Computational Type Theory

Robert Harper
Carnegie Mellon University

Draft of July 16, 2018

Preface

The central dogma of constructive type theory may be expressed in one sentence:

A sufficiently expressive programming language provides a foundation for all of mathematics.

This simply-stated principle, implicit in Brouwer's intuitionistic foundation for mathematics and made explicit in Martin-Löf's type theory, unifies computation and mathematics. Computer science, rather than being a branch of mathematics, is instead a foundation for it. And programming languages, rather than being arbitrary inventions, are but rigorous expressions of mathematical thought.

The purpose of this monograph is to explain and to explore the computational meaning of constructive type theory. Rather than being a sterile formal system, a type theory ought to express and to solve programming problems that state and solve mathematical problems.

Pittsburgh
Spring, 2018

Acknowledgements

Jon Sterling, Carlo Angiuli.

Things Left To Do

Chapter 1

Introduction

Intuitionistic type theory is a language in which to express constructive mathematics. There are at least two, somewhat different, ways of interpreting this statement. As formulated by Brouwer, what distinguishes constructive mathematics is that all mathematical objects are constructions that can, at least in principle, be carried out by any human being by virtue of our innate understanding of computation. In particular, proofs (in the most general sense of evidence for the truth of propositions) are themselves forms of construction, and hence must amount to computations. As formulated by many of Brouwer's followers, constructive mathematics is distinguished from classical mathematics by the absence of certain principles such as, most famously, the law of the excluded middle. The distinction between these two points of view is that between semantics and formalism. Semantic constructivism is committed to mathematics having a computational meaning; syntactic, or formal, constructivism is committed to exploring the axiomatic freedom afforded by starting with fewer axiomatic commitments. These two conceptions of constructivism have encouraged two distinct formulations of type theory itself, one semantic, one axiomatic.

A *formal*, or *axiomatic*, *type theory* is given by a collection of (axioms and) rules for deriving *(formal) judgments* of the form $\Gamma \vdash A\,\mathsf{type}$, $\Gamma \vdash M : A$, $\Gamma \vdash A \equiv B$, and $\Gamma \vdash M \equiv N : A$, which state, respectively, what are the types, what are the terms of a type, and when two types, or two elements of a type, "definitionally" equivalent.[1] These judgments are all made relative to a context, Γ, specifying the types of the variables involved in the types and terms. The only form of judgment in formal type theory is the assertion that a formal judgment is derivable according to the rules, but it is customary to not distinguish the formal judgment from the judgment that it is derivable. It is inherent in formal type theory that the derivable judgments are semi-decidable, because the derivable judgments are computably enumerable. Methodologically, it is usual to demand that they be moreover decidable, perhaps relative to some pre-suppositions, so that it is possible, at least in principle, to check mechanically whether or not a term has a given type.

All formal logics are defined in a similar manner. For example, the classical predicate calculus is a formal system for deriving judgments of the form $\Delta \vdash t\,\mathsf{term}$, $\Delta ; \Gamma \vdash \phi\,\mathsf{prop}$ and $\Delta ; \Gamma \vdash \phi\,\mathsf{true}$, which define the terms, propositions, and truths, relative to contexts Δ declaring the active vari-

[1]The terminology is mysterious, but firmly established.

ables and assumptions Γ posulating truths. Most importantly, the latter judgment presents a *logical consequence relation*, which means that it validates reflexivity (assumed truths are truths) and transitivity (substitution for assumptions), and, in the case of classical logic, also validates weakening and contraction (variables and hypotheses may be used zero or more times). The same principles apply to formal type theory, and correspond to substitutions acting on types and terms.

One may regard type theory as just another formal logic, but to do so overlooks its *intended meaning* in which types and terms are computations. Propositions are meant to be specifications of programs, or constructions, that witness, or substantiate, their truth in the sense of Brouwer. The classical predicate calculus, by contrast, has no intended meaning, but rather a range of meanings compatible with given assumptions (axioms).[2] The intended meaning of types and terms as computations expresses the very essence of constructivism as advanced by Brouwer.

However, all good intentions aside, formal type theory has no intrinsic computational (or any other) meaning. It therefore can only be construed as constructive only in the weak sense that certain judgments are not derivable according to the rules. For example, the type $A + (A \to \mathbf{0})$ need not be inhabited for every type A, though it may well be for certain of these, which are said to be *decidable*. Absent a computational interpretation, it is a matter of curiosity why it should be that certain types are decidable, and others not. Were there a computational interpretation, though, a decidable type would be one for which there is an effective decision procedure to affirm or refute the proposition A. Not all A's need admit such a decision procedure, though nothing precludes there being one.[3]

The intended computational content of a constructive formal type theory is provided by *semantic*, or *computational*, *type theory*, which is a general theory of specification and verification of program behavior. Following intuitionistic principles, the concept of computation is given (perhaps partially) at the outset as a collection of programs that are either fully evaluated, called *canonical*, or are capable of being simplified by a further step of evaluation. Evaluation is deterministic in that there is ever at most one next step to be taken, and it is self-evident how to take that step. Thus, program evaluation can be carried out by a robot with little or no reasoning ability, a fundamental precept of the theory. With the notion of program in hand, types are defined as certain programs whose canonical form determines a specification of program behavior. The elements of a type are those programs that behave according to the specification given by the type.

Computational type theory is defined by two *basic judgments*, written $A \doteq A'$ and $M \doteq M' \in A$, stating that the programs A and A' designate the same specification, and that M and M' behave equally according to the specification given by A. The derived judgment $A\ \mathsf{type}$ means that $A \doteq A$, and the derived judgment $M \in A$ means that $M \doteq M \in A$, presupposing that $A\ \mathsf{type}$. The equality judgments are said to express *exact equality* in the sense that it captures the full meaning of types and their members, in keeping with Quine's precept, "No entity without identity!" Thus, to know that a program is a type or is a member necessarily means to know when it is equal to any other type or any other member of the given type. Importantly, member equality is determined by the type: the same two programs may be equal as members of one type and not equal as members of another.

[2]One could argue that propositions are meant to stand for booleans, but even that assumption is relaxed in mathematical accounts in order to permit a completeness theorem for a class of structures.

[3]Even if A is not *recursively* decidable, one may nevertheless consider an oracle that decides, for example, the self-halting problem for Turing machines.

The *hypothetical judgments* $a : A \gg B \doteq B'$ and $a : A \gg N \doteq N' \in B$, mean, respectively, that $B[M/a] \doteq B'[M'/a]$ and $N[M/a] \doteq N'[M'/a] \in B[M/a]$, whenever $M \doteq M' \in A$. (These generalize in a systematic way to the case of more than one variable or hypothesis.) The hypothetical judgments express two important ideas about variables:

1. A variable is a *placeholder* for members of its type.
2. Variation *respects equality* of members of its type.

Thus, being a type under a hypothesis is to be a *family of types* indexed by a variable in that all instances obtained by plugging in for that variable give rise to types, and, moreover, equal instances give rise to equal types. Similarly, being an element of a type under a hypothesis is to be a *family of members*, that is, a function or mapping, that sends instances to results in such a way as to respect equality of instances.

The judgments of formal type theory are semantic, rather than formal in that they express claims about program behavior. Unlike derivability in formal type theory, there is no *a priori* bound on the difficulty of acquiring or conveying the knowledge expressed by them. Whether a program is a member of a type, or even whether a program is a type at all, is a matter of *semantic truth*, not a matter of *formal proof*. A formal type theory may be considered to be constructive when its derivable judgments are true according to the following scheme:

1. If $\Gamma \vdash A\ \mathsf{type}$, then $|\Gamma| \gg |A| \doteq |A|$.
2. If $\Gamma \vdash M : A$, then $|\Gamma| \gg |M| \in |A|$.
3. If $\Gamma \vdash A \equiv A'$, then $|\Gamma| \gg |A| \doteq |A'|$.
4. If $\Gamma \vdash M \equiv M' : A$, then $|\Gamma| \gg |A| \doteq |A|$ and $|\Gamma| \gg |M| \doteq |M'| \in |A|$.

The interpretation is relative to an *extraction*, operation, notated by $|\cdot|$, which removes syntactic clutter in preparation for execution by the robot, which neither knows nor cares about formalism. If a candidate formal type theory does not validate such a property, then it cannot be considered constructive. If it is constructive, then the computational interpretation implies that certain judgments, such as the universal inhabitation of $A + (A \to \mathbf{0})$, will not be derivable. Moreover, because of the inherent limitations of the axiomatic method, by Gödel's Theorem no formal type theory can provide a definitive account of the truth.

The importance of type theory for constructive mathematics arises from the *propositions-as-types* principle, whereby propositions correspond to types and proofs of propositions correspond to members of the corresponding type. The semantic formulation of this principle, which was historically emphasized by Martin-Löf, captures the inuitionistic concept of truth. According to this view, mathematics is a social process whose principle act consists of conveying mathematical knowledge among people. For this to be possible, there must be a common language with which to express mathematical thought. Mathematics being concerned with the infinite, and communication being necessarily finite, there must be some way to convey an infinite amount of information in a finite amount of time. What makes this possible is the innate, evolutiuonarily determined, concept of an algorithm, or construction, on which language acquisition and mathematical discourse are based. Thus, the natural numbers, in all their infinite glory, can be given by counting by ones starting from zero, as every child understands. Facts about the natural numbers, such as the infinitude of the primes, can, indeed must, be conveyed by a program, in this case by computing an upper bound on the size of a strictly larger prime than a given one. Thus, mathematical objects, including proofs, are programs.

Computational type theory is precisely a language for expressing mathematics in a manner consistent with intuitionistic principles. Mathematical constructions, such as the type N of natural numbers or the type $\mathsf{N} \to \mathsf{N}$ of functions on them describe the behavior of programs, in the former case evaluating to a number, and in the latter transforming numbers into numbers by the mechanical process of evaluation. Because it specifies equality of types and their members, these types also tell us when two numbers are equal (when both are zero or both are successors of equal numbers) and when two functions on the natural numbers are equal (when they have the same input/output behavior). Mathematical proofs are forms of construction whose types express the (constructive) truth conditions for a proposition. For example, the truth of an implication is a function transforming proofs of the antecedent to proofs of the consequent. Being a function, it must respect equality of proofs, as must any other function in the theory. In general a proposition expresses an aspiration (namely, to be proved true) that is fulfilled by a program that satisfies its truth requirements.

The formal version of the propositions-as-types principle is popularly known as the *Curry-Howard Correspondence*.[4] The formal correspondence codifies the structure of formal proofs in Gentzen's systems of natural deduction as the formal members of the type corresponding to the proposition proved. Thus, the formal judgment $A\ \mathsf{type}$ corresponds to the judgment $A\ \mathsf{prop}$ in a formal constructive logic, and the judgment $M : A$ corresponds to the judgment $A\ \mathsf{true}$ in formal logic, with M being the representation of the derivation of A in natural deduction. It is historically remarkable that the terms of formal type theory, which were introduced as notations for programs, and the derivations introduced by Gentzen, which codified familiar informal practices, turned out to coincide. But, once noticed, the correspondence is also rather elementary: the rules that govern the binding and scoping of variables in a program are exactly those that govern the positing and discharging of hypotheses in a proof. Church's λ-notation provides an elegant, and by now standard, way to unify these concepts.

