of Paris, 1972.
- [25] J.-Y. Girard. Linear logic. *Theoretical Comp. Science*, 50:1–102, 1987.
- [26] J-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge University Press, 1989.
- [27] J.C. Guzman and P. Hudak. Single-threaded polymorphic lambda calculus. In *IEEE Symposium on Logic in Computer Science*, pages 333–343, 1990.
- [28] S. Hayashi and H. Nakano. *PX : A Computational Logic*. Foundations of Computing. MIT Press, Cambridge, MA, 1989.
- [29] M.C. Henson. Program development in the constructive set theory TK. *Formal Aspects of Computing*, 1:173–192, 1989.
- [30] M.C. Henson and R. Turner. A constructive set theory for program development. In *Proc. Foundations of Software Technology and Theoretical Computer Science*, volume 338 of *LNCS*, pages 329–347. Springer-Verlag, 1988.
- [31] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–583, 1969.
- [32] C. A. R. Hoare et al. Laws of programming. *Communications of the ACM*, 30(8):672–686, August 1987.
- [33] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, NY, 1980.

- [34] P. Hudak and A. Bloss. The aggregate update problem in functional programming systems. In *ACM Symposium on Principles of Programming Languages*, pages 300–314, 1985.
- [35] P. Hudak, S. P. Jones, and P. Wadler (editors). Report on the programming language Haskell, A non-strict purely functional language (Version 1.2). *SIGPLAN Notices*, May 1992. To appear.
- [36] P. Hudak and R. Sundaresh. On the expressiveness of purely functional I/O systems. Technical Report YALEU/DCS/RR665, Yale University, Dec 1988.
- [37] G. Huet. Confluent reductions: abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, October 1980. (Previous version in *Proc. Symp. Foundations of Computer Science*, Oct 1977).
- [38] J. Hughes. Why functional programming matters. In *Research Topics in Functional Programming*, Univ. of Texas at Austin Year of Programming Series, chapter 2, pages 17–42. Addison-Wesley, 1990.
- [39] E. Ireland. Assignable variables and referential transparency. Functional Programming mailing list, June 1989.
- [40] K. Karlsson. Nebula, A functional operating system. Tech. report, Chalmers University, 1981.
- [41] G. Lindstrom. Functional programming and the logical variable. In *ACM Symposium on Principles of Programming Languages*, 1985.
- [42] J.M. Lucassen and D.K. Gifford. Polymorphic effect systems. In *ACM Symposium on Principles of Programming Languages*, pages 47–57, 1988.
- [43] D. MacQueen. Modules for Standard ML. In *ACM Symposium on LISP and Functional Programming*, pages 198–207, 1984. (Revised version in *Polymorphism Newsletter*, II, 2, October 1985).
- [44] Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill Book Company, 1974.

- [45] P. Martin-Löf. An intuitionistic theory of types: predicative part. In H. E. Rose and J. C. Shepherdson, editors, *Logic Colloquium III*, pages 73–118. North-Holland, Amsterdam, 1973.
- [46] P. Martin-Löf. Constructive mathematics and computer programming. In L. J. Cohen, J. Los, H. Pfeiffer, and K.P. Podewski, editors, *Proceedings of the Sixth International Congress for Logic, Methodology and Philosophy of Science*, volume 104 of *Stud. Logic Foundations Math.*, pages 153–175, Amsterdam, 1982. North-Holland.
- [47] P. Martin-Löf. *Intuitionistic Type Theory : notes by Giovanni Sambin of a series of lectures given in Padua, June 1980*. Studies in Proof Theory, Lecture Notes. Bibliopolis, Napoli, 1984.
- [48] I. A. Mason and C. Talcott. Axiomatizing operational equivalence in the presence of side effects. In *IEEE Symposium on Logic in Computer Science*, pages 284–293, 1989.
- [49] I. A. Mason and C. Talcott. A sound and complete axiomatization of operational equivalence between programs with memory. Technical Report STAN-CS-89-1250, Stanford University, 1989.
- [50] L. M. McLoughlin and S. Hayes. Interlanguage working from a pure functional language. Functional Programming mailing list, Nov 1988.
- [51] N. Mendler, R. Constable, and Panangaden P. Infinite objects in type theory. In *IEEE Symposium on Logic in Computer Science*, pages 249–257, June 1986.
- [52] P. F. Mendler. *Inductive Definition in Type Theory*. PhD thesis, Cornell University, 1987. Technical Report 87-870.
- [53] R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML*. The MIT Press, Cambridge, Massachusetts, 1990.
- [54] G. Mirkowska and A. Salwicki. *Algorithmic Logic*. PWN - Polish Scientific Publishers, Warszawa, Poland, 1987.