The formal correspondence also addresses equality of proofs based on Gentzen's analysis of derivations in natural deduction. Gentzen's criterion is based on a normative principle for systems of natural deduction that classifies rules as either *introductory* or *eliminatory* for a connective. The introductory rules define the information that goes into a derivation of a proposition built from that connective, and the eliminatory rules define the information that can be obtained from such a derivation. The *inversion principle*, which might also be called a conservation principle, states that the eliminatory rules are post-inverse to the introductory rules—no more can be extracted from a derivation than was put into it in the first place. Derivations may be simplified by cancellation of an eliminatory rule applied to an introductory rule, avoiding the needless digression. When all such occurrences are cancelled, the derivation is said to be in *normal form*. Two derivations are considered the same whenever they have the same normal form. Transposed to type theory,

[4]So called, though doing so denies credit to the many others who contributed to its discovery and promoted its use. The correspondence is often elevated to an "isomorphism", which is wholly unjustified, not least because there is no *a priori* concept of equality of formal proofs on which to base the claim.

definitional equivalence exactly expresses the inversion principle.[5]

Definitional equivalence is usually described as *intensional* in the sense that two derivations are considered to be the same if they exhibit the same structure modulo inversions. This is constrasted with an *extensional* formulation of proof equality in which, for example, two proofs of an implication are equated whenever they have the same input/output behavior. Thus, exact equality is said to be extensional, in contrast to definitional equivalence, which is said to be intensional. But this description is misleading, because in any case functions preserve (the relevant notion of) equality! Two function abstractions are definitionally equivalent whenever they give equivalent results for equivalent arguments, which is exactly the principle of function extensionality. The same property holds for exact equality: exactly equal arguments are sent to exactly equal results.

In dependent type theory member equalities induce type equalities, because families respect equality of instances. But type equality usually means more than just equality of instances of families. For example, in the presence of inductive types one may compute types as a function of data values, using a discriminator of some form. More importantly, one may wish to "equate" types that are merely isomorphic on the grounds that types are classifiers and the isomorphism provides a way to "transport" elements across the isomorphism. The trouble with this description is that isomorphism is a structure consisting, at least, of pairs of maps, whereas equality is a property that "at most holds" with no further evidence required or available.

Thus something other than equality is needed to account for isomorphism as a principle of interchangeability of types. The required concept is called an *identification*, or a *path*, both within a type (that is, among its members) and between types. Paths should be reflexive (there is a trivial path from an object to itself), reversible (symmetric), and concatenable (transitive). As the terminology suggests, type identifications are paths along which elements can be transported from one type to the other. In the presence of *universes*, which are types whose members are types, type identifications become member identifications coarser than equality. Because of this, the notion of type isomorphism has to be generalized to *(homotopy) equivalence*, which permits the maps to be mutually inverse up to further identification—equivalence is "isomorphism up to identification."

Voevodsky's *univalence principle*, which states that equivalent types are identified, has far-reaching consequences, both conceptually and pragmatically. In practice univalence elevates the expressive power of type theory as a language in which to formulate mathematics, because it obliterates fine distinctions that would otherwise obstruct the flow of an argument. As a principle of programming languages, univalence constitutes a powerful concept of implicit coercion between types, allowing the automatic interchange of their values in any context, avoiding the clutter that would otherwise obscure the clarity of the code.

What follows is a systematic study of type theory carried out with these goals in mind:

1. To develop the computational semantics of type theory.

2. To relate formal to semantic accounts of type theory.

3. To clarify the meaning of equality in type theory.

4. To enrich type theory to account for identification.

[5]Some variations have been considered in the literature, but it seems that the baseline is to consider only inversions as the generators for definitional equivalence.

The outcome will be a dependent type theory with a computational account of univalence and higher-dimensional inductive types.

The development begins with the negative propositional types (unit, product, and function), which correspond to the negative propositional connectives (truth, conjunction, and implication). The primary goal is to introduce Tait's method, a fundamental tool for expressing the computational semantics of type theory. This method comes in two forms, one that establishes the logical consistency of the theory, and one that furthermore accounts for equality of types and at each type. The secondary goal is to define and analyze a formal account of unit, product, and function types in which definitional equivalence is defined according to the inversion principle.

Accounting for positive types (void and sums, corresponding to falsehood and disjunction) are more difficult to handle formally, but present no difficulties semantically. The computational semantics is extended to account for sums, obtaining a canonicity property stating, for example, that every boolean-typed closed expression evaluates to true or false. In the formal setting definitional equivalence for positive types poses challenges to account for reasoning by induction (cases). It is possible to account for finitary positive types, but these methods do not scale to infinitary types such as the natural numbers.

An important method for handling positive types in the formal setting is to develop a non-standard computational semantics in which evaluation is performed on terms with free variables. Doing so allows for forms of evaluation that descend within the scopes of binders to define normal forms. The semantics for this form of evaluation makes use of Kripke-like, or pre-sheaf, interpretations. Besides its intrinsic interest these interpretations foreshadow the use of variables to present higher-dimensional structure. In that setting standard evaluation is on terms with no free ordinary variables, but which may involve free variables specifying variation in a dimension. Pre-sheaf methods are therefore required in the higher-dimensional case.

Predicative type theories, as developed by Martin-Löf, admit a stratified semantics in which the meaning of a compound type is given in terms of the meanings of its constituent types. Impredicative theories, as advocated by Girard, do not admit such a stratification; stronger semantic methods are therefore required. The key idea, known as Girard's method, is to define the collection of "all possible types", called *type candidates*, and to admit quantification over this collection as a form of type. The binary form of the candidates method accounts for Reynolds' concept of *parametricity* as a means of reasoning about abstract types.

Dependent types are those whose meaning may depend not only on other types, but also on elements of other types. Dependent types may be viewed as families of types indexed by another type in the sense that each member of the indexing type gives rise to a type in such a way that equal indices (according to the type) determine equal instances. Type families are used to represent predicates and relations, which are propositions about the members of some type. The negative connectives generalize to dependent forms in which the result type of a function may depend on its argument, and the type of the second component of a pair may depend on the first component (itself, not just its type). The eliminatory forms for the positive connectives must also be generalized to account for dependency on the eliminated expression in the type of the eliminator. Dependency raises the issue of type equality, which motivates and informs much of the rest of the book.

Equality of members of a type may be expressed as a type family indexed by pairs of members of that type. Semantically, such equality types are inhabited by at most one element, which witnesses that the stated equation holds. Using equality types it is possible to derive the extensionality principle for functions and to formulate induction principles for establishing properties of the members of a type. Formally, it is not possible to axiomatize equality in the semantic sense, because all axiomatic theories are relentlessly recursively enumerable, whereas the quantifier complexity of semantic equality varies with the structure of the type. The usual approach is to formulate the least reflexive family of relations as a type, called the *identification*, or *identity*, type. Crucially, the identification type is defined uniformly in the underlying type, avoidng the complexities of type-specific reasoning. Indeed, for closed terms the least reflexive family is characterized exactly by definitional equivalence. The elimination principle witnesses the interchangeability of types in this relation, a necessary condition for it to be interpreted as a form of equality.

Homotopy type theory arises by considering additional identifications beyond reflexivity, while retaining the universal properties of the identification type. There are two main sources of identifications, univalence and higher inductive types. Univalence permits any homotopy equivalence of types to be regarded as identifying them. Higher inductive types permit the axiomatic definition of types that contain path, as well as point, generators. These extensions can be given a mathematical justification in terms of simplicial sets, but a computational interpretation is lacking, and may be impossible to define. The difficulty is, how is the elimination form to compute on non-reflexive identifications?

To resolve this question requires a full development of the higher-dimensional structure of types that is suggested by the addition of non-reflexive identifications. A natural formulation is to consider that types have the structure of *cubes*. In the cubical setting ordinary elements are points of the type, and identifications are paths between lower-dimensional cubes. Maps are required to respect the cubical structure of their domain, sending cubes of arbitrary dimension in the domain to cubes of the same dimension in the codomain. Cubes are given by terms that have free dimension variables that, implicitly, range over the unit interval, and n-cube having up to n such variables within it. Along a dimension x the left and right faces of the cube are given by substituting 0 and 1 for x, respectively. Any cube can be thought of as vacuously involving any dimension variables, so that dimensions are closed under weakening, or degeneracy. Substituting one dimension variable for another traces out the diagonal of the square given by these two variables, corresponding to closure under contraction.

All of the usual type constructors extend to higher dimensions in a natural way. Moreover, there are new type constructors induced by the higher-dimensional strucuture, notably the *path type* whose points represent lines one dimension higher. Paths may be constructed using any of the other constructs of type theory, including induction principles that map out of positive types. The higher-dimensional structure of the "multiverse" of types [6] admits identifications given by the univalence principle. This structure may be internalized as the elements of a universe of types, much as it arises in homotopy type theory. The difference, however, is that the type theory is inherently computational in that all types and members are programs, and which enjoys the same canonicity properties as its zero-dimensional forerunners.

[6]The use of this neologism is forced by prior usage of the term "universe" to mean a type of types that, emphatically, is not universal!

Chapter 2

The Framework

The technical development of type theory is structured into three major components:

1. A *semantic framework* for defining semantic judgments that express the meanings of types as specifications of program behavior.

2. A *syntactic framework* for defining rules for deriving formal typing and equivalence judgments.

3. An *interpretation* of the syntax into semantics that demonstrates the semantic validity of formally derivable judgments.

This chapter describes these components in enough generality for the development of dependent type theory. To account for higher dimensions this account must be further generalized to equip types and elements with the structure of "cubes."

2.1 Semantic Framework

The computational meaning of types is given in terms of an *untyped* programming language whose forms of expression are notated in this color. The execution of programs is specified by an inductive definition of the judgments $E\ \mathsf{val}$, defining the *values*, and $E \longmapsto E'$, defining one step of *simplification*, on closed untyped terms. Values are final states in that if $E\ \mathsf{val}$, then there is no E' such that $E \longmapsto E'$. Simplification is always *deterministic* in that if $E \longmapsto E_1$ and $E \longmapsto E_2$, then E_1 is E_2. These judgments induce an *evaluation* judgment, written $E \Downarrow V$ and defined by $E \longmapsto^* V$ and $V\ \mathsf{val}$, stating that E evaluates to V. Determinacy ensures that this relation is the graph of a partial function of E.

Definition 2.1. *A* semantic type system *defines two forms of judgment, called* exact equality *judgments, on closed untyped terms:*

1. *Equality of types, written* $A \doteq A'$*,*

2. *For A such that $A \doteq A$, equality of elements of A, written $M \doteq M' \in A$.*

These judgments satisfy the following requirements:

1. *Type and member equality are symmetric and transitive.*
2. *Equal types determine the same member equality: if $A \doteq A'$, then $M \doteq M' \in A$ iff $M \doteq M' \in A'$.*
3. *Equality preserves and reflects evaluation: $A \doteq A'$ iff $A \Downarrow A_0$, $A' \Downarrow A_0'$, and $A_0 \doteq A_0'$, and $M \doteq M' \in A$ iff $A \Downarrow A_0$, $M \Downarrow M_0$, $M' \Downarrow M_0'$, and $M_0 \doteq M_0' \in A_0$.*

The exact equality judgments induce two auxiliary judgments, called *type formation* and *type membership* judgments on closed untyped terms:

1. Type formation, written A type, meaning $A \doteq A$.
2. Membership in A, where A type, and written $M \in A$, meaning $M \doteq M \in A$.

These satisfy the conditions that equal types determine the same members, and that both type formation and type membership preserve and reflect evaluation. Note that if $A \doteq A'$, then by symmetry and transitivity $A \doteq A$, which is to say A type. Similarly, if $M \doteq M' \in A$, then $M \in A$.

Lemma 2.1 (Closure Under Evaluation). *1. If $A \doteq A'$, then $A \longmapsto B$ implies $B \doteq A'$, and $A' \longmapsto B'$ implies $A \doteq B'$.*

2. If $M \doteq M' \in A$, then $M \longmapsto N$ implies $N \doteq M' \in A$ and $M' \longmapsto N'$ implies $M \doteq N' \in A$.

Proof. Determinacy of evaluation ensures that if $E \Downarrow V$ and $E \longmapsto E'$, then $E' \Downarrow V$. □

Lemma 2.2 (Closure Under Reverse Evaluation). *1. If $A \longmapsto B$ and $B \doteq A'$, then $A \doteq A'$. Therefore if $A' \longmapsto B'$ and $A \doteq B'$, then $A \doteq A'$.*

2. If $M \longmapsto N$ and $N \doteq M' \in A$, then $M \doteq M' \in A$. Therefore if $M' \longmapsto N'$ and $M \doteq N' \in A$, then $M \doteq M' \in A$.

Proof. Evaluation is itself closed under reverse evaluation: if $F \Downarrow V$ and $E \longmapsto F$, then $E \Downarrow V$. □

A *typing context*, Γ, is a finite sequence of *declarations*, or *bindings*, of the form $x_1 : A_1, \ldots, x_n : A_n$ with *domain* $x_1, \ldots, x_n$. For contexts Γ_1 and Γ_2 with disjoint domain, the context $\Gamma_1\,\Gamma_2$ is their concatentation. The empty context is written $\bullet$. A non-empty context can always be written in the form $\Gamma\, , a : A$, displaying its last declaration. A *substitution*, γ, is an assignment of a closed term to each of a finite set of variables, its *domain*.

Exact equality of types and members extends to exact equality of contexts and substitutions, variable by variable. The judgment $\Gamma \doteq \Gamma'$ is means that Γ and Γ' have the same domain and assign equal types to each variable in their common domain. In particular, the order of declarations in the two contexts is irrelevant to their equality. The judgment $\gamma \doteq \gamma' \in \Gamma$ means that Γ, γ, and γ' have the same domain, and that the substitutions assign equal elements of the declared type for each variable in their common domain. As with types and their elements, the judgment Γ ctx is defined to mean $\Gamma \doteq \Gamma$, and $\gamma \in \Gamma$ is defined to mean $\gamma \doteq \gamma \in \Gamma$.

The *hypothetical*, or *general*,[1] judgments, $\Gamma \gg A \doteq A'$ and $\Gamma \gg M \doteq M' \in A$, express *functionality* in the specified variables. The judgment $\Gamma \gg A \doteq A'$ means that if $\gamma \doteq \gamma' \in \Gamma$, then $A\,\gamma \doteq A\,\gamma'$. Similarly, the judgment $\Gamma \gg M \doteq M' \in A$ means that if $\gamma \doteq \gamma' \in \Gamma$, then $M\,\gamma \doteq M'\,\gamma' \in A\,\gamma$.

Lemma 2.3. *The judgment $\Gamma \gg A \doteq A'$ is equivalent to the disjunction of two conditions:*

1. *either $\Gamma = \bullet$ and $A \doteq A'$, or*
2. *$\Gamma = \Gamma'\,, a : B$, where $\Gamma' \gg B$ type, and if $\Gamma' \gg N \doteq N' \in B$, then $\Gamma' \gg A\,[N/a] \doteq A'\,[N'/a]$.*

If $\Gamma \gg A$ type, then, $\Gamma \gg M \doteq M' \in A$ is equivalent to the disjunction of two conditions:

1. *either $\Gamma = \bullet$ and $M \doteq M' \in A$, or*
2. *$\Gamma = \Gamma'\,, a : B$ such that $\Gamma' \gg B$ type, and if $\Gamma' \gg N \doteq N' \in B$, then $\Gamma' \gg M\,[N/a] \doteq M'\,[N'/a] \in A$.*

Lemma 2.4. 1. *The hypothetical equality judgments are symmetric and transitive.*

2. *The hypothetical membership judgment respects type equality.*

Proof. Suppose that $\Gamma \gg A \doteq A'$ in order to show that $\Gamma \gg A' \doteq A$. To this end suppose that $\gamma' \doteq \gamma \in \Gamma$. By symmetry $\gamma \doteq \gamma' \in \Gamma$, so that by assumption $A\,\gamma \doteq A'\,\gamma'$, and the result follows by symmetry. Now suppose that $\Gamma \gg A \doteq A'$ and $\Gamma \gg A' \doteq A''$ with the intent to show $\Gamma \gg A \doteq A''$. Suppose that $\gamma \doteq \gamma'' \in \Gamma$. By reflexivity and transitivity $\gamma \doteq \gamma \in \Gamma$, and so that by the first assumption $A\,\gamma \doteq A'\,\gamma$, and by the second assumption $A'\,\gamma \doteq A\,\gamma''$. The result follows by transitivity. Similar arguments are used to establish symmetry and transitivity for the hypothetical membership judgments. Respect for type equality follows immediately. □

2.2 Formal Framework

The syntax of *typed terms* is written in this color, and is stratified into two sorts, *types* and *terms*. The terms are usually decorated with types so as to ensure decidability of the formal judgments.

A *formal type theory* is an inductive definition of the following judgments:

1. *Types*: formation, $\Gamma \vdash A$, and definitional equivalence, $\Gamma \vdash A \equiv A'$.[2]
2. *Terms*: formation, $\Gamma \vdash M : A$, and definitional equivalence, $\Gamma \vdash M \equiv M' : A$.

The judgments of a formal type theory are called, oddly enough, *formal judgments* to distinguish them from their semantic counterparts. As the name suggests, definitional equivalence is intended to be an equivalence relation that is compatible with all type- and term formation constructs. The rules in Figure 2.2 ensure that type and term equivalence are reflexive, symmetric, and transitive, which are tacitly included in any given formal type theory. Each formalism, though, must ensure that definitional equivalence is compatible with the constructs specific to it.

Two ancillary judgments may be defined in terms of the primary judgments:

[1]Or even *hypothetico-general*!

[2]The term "definitional equivalence" is obtuse, but well-established; it is best to regard it as a term of art, rather than analyze its etymology. The terminology is comparable to the inexplicable use of the term "call by name" in programming languages.

1. *Contexts*: formation, Γ ctx and equivalence of contexts, $\Gamma \equiv \Gamma'$.
2. *Substitutions*: formation, $\Gamma' \vdash \gamma : \Gamma$, and equivalence, $\Gamma' \vdash \gamma \equiv \gamma' : \Gamma$.

These are defined by the rules in Figure 2.1, relative to the judgments defining a formal type theory. It is sometimes useful to let $\Gamma \vdash J$ stand for any of the formal judgments given above.

The action of a substitution on a type or term, called *instantiation*, is written in post-fix notation, $A\,\gamma$ or $M\,\gamma$, respectively. When considering the judgment $\Gamma \vdash J$ and a substitution $\Gamma' \vdash \gamma : \Gamma$, the notation $\Gamma' \vdash J\,\gamma$ stands for the formal judgment obtained by instantiating the components of J by γ. Thus, if $\Gamma \vdash J$ is $\Gamma \vdash M \equiv M' : A$, then $\Gamma' \vdash J\,\gamma$ stands for $\Gamma' \vdash M\,\gamma \equiv M'\,\gamma : A\,\gamma$.

Definition 2.2. *The* structural properties *of formal typing judgments are as follows:*

1. *(Reflexivity) If Γ ctx and $\Gamma = \Gamma_1\,, a : A\,\Gamma_2$, then $\Gamma \vdash a : A$.*
2. *(Substitution) If $\Gamma' \vdash \gamma : \Gamma$ and $\Gamma \vdash J$, then $\Gamma' \vdash J\,\gamma$.*
3. *(Invariance) Suppose that $\Gamma \vdash A \equiv A'$.*
   (a) If $\Gamma \vdash M : A$, then $\Gamma \vdash M : A'$.
   (b) If $\Gamma \vdash M \equiv M' : A$, then $\Gamma \vdash M \equiv M' : A'$.

   Moreover, if $\Gamma' \equiv \Gamma$ and $\Gamma \vdash J$, then $\Gamma' \vdash J$.
4. *(Unicity) If $\Gamma \vdash M : A$ and $\Gamma \vdash M : A'$, then $\Gamma \vdash A \equiv A'$. Moreover, if $\Gamma'' \vdash \gamma : \Gamma$ and $\Gamma'' \vdash \gamma : \Gamma'$, then $\Gamma \equiv \Gamma'$.*
5. *(Regularity)*
   (a) If $\Gamma \vdash A$, then Γ ctx.
   (b) If $\Gamma \vdash M : A$, then $\Gamma \vdash A$.
   (c) If $\Gamma \vdash A \equiv A'$, then $\Gamma \vdash A$ and $\Gamma \vdash A'$.
   (d) If $\Gamma \vdash M \equiv M' : A$, then $\Gamma \vdash M : A$ and $\Gamma \vdash M' : A$.

   Moreover, if $\Gamma' \vdash \gamma : \Gamma$, then Γ ctx and Γ' ctx.

The additional conditions on substitutions follow from the primary conditions with which they are associated. The substitution property implies the following two special cases of interest:

1. *(Weakening and Exchange)* If $\Gamma \vdash J$ and Γ' ctx and $\Gamma \subseteq \Gamma'$, then $\Gamma' \vdash J$.
2. *(Contraction)* If $\Gamma_1\,, a_1 : A\,, a_2 : A\,\Gamma_2 \vdash J$, then $\Gamma_1\,, a : A\,\Gamma_2\,\gamma \vdash J\,\gamma$, where γ is the substitution $id_{\Gamma_1\,\Gamma_2}[a_1 \mapsto a][a_2 \mapsto a]$.

In each case it is only necessary to choose γ appropriately, apply the substitution property, and then reason about the action of γ on types and terms.

It is normative for a formal type system to satisfy the structural properties given in Definition 2.2. The rules in Figure 2.3 ensure reflexivity and invariance. Substitution must be established

$$\textsc{CF-Emp}\quad \frac{}{\bullet\ \mathsf{ctx}} \qquad\qquad \textsc{CQ-Emp}\quad \frac{}{\bullet \equiv \bullet}$$

$$\textsc{CF-Ext}\quad \frac{\Gamma \vdash A \qquad (a \notin \Gamma)}{\Gamma\,,a : A\ \mathsf{ctx}} \qquad\qquad \textsc{CQ-Ext}\quad \frac{\Gamma \equiv \Gamma' \qquad \Gamma \vdash A \equiv A' \qquad (a \notin \Gamma, a \notin \Gamma')}{\Gamma\,,a : A \equiv \Gamma'\,,a : A'}$$

$$\textsc{SF-Emp}\quad \frac{\Gamma\ \mathsf{ctx}}{\Gamma \vdash \emptyset : \bullet} \qquad\qquad \textsc{SQ-Emp}\quad \frac{\Gamma\ \mathsf{ctx}}{\Gamma \vdash \emptyset \equiv \emptyset : \bullet}$$

$$\textsc{SF-Ext}\quad \frac{\Gamma' \vdash M : A \qquad \Gamma' \vdash \gamma : \Gamma \qquad (a \notin \Gamma)}{\Gamma' \vdash \gamma[a \mapsto M] : \Gamma\,,a : A} \qquad \textsc{SQ-Ext}\quad \frac{\Gamma' \vdash \gamma \equiv \gamma' : \Gamma \qquad \Gamma' \vdash M \equiv M' : A \qquad (a \notin \Gamma)}{\Gamma' \vdash \gamma[a \mapsto M] \equiv \gamma'[a \mapsto M'] : \Gamma\,,a : A}$$

Figure 2.1: Contexts and Substitutions

for each formalism, by analysis of the formation and equivalence rules for the constructs specific to it. Unicity and regularity require an analysis of the rules *in toto*, which are chosen so as to facilitate the argument.