- [55] J. C. Mitchell. Type systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 8, pages 365–458. North-Holland, Amsterdam, 1990.
- [56] J. C. Mitchell and R. Harper. The essence of ML. In *ACM Symposium on Principles of Programming Languages*, pages 28–46. ACM, 1988.
- [57] J. C. Mitchell and G. D. Plotkin. Abstract types have existential types. In *ACM Symposium on Principles of Programming Languages*, pages 37–51, January 1985.
- [58] C. Mohring. Algorithm development in the calculus of constructions. In *IEEE Symposium on Logic in Computer Science*, pages 84–91, 1986.
- [59] B. Nordstrom. Programming in constructive set theory: some examples. In *Proc. ACM Conf. on functional programming languages and computer architecture*, pages 141–154, October 1981.
- [60] N. Perry. *The Implementation of Practical Functional Programming Languages*. PhD thesis, Imperial College of Science, Technology and Medicine, University of London, 1990.
- [61] K. Petersson. A programming system for type theory. LPM Memo 21, Dept. of Computer Science, Chalmers University of Technology, Goteborg, Sweden, 1982.
- [62] V. Pratt. Semantic considerations on Floyd-Hoare logic. In *IEEE Symposium on Foundations of Computer Science*, pages 109–121, 1976.
- [63] D. Prawitz. Ideas and results in proof theory. In *Proc. Second Scandinavian Logic Symposium*, 1971.
- [64] J. Rees and W. Clinger (editors). Revised³ report on the algorithmic language scheme. *ACM SIGPLAN Notices*, 21(12):37–79, Dec 1986.
- [65] J. C. Reynolds. Towards a theory of type structure. In *Coll. sur la Programmation*, volume 19 of *LNCS*, pages 408–425. Springer-Verlag, 1974.
- [66] 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.

- [67] J. C. Reynolds. Idealized Algol and its specification logic. In D. Neel, editor, *Tools and Notions for Program Construction*, pages 121–161. Cambridge University Press, 1982.
- [68] J. C. Reynolds. Three approaches to type structure. *Mathematical Foundations of Software Development*, 185:97–138, 1985.
- [69] J.C. Reynolds. Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159, Carnegie Mellon University, June 1988.
- [70] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.
- [71] D. A. Schmidt. Detecting global variables in denotational specifications. *ACM Transactions on Programming Languages and Systems*, 7(2):299–310, Apr 1985.
- [72] D. Scott. Constructive validity. In *Symposium on Automatic Demonstration*, volume 137 of *Lecture Notes in Mathematics*, pages 237–275. Springer-Verlag, 1970.
- [73] J. Smith. The identification of propositions and types in Martin-Löf’s type theory: a programming example. volume 158 of *LNCS*, pages 445–456. Springer-Verlag, 1983.
- [74] S.F. Smith. *Partial Objects in Type Theory*. PhD thesis, Cornell University, Department of Computer Science, August 1988.
- [75] S.F. Smith and R.L. Constable. Partial objects in constructive type theory. In *IEEE Symposium on Logic in Computer Science*, pages 183–193, 1987.
- [76] S. Stenlund. *Combinators, Lambda-Terms and Proof Theory*. Reidel, D., Dordrecht, Holland, 1972.
- [77] J. E. Stoy. *Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory*. MIT Press, 1977.
- [78] C. Strachey and C. P. Wadsworth. Continuations - a mathematical semantics for handling full jumps. Tech. Monograph PRG-11, Programming Research Group, University of Oxford, 1974.

- [79] V. Swarup. The UIPRL proof development system. Technical Report UIUCDCS-R-88-1434, University of Illinois at Urbana-Champaign, 1988.
- [80] V. Swarup and U.S. Reddy. A logical view of assignments. In *Proc. Conf. on Constructivity in Computer Science*, June 1991.
- [81] V. Swarup, U.S. Reddy, and E. Ireland. Assignments for applicative languages. In *Conference on Functional Programming Languages and Computer Architecture*, volume 523 of *LNCS*, pages 192–214. Springer-Verlag, August 1991.
- [82] W. W. Tait. A realizability interpretation of the theory of species. In R. Parikh, editor, *Proceedings of Logic Colloquium*, volume 453 of *Lecture Notes in Mathematics*, pages 240–251. Springer-Verlag, Berlin, 1975.
- [83] 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.
- [84] M. Tofte. *Operational semantics and polymorphic type inference*. PhD thesis, Edinburgh University, 1988. Available as Edinburgh Univ. Lab. for Foundations of Computer Science Technical Report ECS-LFCS-88-54.
- [85] P. Wadler. Comprehending monads. In *ACM Symposium on LISP and Functional Programming*, 1990.
- [86] P. Wadler. Linear types can change the world. In *IFIP Working Conf. on Programming Concepts and Methods*, Sea of Gallilee, Israel, Apr 1990.
- [87] P. Wadler. Is there a use for linear logic? In *Proc. ACM SIGPLAN Conf. on Partial Evaluation and Semantics-Based Program Manipulation*, volume 26(9) of *SIGPLAN Notices*, pages 255–273, September 1991.
- [88] D. H. D. Warren, L. M. Pereira, and F. Pereira. Prolog - the language and its implementation compared with Lisp. *SIGPLAN Notices*, 12(8), 1977.

Vita

Vipin Swarup was born in Redwood City, California on June 28, 1962. He grew up in India, and in 1984 he graduated from the Indian Institute of Technology, Bombay with a B.Tech. degree in Computer Science and Engineering. He entered the graduate program at the University of Illinois at Urbana-Champaign, and in 1988 was awarded an M.S. degree in Computer Science. In the fall of 1990, he left the confines of Urbana-Champaign and moved to the Boston area. He completed his Ph.D. thesis while working at the MITRE Corporation.