The rules in Figures 2.1, 2.2, and 2.3 are collectively called the *framework rules*, which are to be included in any formal type theory.

It is also normative for a formalism to be *decidable* in that there should be an effective (ideally, but seldom, also efficient) procedure to determine whether or not $\Gamma \vdash A$ and $\Gamma \vdash M : A$. Achieving this often reduces to proving that it is decidable whether or not $\Gamma \vdash A \equiv A'$ and $\Gamma \vdash M \equiv M' : A$. Devising and verifying an algorithm to test definitional equivalence is often the biggest source of difficulty. It is often possible to employ one of several known decision methods, such as reducing both sides to a common normal form, or recursively simplifying at the root and comparing components structurally. The correctness proof often involves the use of Kripke-style logical relations (that is, pre-sheaf interpretations) to account for free variables.

2.3 Interpretation

The purpose of a formal type theory is to provide access to the truth as defined by a semantic type theory. Because formal typing judgments are inherently computably enumerable (generated by rules) there is no possibility for any particular formalism to be definitive in that it captures all and only the truths for a particularly system of types.[3] Thus, formalisms are chosen based on pragmatic considerations, such as convenience of use in a particular implementation, and may be

[3]This is the essence of Gödel's Theorem, which can only be evaded for very simplistic, inexpressive type theories that are not even capable of expressing the natural numbers.

$$\textsc{TQ-R}\quad \frac{\Gamma \vdash A}{\Gamma \vdash A \equiv A} \qquad\qquad \textsc{MQ-R}\quad \frac{\Gamma \vdash M : A}{\Gamma \vdash M \equiv M : A}$$

$$\textsc{TQ-S}\quad \frac{\Gamma \vdash A \equiv A'}{\Gamma \vdash A' \equiv A} \qquad\qquad \textsc{MQ-S}\quad \frac{\Gamma \vdash M \equiv M' : A}{\Gamma \vdash M' \equiv M : A}$$

$$\textsc{TQ-T}\quad \frac{\Gamma \vdash A \equiv A' \qquad \Gamma \vdash A' \equiv A''}{\Gamma \vdash A \equiv A''} \qquad\qquad \textsc{MQ-T}\quad \frac{\Gamma \vdash M \equiv M' : A \qquad \Gamma \vdash M' \equiv M'' : A}{\Gamma \vdash M \equiv M'' : A}$$

Figure 2.2: Definitional Equivalence

$$\textsc{VAR}\quad \frac{\Gamma = \Gamma_1 \, , a : A \, \Gamma_2 \qquad \Gamma\ \mathsf{ctx}}{\Gamma \vdash a : A} \qquad\qquad \textsc{M-INV}\quad \frac{\Gamma \vdash M : A \qquad \Gamma \vdash A \equiv A'}{\Gamma \vdash M : A'}$$

$$\textsc{Q-INV}\quad \frac{\Gamma \vdash M \equiv M' : A \qquad \Gamma \vdash A \equiv A'}{\Gamma \vdash M \equiv M' : A'}$$

Figure 2.3: Structural Rules of a Formal Type System

varied at will in pursuit of these goals. However, any formalism should at least be sound in the sense of deriving only semantically valid judgments.

In general the syntax of a formalism is more elaborate than required for its semantics, often carrying typing information that facilitates the derivation of a type checking or definitional equivalence algorithm. If this information is not required for defining the dynamics of the language, it must be "erased" on passage to the semantic realm. Often the erasure is self-evident, and is not explicitly defined, but for the sake of accuracy it is important to account for it in the statement of soundness for a formal type theory. Thus, the *erasure* of a type expression, A, or term, M, is the untyped expressions $|A|$ or $|M|$, respectively. Erasure of variables is always defined by $|a| = a$, relying on a bijection between variables a and a indicated by the recoloring. Moreover, erasure for any particular formalism must be defined *compositionally* in that it commutes with substitution, $|A[M/a]| = |A|[|M|/a]$ and $|M[N/a]| = |M|[|N|/a]$.

Cohere problems arise if erasure depends on derivations rather than syntax.

Definition 2.3. *A formal type theory is* sound *with respect to a semantic type theory, via a given notion of erasure, iff the following properties hold:*

1. *If* $\Gamma \vdash A$, *then* $|\Gamma| \gg |A|$ type.
2. *If* $\Gamma \vdash A \equiv A'$, *then* $|\Gamma| \gg |A| \doteq |A'|$.
3. *If* $\Gamma \vdash M : A$, *then* $|\Gamma| \gg |M| \in |A|$.
4. *If* $\Gamma \vdash M \equiv M' : A$, *then* $|\Gamma| \gg |M| \doteq |M'| \in |A|$.

Viewing the semantic type theory as specifying program behavior, the soundness property states that formally well-formed types and terms of a type behave as types and elements of the given type should, and that definitionally equivalent types and terms give rise to equal behaviors. The *fundamental theorem* for a given formal and semantic type system states that a given notion of erasure is sound.

Chapter 3

The Negative Fragment

The trio of unit, product, and function types are a natural starting point for the development of the computational perspective on type theory. These types are described as *negative* in the sense that they may be characterized by the principles for using their elements to construct elements of other types. The next chapter considers the *positive* void and sum types that may be characterized by the principles for building elements of these types from elements of other types.

3.1 Semantics

Definition 3.1 (Untyped Syntax)**.** *The syntax of untyped expressions for the negative fragment is given by the following grammar:*

$$E ::= a \mid 1 \mid \langle\,\rangle \mid E_1 \times E_2 \mid \langle E_1\,,E_2\rangle \mid E\cdot 1 \mid E\cdot 2 \mid E_1 \to E_2 \mid \lambda\,(a.E) \mid \mathsf{ap}\,(E_1\,,E_2)$$

There is no syntactic distinction between types and elements, but often the meta-variables A, B, and C are used for expressions that are to be types, and M, N, O, P, Q, and R for expressions that are to be elements.

Definition 3.2 (Dynamics)**.** *The assertions $E\ \mathsf{val}$ and $E \longmapsto E'$ are inductively defined for closed expressions E and E' by the rules given in Figure 3.1.*

Lemma 3.1 (Determinacy)**.** *For any closed E, E_1, and E_2, if $E \longmapsto E_1$ and $E \longmapsto E_2$, then E_1 is E_2.*

Proposition 3.2 (Negative Type Constructors)**.** *There exists a semantic type system for which the following conditions are valid.*

1. *The judgment $A \doteq A'$ holds if $A \Downarrow A_0$, $A' \Downarrow A_0'$, and either*
   (a) *$A_0 = 1$ and $A_0' = 1$, or*
   (b) *$A_0 = A_1 \times A_2$ and $A_0' = A_1' \times A_2'$ and $A_1 \doteq A_1'$ and $A_2 \doteq A_2'$, or*

$$\frac{}{\mathbf{1}\ \mathsf{val}} \qquad \frac{}{\langle\,\rangle\ \mathsf{val}}$$

$$\frac{}{E_1 \times E_2\ \mathsf{val}} \qquad \frac{}{\langle E_1\,, E_2\rangle\ \mathsf{val}}$$

$$\frac{E \longmapsto E'}{E \cdot 1 \longmapsto E' \cdot 1} \qquad \frac{E \longmapsto E'}{E \cdot 2 \longmapsto E' \cdot 2}$$

$$\frac{}{\langle E_1\,, E_2\rangle \cdot 1 \longmapsto E_1} \qquad \frac{}{\langle E_1\,, E_2\rangle \cdot 2 \longmapsto E_2}$$

$$\frac{}{E_1 \to E_2\ \mathsf{val}} \qquad \frac{}{\lambda\,(a.E)\ \mathsf{val}}$$

$$\frac{E_1 \longmapsto E_1'}{\mathsf{ap}\,(E_1\,, E_2) \longmapsto \mathsf{ap}\,(E_1'\,, E_2)} \qquad \frac{}{\mathsf{ap}\,(\lambda\,(a.E)\,, E_2) \longmapsto E\,[E_2/a]}$$

Figure 3.1: Dynamics of Negative Types

   (c) $A_0 = A_1 \to A_2$ *and* $A'_0 = A'_1 \to A'_2$ *and* $A_1 \doteq A'_1$, *and* $A_2 \doteq A'_2$.

2. *Given that* A type, *the judgment* $M \doteq M' \in A$ *holds if* $A \Downarrow A_0$, $M \Downarrow M_0$, *and* $M' \Downarrow M'_0$, *and either*

   (a) $A_0 = 1$ *and* $M_0 = \langle\,\rangle = M'_0$, *or*

   (b) $A_0 = A_1 \times A_2$, $M \Downarrow \langle M_1 , M_2 \rangle$, $M' \Downarrow \langle M'_1 , M'_2 \rangle$, $M_1 \doteq M'_1 \in A_1$, *and* $M_2 \doteq M'_2 \in A_2$, *or*

   (c) $A_0 = A_1 \to A_2$, $M \Downarrow \lambda\,(a.M_2)$, $M' \Downarrow \lambda\,(a.M'_2)$, *and, for all closed* M_1 *and* M'_1, *if* $M_1 \doteq M'_1 \in A_1$, *then* $M_2\,[M_1/a] \doteq M'_2\,[M'_1/a] \in A_2$.

The negative connectives may be characterized by their behavior under elimination.

Corollary 3.3. *For the semantic type system given by Proposition 3.2,*

1. $M \doteq M' \in A_1 \times A_2$ *iff* $M \cdot 1 \doteq M' \cdot 1 \in A_1$ *and* $M \cdot 2 \doteq M' \cdot 2 \in A_2$

2. $M \doteq M' \in A_1 \to A_2$ *iff* $a : A_1 \gg \mathsf{ap}\,(M\,,a) \doteq \mathsf{ap}\,(M'\,,a) \in A_2$.

Proof. 1. For the forward direction if $M \doteq M' \in A_1 \times A_2$, then $M \Downarrow \langle M_1 , M_2 \rangle$ and $M' \Downarrow \langle M'_1 , M'_2 \rangle$ with $M_1 \doteq M'_1 \in A_1$ and $M_2 \doteq M'_2 \in A_2$. But then $M \cdot 1 \longmapsto^* M_1$ and $M' \cdot 1 \longmapsto^* M'_1$, and similarly for the right projection. The result follows by closure under reverse evaluation. For the reverse direction, the assumptions ensure that $M \Downarrow \langle M_1 , M_2 \rangle$, $M' \Downarrow \langle M'_1 , M'_2 \rangle$, $M_1 \doteq M'_1 \in A_1$, and $M_2 \doteq M'_2 \in A_2$, as required.

2. In the forward direction use converse transition, and in the reverse the definition of evaluation and equality in a type.

□

If, for example, application defined for "foreign functions" that are not given by λ-abstractions, then the characterization given by Corollary 3.3 would not be valid.

Lemma 3.4 (Formation Principles).

1. $\Gamma \gg 1 \doteq 1$.

2. *If* $\Gamma \gg A_1 \doteq A'_1$ *and* $\Gamma \gg A_2 \doteq A'_2$, *then* $\Gamma \gg A_1 \times A_2 \doteq A'_1 \times A'_2$.

Lemma 3.5 (Introduction Principles). 1. *If* $\Gamma \gg M_1 \doteq M'_1 \in A_1$ *and* $\Gamma \gg M_2 \doteq M'_2 \in A_2$, *then* $\Gamma \gg \langle M_1 , M_2 \rangle \doteq \langle M'_1 , M'_2 \rangle \in A_1 \times A_2$.

2. *If* $\Gamma\,, a : A_1 \gg M_2 \doteq M'_2 \in A_2$, *then* $\Gamma \gg \lambda\,(a.M_2) \doteq \lambda\,(a.M'_2) \in A_1 \to A_2$.

Proof. Follow directly from the definitions of the hypothetical judgment and the type constructors, together with closure of equality under reverse evaluation (Theorem 2.2). Consider the second of the two principles. Suppose that $\gamma \doteq \gamma' \in \Gamma$, with the goal to show that $\lambda\,(a.M_2\,\gamma) \doteq \lambda\,(a.M'_2\,\gamma') \in A_1\,\gamma \to A_2\,\gamma$. Assuming that $M_1 \doteq M'_1 \in A_1\,\gamma$, by closure under reverse evaluation it suffices to prove that $M_2\,\gamma\,[M_1/a] \doteq M_2\,\gamma'\,[M'_1/a] \in A_2\,\gamma$. Let $\delta \triangleq \gamma[a \mapsto M_1]$, $\delta' \triangleq \gamma'[a \mapsto M'_1]$, and note that $\delta \doteq \delta' \in \Gamma\,, a : A_1\,\gamma$. The result follows by inductive hypothesis and noting that $M_2\,\delta = M_2\,\gamma\,[M_1/a]$ and $M'_2\,\delta' = M'_2\,\gamma'\,[M'_1/a]$. □

Lemma 3.6 (Elimination Principles). *1. If $\Gamma \gg M \doteq M' \in A_1 \times A_2$, then $\Gamma \gg M \cdot 1 \doteq M' \cdot 1 \in A_1$ and $\Gamma \gg M \cdot 2 \doteq M' \cdot 2 \in A_2$.*

2. If $\Gamma \gg M \doteq M' \in A_1 \to A_2$ and $\Gamma \gg M_1 \doteq M_1' \in A_1$, then $\Gamma \gg \mathsf{ap}(M, M_1) \doteq \mathsf{ap}(M', M_1') \in A_2$.

Proof. Both follow directly from Theorem 3.3 and the definition of the hypothetical judgment. □

Lemma 3.7 (Computation Principles).

1. *If $\Gamma \gg A_1$ type, $\Gamma \gg A_2$ type, $\Gamma, a_1 : A_1 \gg M_2 \in A_2$ and $\Gamma \gg M_1 \in A_1$, then*

$$\Gamma \gg \mathsf{ap}(\lambda(a_1.M_2), M_1) \doteq M_2[M_1/a_1] \in A_2.$$

2. *If $\Gamma \gg M_1 \in A_1$ and $\Gamma \gg M_2 \in A_2$, then*

$$\Gamma \gg \langle M_1, M_2\rangle \cdot 1 \doteq M_1 \in A_1 \quad \text{and} \quad \Gamma \gg \langle M_1, M_2\rangle \cdot 2 \doteq M_2 \in A_2.$$

Lemma 3.8 (Unicity Principles).

1. *If $\Gamma \gg M \in A_1 \to A_2$, then $\Gamma \gg M \doteq \lambda(a_1.\mathsf{ap}(M, a_1)) \in A_1 \to A_2$.*
2. *If $\Gamma \gg M \in A_1 \times A_2$, then $\Gamma \gg M \doteq \langle M \cdot 1, M \cdot 2\rangle \in A_1 \times A_2$.*

In what sense is Proposition 3.2 true? From one point of view it is a *scientific law*, certain "facts of nature" about programs that are empirically justified by mathematical experience rooted in the mental capacities of human beings. Although it is common to regard mathematics as transcending humanity, the core contention of constructivism is that it is not necessary to do so, it being quite sufficient to rely on the intuitive (evolutionarily inherent) concept of computation. It is possible to push the locus of such assumptions elsewhere by "explaining" type theory in other terms. But then those other terms need explanation, and there is no end to the regress. Some things must be accepted as basic, and it may as well be computation.

Nevertheless, it is useful to explain why Proposition 3.2 is true in terms of a theory of relations. There are two concerns. First, the stated conditions are apparently circular in that type equality and element equality are characterized in terms of themselves. Perhaps they are *viciously* circular in that no semantic type system satisfies them? Second, the conditions intermix evaluation with characterizations of values as types and elements. Is this properly defined?

A *candidate type system* τ is a three-place relation on two closed values and a binary relation, ϕ, on closed values. The assertion $\tau(A_0, A_1, \phi)$ specifes that A_0 and A_1 are equal types with common element equality given by ϕ. A candidate type system, τ, is extended to closed expressions by defining $\widehat{\tau}(A, A', \phi)$ iff $A \Downarrow A_0$, $A' \Downarrow A_0'$, and $\tau(A_0, A_0', \phi)$. An element equality relation, ϕ, is similarly extended by defining $\widehat{\phi}(M, M')$ iff $M \Downarrow M_0$, $M' \Downarrow M_0$, and $\phi(M_0, M_0')$.

A candidate type system τ is a *proper type system* if it satisfies these additional conditions:

1. τ is symmetric and transitive (viewed as a binary relation on its first two arguments).
2. If $\tau(A_0, A_0', \phi)$, then ϕ is symmetric and transitive.

3. If $\tau(A_0, A_0', \phi)$ and $\tau(A_0, A_0', \phi')$, then $\phi = \phi'$.

The last condition is called *functionality*, because it expresses that ϕ is determined by the equal types A and A'.

The meaning of the equality judgments relative to a proper type system τ is defined as follows:

1. $\tau \models A \doteq A'$ means that there exists ϕ such that $\widehat{\tau}(A, A', \phi)$.
2. Assuming $\tau \models A\,\mathsf{type}$, $\tau \models M \doteq M' \in A$ means that $\widehat{\tau}(A, A, \phi)$ and $\widehat{\phi}(M, M')$ for some ϕ.

The meaning of the judgments $A\,\mathsf{type}$ and $M \in A$, relative to τ, are defined as the reflexive instances of the corresponding equality judgments. Context and substitution equality relative to τ is defined variable-by-variable. The meanings of the hypothetical judgments relative to a proper type system τ are given by these conditions:

1. $\tau \models \Gamma \gg A \doteq A'$ iff for all γ and γ', if $\tau \models \gamma \doteq \gamma' \in \Gamma$, then $\tau \models A\,\gamma \doteq A'\,\gamma'$.
2. $\tau \models \Gamma \gg M \doteq M' \in A$ iff for all γ and γ', if $\tau \models \gamma \doteq \gamma' \in \Gamma$, then $\tau \models M\,\gamma \doteq M'\,\gamma' \in A\,\gamma$.

The properties expressed by Proposition 3.2 relativize to a proper type system τ as follows:

1. $\tau \models 1 \doteq 1$, and $\tau \models \langle\,\rangle \doteq \langle\,\rangle \in 1$.
2. $\tau \models A_1 \times A_2 \doteq A_1' \times A_2'$ iff $\tau \models A_1 \doteq A_1'$ and $\tau \models A_2 \doteq A_2'$; and $\tau \models \langle M_1, M_2\rangle \doteq \langle M_1', M_2'\rangle \in A_1 \times A_2$ iff $\tau \models M_1 \doteq M_1' \in A_1$ and $\tau \models M_2 \doteq M_2' \in A_2$.
3. $\tau \models A_1 \to A_2 \doteq A_1' \to A_2'$ iff $\tau \models A_1 \doteq A_1'$ and $\tau \models A_2 \doteq A_2'$; and $\tau \models \lambda\,(a.M_2) \doteq \lambda\,(a.M_2') \in A_1 \to A_2$ iff $\tau \models M_2\,[M_1/a] \doteq M_2'\,[M_1'/a] \in A_2$ whenever $\tau \models M_1 \doteq M_1' \in A_1$.

These conditions define when a type system, τ, defines the negative connectives. But why is there any such type system? As mentioned, one answer is to simply posit that there is one, which amounts to considering the stated conditions as a valid definition of τ. The apparent circularity of the conditions can be justified by noting that the self-references are always with "smaller" types than the one under consideration. For example, the third condition for type equality relative to τ appeals to type equality relative to τ, but for "smaller" expressions A_1 and A_2 than $A_1 \to A_2$.

Another answer is to appeal to a theory of self-referential relations that justifies the existence of such a type system.[1] The main tool here is called *Tarski's fixed point theorem*, which ensures that any monotone (order-preserving) function on a complete lattice has a least fixed point. (See Appendix B for a brief review of Tarski's theorem.) Now candidate type systems ordered by containment form a complete lattice, so a proper type system may be defined by giving a monotone operator on this lattice, and showing that its least fixed point is a proper type system.

For the case at hand, the operator $T(\tau)$ is defined on candidate type systems as follows.

Definition 3.3. *The operator T on candidate type systems is defined by the equation*

$$T(\tau) = \textsc{One} \cup \textsc{Prod}(\tau) \cup \textsc{Fun}(\tau),$$

[1]Of course, it would be necessary to justify that theory of relations, but never mind.

where

$$\begin{aligned}
\textsc{One} &= \{\, (1, 1, \mathbf{1}) \,\} \\
\textsc{Prod}(\tau) &= \{\, (A_1 \times A_2, A_1' \times A_2', \phi_1 \times \phi_2) \mid \tau(A_1, A_1', \phi_1) \wedge \tau(A_2, A_2', \phi_2) \,\} \\
\textsc{Fun}(\tau) &= \{\, (A_1 \to A_2, A_1' \to A_2', \phi_1 \to \phi_2) \mid \tau(A_1, A_1', \phi_1) \wedge \tau(A_2, A_2', \phi_2) \,\}
\end{aligned}$$

with

$$\begin{aligned}
\mathbf{1} &= \{\, (\langle\rangle, \langle\rangle) \,\} \\
\phi_1 \times \phi_2 &= \{\, (\langle M_1, M_2\rangle, \langle M_1', M_2'\rangle) \mid \phi_1(M_1, M_1') \wedge \phi_2(M_2, M_2') \,\} \\
\phi_1 \to \phi_2 &= \{\, (\lambda\,(a.M_2), \lambda\,(a.M_2')) \mid \phi_1(M_1, M_1') \supset \phi_2(M_2\,[M_1/a], M_2'\,[M_1'/a]) \,\}
\end{aligned}$$

The operator T given by Definition 3.3 is clearly monotone with respect to containment, and thus has a least fixed point, $\tau_0 = T(\tau_0)$, given by Tarski's Theorem. It is obvious that τ_0 defines the negative connectives to have the properties described above. It remains only to check that τ_0 is a proper type system.

Theorem 3.9. *The candidate type system τ_0 defined as the least fixed point of the operator T is a proper type system.*

Proof. For example, to show that τ_0 is functional, define the candidate type system τ so that $\tau(A, A', \phi)$ iff for all ϕ', if $\tau_0(A, A', \phi')$, then $\phi = \phi'$. If $T(\tau)) \subseteq \tau$, then $\tau_0 \subseteq \tau$, which expresses functionality for τ_0. It is sufficent to prove these three facts about τ:

1. $\textsc{One} \subseteq \tau$;

2. $\textsc{Prod}(\tau) \subseteq \tau$; and

3. $\textsc{Fun}(\tau) \subseteq \tau$.

Addressing each in turn:

1. If $\tau_0(1, 1, \phi')$, then $\phi' = \mathbf{1}$, by construction of τ_0.

2. Suppose that $\textsc{Prod}(\tau)(A, A', \phi)$. Then $A = A_1 \times A_2$, $A' = A_1' \times A_2'$, and $\phi = \phi_1 \times \phi_2$, where $\tau(A_1, A_1', \phi_1)$ and $\tau(A_2 A_2', \phi_2)$. Now, if $\tau_0(A, A', \phi')$, then $\phi' = \phi_1' \times \phi_2'$, where $\tau_0(A_1, A_1', \phi_1')$ and $\tau_0(A_2, A_2', \phi_2')$. But then $\phi_1' = \phi_1$ and $\phi_2' = \phi_2$, and so $\phi' = \phi$.

3. Analogous to the previous case.

Similar arguments may be used to prove that τ_0 is symmetric and transitive, as is each type element equality relation given by it.

□

The use of Tarski's theorem is overkill for the purposes at hand, but is illustrative of a general method for justifying the existence of much more complicated type systems by appeal to a theory of relations.

3.2 Formalisms

The formal syntax for the negative fragment is given as follows.

Definition 3.4 (Formal Syntax)**.**

$$\begin{aligned} A &::= 1 \mid A_1 \times A_2 \mid A_1 \to A_2 \\ M &::= a \mid \langle\,\rangle \mid \langle M_1 \,, M_2 \rangle \mid M \cdot 1 \mid M \cdot 2 \mid \lambda[A](a.M) \mid \mathsf{ap}(M_1 \,, M_2) \end{aligned}$$

It is immediate from the syntax that $A\,[M/a]$ is A for the negative fragment, simply because the variable a cannot occur within it.

A formalism for the negative connectives is given by the rules in Figure 3.2, tacitly augmented by the framework rules given in Chapter 2.

Theorem 3.10. 1. *(Independence) If $\Gamma \vdash A$, then $\bullet \vdash A$, and if $\Gamma \vdash A \equiv A'$, then $\bullet \vdash A \equiv A'$.*

2. *(Degeneracy) If $\bullet \vdash A \equiv A'$, then A is A'.*

3. *(Unicity) If $\Gamma \vdash M : A$ and $\Gamma \vdash M : A'$, then $\Gamma \vdash A \equiv A'$.*

4. *(Regularity) If $\Gamma \vdash M : A$, then $\Gamma \vdash A$; if $\Gamma \vdash A$, then Γ ctx; if $\Gamma \vdash M \equiv M' : A$, then $\Gamma \vdash M : A$ and $\Gamma \vdash M' : A$; if $\Gamma \vdash A \equiv A'$, then $\Gamma \vdash A$ and $\Gamma \vdash A'$.*

Type formation extends to contexts variable-by-variable, so that $\Gamma' \vdash \bullet$, and if $\Gamma' \vdash A$ and $\Gamma' \vdash \Gamma$ then $\Gamma' \vdash \Gamma \,, a : A$. Substitutions are elements of contexts, defined variable-by-variable by these conditions: $\Gamma' \vdash \emptyset : \bullet$, and if $\Gamma' \vdash \gamma : \Gamma$ and $\Gamma' \vdash M : A\,\gamma$, then $\Gamma' \vdash \gamma[a \mapsto M] : \Gamma \,, a : A$.

Theorem 3.11 (Substitution)**.** *The following rule is admissible for the formalism defined by the rules in Figure 3.2 for each basic judgment form, J:*

$$\frac{\Gamma \vdash J \qquad \Gamma' \vdash \gamma : \Gamma}{\Gamma' \vdash J\,\gamma}$$

Proof. The proof proceeds by induction on the derivation of the first premise according to the rules given in Figure 3.2. For example, consider rule VAR. The context Γ is of the form $\Gamma_1 \,, a : A\,\Gamma_2$, and it is required to show $\Gamma' \vdash M : A$. But then $\gamma(a) = M$ where $\Gamma' \vdash M : A$, because of the premise $\Gamma' \vdash \gamma : \Gamma$ on the rule. The other cases are handled inductively. □

TF-1
$$\frac{\Gamma\ \mathsf{ctx}}{\Gamma \vdash 1}$$

TQ-1
$$\frac{\Gamma\ \mathsf{ctx}}{\Gamma \vdash 1 \equiv 1}$$

MF-1-I
$$\frac{\Gamma\ \mathsf{ctx}}{\Gamma \vdash \langle\,\rangle : 1}$$

MQ-1-I
$$\frac{\Gamma\ \mathsf{ctx}}{\Gamma \vdash \langle\,\rangle \equiv \langle\,\rangle : 1}$$

TF-×
$$\frac{\Gamma \vdash A_1 \qquad \Gamma \vdash A_2}{\Gamma \vdash A_1 \times A_2}$$

TQ-×
$$\frac{\Gamma \vdash A_1 \equiv A_1' \qquad \Gamma \vdash A_2 \equiv A_2'}{\Gamma \vdash A_1 \times A_2 \equiv A_1' \times A_2'}$$

MF-×-I
$$\frac{\Gamma \vdash M_1 : A_1 \qquad \Gamma \vdash M_2 : A_2}{\Gamma \vdash \langle M_1 \,, M_2 \rangle : A_1 \times A_2}$$

MQ-×-I
$$\frac{\Gamma \vdash M_1 \equiv M_1' : A_1 \qquad \Gamma \vdash M_2 \equiv M_2' : A_2}{\Gamma \vdash \langle M_1 \,, M_2 \rangle \equiv \langle M_1' \,, M_2' \rangle : A_1 \times A_2}$$

MF-×-E-L
$$\frac{\Gamma \vdash M : A_1 \times A_2}{\Gamma \vdash M \cdot 1 : A_1}$$

MQ-×-E-L
$$\frac{\Gamma \vdash M \equiv M' : A_1 \times A_2}{\Gamma \vdash M \cdot 1 \equiv M' \cdot 1 : A_1}$$

MF-×-E-R
$$\frac{\Gamma \vdash M : A_1 \times A_2}{\Gamma \vdash M \cdot 2 : A_2}$$

MQ-×-E-R
$$\frac{\Gamma \vdash M \equiv M' : A_1 \times A_2}{\Gamma \vdash M \cdot 2 \equiv M' \cdot 2 : A_2}$$

MQ-×-C-L
$$\frac{\Gamma \vdash M_1 : A_1 \qquad \Gamma \vdash M_2 : A_2}{\Gamma \vdash \langle M_1 \,, M_2 \rangle \cdot 1 \equiv M_1 : A_1}$$

MQ-×-C-R
$$\frac{\Gamma \vdash M_1 : A_1 \qquad \Gamma \vdash M_2 : A_2}{\Gamma \vdash \langle M_1 \,, M_2 \rangle \cdot 2 \equiv M_2 : A_2}$$

TF-→
$$\frac{\Gamma \vdash A_1 \qquad \Gamma \vdash A_2}{\Gamma \vdash A_1 \to A_2}$$

TQ-→
$$\frac{\Gamma \vdash A_1 \equiv A_1' \qquad \Gamma \vdash A_2 \equiv A_2'}{\Gamma \vdash A_1 \to A_2 \equiv A_1' \to A_2'}$$

MF-→-I
$$\frac{\Gamma \,, a : A_1 \vdash M_2 : A_2}{\Gamma \vdash \lambda[A_1](a.M_2) : A_1 \to A_2}$$

MQ-→-I
$$\frac{\Gamma \vdash A_1 \equiv A_1' \qquad \Gamma \,, a : A_1 \vdash M_2 \equiv M_2' : A_2}{\Gamma \vdash \lambda[A_1](a.M_2) \equiv \lambda[A_1'](a.M_2') : A_1 \to A_2}$$

MF-→-E
$$\frac{\Gamma \vdash M_1 : A_2 \to A \qquad \Gamma \vdash M_2 : A_2}{\Gamma \vdash \mathsf{ap}\,(M_1 \,, M_2) : A}$$

MQ-→-E
$$\frac{\Gamma \vdash M_1 \equiv M_1' : A_2 \to A \qquad \Gamma \vdash M_2 \equiv M_2' : A_2}{\Gamma \vdash \mathsf{ap}\,(M_1 \,, M_2) \equiv \mathsf{ap}\,(M_1' \,, M_2') : A}$$

MQ-→-C
$$\frac{\Gamma \,, a : A_1 \vdash M_2 : A_2 \qquad \Gamma \vdash M_1 : A_1}{\Gamma \vdash \mathsf{ap}\,(\lambda[A_1](a.M_2) \,, M_1) \equiv M_2\,[M_1/a] : A_2}$$

Figure 3.2: Formal Type System for Negative Types

3.3 Interpretation

Definition 3.5. *The* erasure *of derivations is defined by the following equations:*

$$\begin{aligned} |1| &= 1 \\ |A_1 \times A_2| &= |A_1| \times |A_2| \\ |A_1 \to A_2| &= |A_1| \to |A_2| \end{aligned}$$

$$\begin{aligned} |\langle\,\rangle| &= \langle\,\rangle \\ |\langle M\,, N\rangle| &= \langle |M|\,, |N|\rangle \\ |M \cdot 1| &= |M| \cdot 1 \\ |M \cdot 2| &= |M| \cdot 2 \\ |\lambda[A](a.M)| &= \lambda\,(a.|M|) \\ |\mathsf{ap}\,(M\,, N)| &= \mathsf{ap}\,(|M|\,, |N|) \end{aligned}$$

Thus erasure merely removes the domain type of the λ-abstraction.

Theorem 3.12.
1. *If Γ ctx, then $|\Gamma|$ ctx.*
2. *If $\Gamma \vdash A$, then $|\Gamma| \gg |A|$ type.*
3. *If $\Gamma \vdash A \equiv A'$, then $|\Gamma| \gg |A| \doteq |A'|$.*
4. *If $\Gamma \vdash M : A$, then $|\Gamma| \gg |M| \in |A|$.*
5. *If $\Gamma \vdash M \equiv M' : A$, then $|\Gamma| \gg |M| \doteq |M'| \in |A|$.*

Proof. Simultaneously, by induction on the derivations of the formal judgments in accordance with the rules given in Figure 3.2. Two main facts are central to the proof. First, the semantics of the hypothetical judgment is defined so as to preserve exact equality. Second, the element equality relations are closed under head expansion, which means that if $M' \longmapsto M$ and $M \doteq N \in A$, then $M' \doteq N \in A$ (and symmetrically). □

The formalism given in Figure 3.2 may be augmented with additional rules of definitional equivalence that express the uniqueness properties of the elimination forms for each type. These additional rules are given in Figure 3.3. They are compatible with Theorem 3.12, but complicate the proof of decidability of type checking.

The uniqueness principles are sometimes called *extensionality principles*. In particular the following rule of definitional equivalence of functions is derivable in their presence:

$$\text{MQ-}\to\text{-EXT}\quad \frac{\Gamma\,, a : A_2 \vdash \mathsf{ap}\,(M\,, a) \equiv \mathsf{ap}\,(M'\,, a) : A}{\Gamma \vdash M \equiv M' : A_2 \to A}$$

For example, $\Gamma \vdash \lambda[A](a.a) \equiv \lambda[A](a.\langle\,\rangle) : A \to 1$ is derivable. By Theorem 3.3 the corresponding principle of exact equality holds, and so Theorem 3.12 extends to account for extensionality.

$$\textsc{mq-1-u}\quad \frac{\Gamma \vdash M : 1}{\Gamma \vdash M \equiv \langle\,\rangle : 1}$$

$$\textsc{mq-}\times\textsc{-u}\quad \frac{\Gamma \vdash M : A_1 \times A_2}{\Gamma \vdash M \equiv \langle M \cdot 1 , M \cdot 2 \rangle : A_1 \times A_2}$$

$$\textsc{mq-}\rightarrow\textsc{-u}\quad \frac{\Gamma \vdash M : A_1 \rightarrow A_2}{\Gamma \vdash M \equiv \lambda[A_1](a.\mathsf{ap}\,(M\,,a)) : A_1 \rightarrow A_2}$$

Figure 3.3: Unicity Principles for Negative Types

3.4 Exercises

1. Explore algorithms for deciding definitional equivalence. Two classes: reduction-based and comparison-based.

Chapter 4

The Positive Fragment

This chapter extends the negative fragment defined in chapter 3 with finite sums. The elements of a sum are elements of one of the summands, marked as such, with the same equality. Consequently, the elimination form branches on the summand, providing an action on the underlying type; when there are no branches, it vacuously inhabits its target type, there being no possible argument for it. Finite sums are a special case of inductive types, which permit not only heterogeneity, but also self-reference, allowing for example the definition of the type of natural numbers.

4.1 Semantics

The untyped syntax of the positive fragment comprises injection into finite sums and case analysis of the summand of an injected element. Finite sums generalize finite types given by their elements, and specialize the concept of an inductive type by disallowing self-reference.

Definition 4.1 (Untyped Syntax)**.** *The untyped syntax of the expressions of the positive fragment is given by the following grammar:*

$$E ::= 0 \mid \mathsf{case}\,\{\,\}\,(E) \mid E_1 + E_2 \mid \mathit{1} \cdot E \mid \mathit{2} \cdot E \mid \mathsf{case}\,\{a_1.E_1 \mid a_2.E_2\}\,(E)$$

Whereas the binary case analysis is familiar in other languages, the nullary case analysis is less so. It is, nevertheless, a means to branch on the summand of its argument, despite that no closed argument can be given.

The dynamics of these constructs is inductively defined by these rules.

Definition 4.2 (Dynamics).

$$\frac{}{0\ \mathsf{val}} \qquad \frac{M \longmapsto M'}{\mathsf{case}\,\{\,\}\,(M) \longmapsto \mathsf{case}\,\{\,\}\,(M')}$$

$$\frac{}{A_1 + A_2\ \mathsf{val}} \qquad \frac{}{\mathit{1} \cdot M\ \mathsf{val}} \qquad \frac{}{\mathit{2} \cdot M\ \mathsf{val}}$$

$$\frac{M \longmapsto M'}{\mathsf{case}\,\{a_1.M_1 \mid a_2.M_2\}\,(M) \longmapsto \mathsf{case}\,\{a_1.M_1 \mid a_2.M_2\}\,(M')}$$

$$\frac{}{\mathsf{case}\,\{a_1.M_1 \mid a_2.M_2\}\,(\mathit{1} \cdot M) \longmapsto M_1\,[M/a_1]}$$

$$\frac{}{\mathsf{case}\,\{a_1.M_1 \mid a_2.M_2\}\,(\mathit{2} \cdot M) \longmapsto M_2\,[M/a_2]}$$

Proposition 4.1 (Positive Type Constructors). *There exists a semantic type system satisfying the following conditions:*

1. *The judgment $A \doteq A'$ holds if $A \Downarrow A_0$, $A' \Downarrow A_0'$, and either*
   (a) *A_0 is 0 and A_0' is 0, or*
   (b) *A_0 is $A_1 + A_2$, A_0' is $A_1' + A_2'$.*
2. *Given that A type, the judgment $M \doteq M' \in A$ holds if $A \Downarrow A_0$ and either*
   (a) *A_0 is 0 and false, or*
   (b) *A_0 is $A_1 + A_2$ and either*
      i. *$M \Downarrow \mathit{1} \cdot M_1$, $M' \Downarrow \mathit{1} \cdot M_1'$, and $M_1 \doteq M_1' \in A_1$, or*
      ii. *$M \Downarrow \mathit{2} \cdot M_2$, $M' \Downarrow \mathit{2} \cdot M_2'$, and $M_2 \doteq M_2' \in A_2$.*

Thus, the type 0 is empty in that it has no closed values as elements. As a result, other types, such as $1 \to 0$, are also uninhabited.

Lemma 4.2 (Formation Principles).

1. $\Gamma \gg 0 \doteq 0$.
2. *If* $\Gamma \gg A_1 \doteq A_1'$ *and* $\Gamma \gg A_2 \doteq A_2'$, *then* $\Gamma \gg A_1 + A_2 \doteq A_1' + A_2'$.

Lemma 4.3 (Introduction Principles).

1. $\Gamma, a_1 : A_1 \gg \mathit{1} \cdot a_1 \in A_1 + A_2$.
2. $\Gamma, a_2 : A_2 \gg \mathit{1} \cdot a_2 \in A_1 + A_2$.

Proof. These follow immediately from the definition of the hypothetical judgment and Proposition 4.1. □

Lemma 4.4 (Elimination Principles).

1. $\Gamma\,, a : 0 \gg \mathit{case}\,\{\,\}\,(a) \in B$.

2. *Suppose that* $\Gamma \gg A_1$ *type,* $\Gamma \gg A_2$ *type, and* $\Gamma\,, a : A_1 + A_2 \gg B$ *type. If*
$$\Gamma\,, a_1 : A_1 \gg N_1 \doteq N_1' \in B\,[1 \cdot a_1 / a]$$
and
$$\Gamma\,, a_2 : A_2 \gg N_2 \doteq N_2' \in B\,[2 \cdot a_2 / a],$$
then
$$\Gamma\,, a : A_1 + A_2 \gg \mathit{case}\,\{a_1.N_1 \mid a_2.N_2\}\,(a) \doteq \mathit{case}\,\{a_1.N_1' \mid a_2.N_2'\}\,(a) \in B.$$

Proof. These follow from the the definition of the hypothetical judgment and Proposition 4.1. The clause for nullary case analysis (item 1) is immediate, because no closed terms are related at type 0. In particular, the standard precondition $\Gamma\,, a : 0 \gg B$ type holds vacuously.

The clause for binary case analysis (item 2) follows from Proposition 4.1 and the meaning of the hypothetical judgment. In particular, suppose that $\gamma \doteq \gamma' \in \Gamma\,, a : A_1 + A_2$. Let $M = \gamma(a)$ and $M' = \gamma'(a)$, and note that $M \doteq M' \in A_1\,\gamma + A_2\,\gamma$. By Proposition 4.1 either $M \Downarrow 1 \cdot M_1$ and $M' \Downarrow 1 \cdot M_1'$ and $M_1 \doteq M_1' \in A_1$, or $M \Downarrow 2 \cdot M_2$ and $M' \Downarrow 2 \cdot M_2'$ and $M_2 \doteq M_2' \in A_2$. In either case the result follows from the suppositions and closure under reverse evaluation (Theorem 2.2). □

Lemma 4.5 (Computation Principles). *Suppose that* $\Gamma \gg A_1$ *type,* $\Gamma \gg A_2$ *type, and* $\Gamma\,, a : A_1 + A_2 \gg B$ *type.*

1. *If* $\Gamma \gg M_1 \doteq M_1' \in A_1$ *and* $\Gamma\,, a_1 : A_1 \gg N_1 \doteq N_1' \in B\,[1 \cdot a_1 / a]$*, then*
$$\Gamma\,, a : A_1 + A_2 \gg \mathit{case}\,\{a_1.N_1 \mid a_2.N_2\}\,(1 \cdot M_1) \doteq N_1'\,[M_1'/a_1] \in B\,[1 \cdot M_1 / a].$$

2. *If* $\Gamma \gg M_2 \doteq M_2' \in A_2$ *and* $\Gamma\,, a_2 : A_2 \gg N_2 \doteq N_2' \in B\,[2 \cdot a_2 / a]$*, then*
$$\Gamma \gg \mathit{case}\,\{a_1.N_1 \mid a_2.N_2\}\,(2 \cdot M_2) \doteq N_2'\,[M_2'/a_2] \in B\,[2 \cdot M_2 / a].$$

Proof. These are proved similarly to the clause for the binary case analysis in the proof of Theorem 4.4. □

Lemma 4.6 (Unicity Principles).

1. *Suppose that* $\Gamma\,, a : 0 \gg B$ *type. If* $\Gamma\,, a : 0 \gg P \in B$*, then*
$$\Gamma\,, a : 0 \gg \mathit{case}\,\{\,\}\,(a) \doteq P \in B.$$

2. *Suppose that* $\Gamma \gg A_1$ *type,* $\Gamma \gg A_2$ *type, and* $\Gamma\,, a : A_1 + A_2 \gg B$ *type. If* $\Gamma\,, a : A_1 + A_2 \gg P \doteq P' \in B$*, then*
$$\Gamma\,, a : A_1 + A_2 \gg \mathit{case}\,\{a_1.P\,[1 \cdot a_1 / a] \mid a_2.P\,[2 \cdot a_2 / a]\}\,(a) \doteq P' \in B.$$

Constructing a type system for sums.

4.2 Formalisms

The syntax of the formal theory of positive types is given by the following grammar:

$$\begin{aligned} A &::= 0 \mid A_1 + A_2 \\ M &::= \mathsf{case}\,[a.A]\,\{\,\}\,(M) \mid \mathsf{in}\,[1]\,[A_1\,,A_2]\,(M) \mid \mathsf{in}\,[2]\,[A_1\,,A_2]\,(M) \mid \\ &\quad \mathsf{case}\,[a.A]\,[A_1\,,A_2]\,\{a_1.N_1 \mid a_2.N_2\}\,(M) \end{aligned}$$

The syntax requires additional type information that is not needed for evaluation, but rather to ensure that the formal typing judgment is decidable.

The basic formalism for positive types is inductively defined by the rules given in Figures 4.1 and 4.2. Definitional equivalence is defined as the least congruence satisfying the inversion principles for sums (rules MQ-+-C-L and MQ-+-C-R). Ideally definitional equivalence would characterize the uniqueness of the elimination form among terms with a free variable of sum type that validate the inversion principles under substitution. This amounts to a proof by induction, which for sums amounts to a binary case analysis. It can be summarized by the following principle, a generalization of the *Shannon expansion* for booleans, stating that any term with a free variable of sum type is definitionally equivalent to one whose outermost form is a case analysis on that variable:

MQ-+-U

$$\frac{\Gamma\,,a : A_1 + A_2 \vdash M : A \qquad M_1 \triangleq \mathsf{in}\,[1]\,[A_1\,,A_2]\,(a_1) \qquad M_2 \triangleq \mathsf{in}\,[2]\,[A_1\,,A_2]\,(a_2)}{\Gamma\,,a : A_1 + A_2 \vdash M \equiv \mathsf{case}\,[a.A]\,[A_1\,,A_2]\,\{a_1.M\,[M_1/a] \mid a_2.M\,[M_2/a]\}\,(a) : A}$$

This equation specifies that the behavior of M as a function of a is fully determined by its behavior on each of the two possible introductory forms.

Deciding equivalence in the presence of this principle would appear to require an intractible cascade of case analyses.[1] For this reason limited special cases, called *commuting conversions*, are often considered in its stead. As an example, application is often postulated to commute with case analysis:[2]

MQ-+-CA

$$\frac{P \triangleq \mathsf{ap}\,(\mathsf{case}\,[a.B_1 \to B_2]\,[A_1\,,A_2]\,\{a_1.M_1 \mid a_2.M_2\}\,(M)\,,N) \qquad P' \triangleq \mathsf{case}\,[a.B_1 \to B_2]\,[A_1\,,A_2]\,\{a_1.\mathsf{ap}\,(M_1\,,N) \mid a_2.\mathsf{ap}\,(M_2\,,N)\}\,(M)}{\Gamma \vdash P \equiv P' : B_2\,[M/a]}$$

Similarly, projections may be postulated to commute with case analysis, as may be case analysis with itself. All of these rules are derivable from rule MQ-+-U, but are not sufficient to derive it. The decision problem is complicated by the commuting conversions, there being no obvious means of implementing it short of the full rule of expansion.

[1]More precisely, the special case of booleans is equivalent to conjectures in computational complexity theory for which an exponential lower bound is widely anticipated, though as yet unproved. The general case cannot be less difficult.

[2]The required typing premises are omitted for the sake of concision.

$$\frac{\Gamma\ \mathsf{ctx}}{\Gamma \vdash 0}\ \textsc{tf-0} \qquad \frac{\Gamma\ \mathsf{ctx}}{\Gamma \vdash 0 \equiv 0}\ \textsc{tq-0}$$

$$\frac{\Gamma \vdash M : 0 \qquad \Gamma, a : 0 \vdash A}{\Gamma \vdash \mathsf{case}\,[a.A]\,\{\,\}\,(M) : A\,[M/a]}\ \textsc{mf-0-e}$$

$$\frac{\Gamma \vdash M \equiv M' : 0 \qquad \Gamma, a : 0 \vdash A \equiv A'}{\Gamma \vdash \mathsf{case}\,[a.A]\,\{\,\}\,(M) \equiv \mathsf{case}\,[a.A']\,\{\,\}\,(M') : A\,[M/a]}\ \textsc{mq-0-e}$$

Figure 4.1: Formal Type System for Zero Type

4.3 Interpretation

Definition 4.3. *The* erasure *of typed to untyped terms is defined by the following clauses:*

$$\begin{aligned}
|0| &= 0 \\
|\mathit{case}\,[a.A]\,\{\,\}\,(M)| &= \mathit{case}\,\{\,\}\,(|M|) \\
|A_1 + A_2| &= |A_1| + |A_2| \\
|\mathit{in}\,[1]\,[A_1, A_2]\,(M)| &= 1 \cdot |M| \\
|\mathit{in}\,[2]\,[A_1, A_2]\,(M)| &= 2 \cdot |M| \\
|\mathit{case}\,[a.A]\,[A_1, A_2]\,\{a_1.M_1 \mid a_2.M_2\}\,(M)| &= \mathit{case}\,\{a_1.|M_1| \mid a_2.|M_2|\}\,(|M|)
\end{aligned}$$

Theorem 4.7.

1. *If* Γ *ctx, then* $|\Gamma|$ *ctx.*
2. *If* $\Gamma \vdash A$*, then* $|\Gamma| \gg |A|$ *type.*
3. *If* $\Gamma \vdash A \equiv A'$*, then* $|\Gamma| \gg |A| \doteq |A'|$.
4. *If* $\Gamma \vdash M : A$*, then* $|\Gamma| \gg |M| \in |A|$.
5. *If* $\Gamma \vdash M \equiv M' : A$*, then* $|\Gamma| \gg |M| \doteq |M'| \in |A|$.

Proof. By simultaneous induction on the derivation of the formal judgments. The generic rules given in Chapter 2 are validated by the definition of the semantic judgments. The rules given in Figures 4.1 and 4.2 are handled by appeal to Lemmas 4.3 and 4.4. □

4.4 Exercises

1. Define the booleans in terms of sum and unit types. More generally, define enumeration types.
2. Give positive formulations of unit and product types. What happens if you try to give a positive formulation for function types?

TF-+

$$\frac{\Gamma \vdash A_1 \qquad \Gamma \vdash A_2}{\Gamma \vdash A_1 + A_2}$$

TQ-+

$$\frac{\Gamma \vdash A_1 \equiv A_1' \qquad \Gamma \vdash A_2 \equiv A_2'}{\Gamma \vdash A_1 + A_2 \equiv A_1' + A_2'}$$

MF-+-I

$$\frac{\Gamma \vdash A_1 \qquad \Gamma \vdash A_2 \qquad \Gamma \vdash M : A_1}{\Gamma \vdash \mathsf{in}\,[1]\,[A_1\,,A_2]\,(M) : A_1 + A_2}$$

MF-+-I

$$\frac{\Gamma \vdash A_1 \qquad \Gamma \vdash A_2 \qquad \Gamma \vdash M : A_2}{\Gamma \vdash \mathsf{in}\,[2]\,[A_1\,,A_2]\,(M) : A_1 + A_2}$$

MQ-+-I-L

$$\frac{\Gamma \vdash A_1 \equiv A_1' \qquad \Gamma \vdash A_2 \equiv A_2' \qquad \Gamma \vdash M \equiv M' : A_1}{\Gamma \vdash \mathsf{in}\,[1]\,[A_1\,,A_2]\,(M) \equiv \mathsf{in}\,[1]\,[A_1'\,,A_2']\,(M') : A_1 + A_2}$$

MQ-+-I-R

$$\frac{\Gamma \vdash A_1 \equiv A_1' \qquad \Gamma \vdash A_2 \equiv A_2' \qquad \Gamma \vdash M \equiv M' : A_2}{\Gamma \vdash \mathsf{in}\,[2]\,[A_1\,,A_2]\,(M) \equiv \mathsf{in}\,[2]\,[A_1'\,,A_2']\,(M') : A_1 + A_2}$$

MF-+-E

$$\frac{\begin{array}{c}\Gamma \vdash A_1 \qquad \Gamma \vdash A_2 \qquad \Gamma\,,a : A_1 + A_2 \vdash A \qquad \Gamma \vdash M : A_1 + A_2 \\ \Gamma\,,a_1 : A_1 \vdash N_1 : A\,[\mathsf{in}\,[1]\,[A_1\,,A_2]\,(a_1)/a] \qquad \Gamma\,,a_2 : A_2 \vdash N_2 : A\,[\mathsf{in}\,[2]\,[A_1\,,A_2]\,(a_2)/a]\end{array}}{\Gamma \vdash \mathsf{case}\,[a.A]\,[A_1\,,A_2]\,\{a_1.N_1 \mid a_2.N_2\}\,(M) : A\,[M/a]}$$

MQ-+-E

$$\frac{\begin{array}{c}\Gamma \vdash A_1 \equiv A_1' \qquad \Gamma \vdash A_2 \equiv A_2' \qquad \Gamma\,,a : A_1 + A_2 \vdash A \equiv A' \qquad \Gamma \vdash M \equiv M' : A_1 + A_2 \\ \Gamma\,,a_1 : A_1 \vdash N_1 \equiv N_1' : A\,[\mathsf{in}\,[1]\,[A_1\,,A_2]\,(a_1)/a] \\ \Gamma\,,a_2 : A_2 \vdash N_2 \equiv N_2' : A\,[\mathsf{in}\,[2]\,[A_1\,,A_2]\,(a_2)/a]\end{array}}{\Gamma \vdash \mathsf{case}\,[a.A]\,[A_1\,,A_2]\,\{a_1.N_1 \mid a_2.N_2\}\,(M) \equiv \mathsf{case}\,[a.A']\,[A_1'\,,A_2']\,\{a_1.N_1' \mid a_2.N_2'\}\,(M') : A\,[M/a]}$$

MQ-+-C-L

$$\frac{\begin{array}{c}\Gamma \vdash A_1 \qquad \Gamma \vdash A_2 \qquad \Gamma\,,a : A_1 + A_2 \vdash A \qquad \Gamma \vdash M \equiv \mathsf{in}\,[1]\,[A_1\,,A_2]\,(M_1) : A_1 + A_2 \\ \Gamma\,,a_1 : A_1 \vdash N_1 : A\,[\mathsf{in}\,[1]\,[A_1\,,A_2]\,(a_1)/a] \qquad \Gamma\,,a_2 : A_2 \vdash N_2 : A\,[\mathsf{in}\,[2]\,[A_1\,,A_2]\,(a_2)/a]\end{array}}{\Gamma \vdash \mathsf{case}\,[a.A]\,[A_1\,,A_2]\,\{a_1.N_1 \mid a_2.N_2\}\,(M) \equiv N_1\,[M_1/a_1] : A\,[M/a]}$$

MQ-+-C-R

$$\frac{\begin{array}{c}\Gamma \vdash A_1 \qquad \Gamma \vdash A_2 \qquad \Gamma\,,a : A_1 + A_2 \vdash A \qquad \Gamma \vdash M \equiv \mathsf{in}\,[2]\,[A_1\,,A_2]\,(M_2) : A_1 + A_2 \\ \Gamma\,,a_1 : A_1 \vdash N_1 : A\,[\mathsf{in}\,[1]\,[A_1\,,A_2]\,(a_1)/a] \qquad \Gamma\,,a_2 : A_2 \vdash N_2 : A\,[\mathsf{in}\,[2]\,[A_1\,,A_2]\,(a_2)/a]\end{array}}{\Gamma \vdash \mathsf{case}\,[a.A]\,[A_1\,,A_2]\,\{a_1.N_1 \mid a_2.N_2\}\,(M) \equiv N_2\,[M_2/a_2] : A\,[M/a]}$$

Figure 4.2: Formal Type System for Sum Types

Chapter 5

Dependency

The type systems considered in Chapters 3 and 4 allow for the (unrealized) possibility that types *depend on* elements of other types. That is, a type may, in general, be an instance of a *family of types* indexed by the elements of some other type. For example, the type $\mathsf{S}\,(n)$ of sequences of natural numbers of length n, itself a natural number, defines a family of types indexed by the type of natural numbers. Thus, in general, a type may be an instance of a family, which would thereby depend on an element of its index type. However, the family in question need not be a primitive family, such as the type of sequences depending on their length, but could be *any* computation on indices that yields a type.

Computational type theory accommodates such families naturally, because types are just certain programs. For example, in the presence of booleans one may judge

$$a : \mathsf{B} \gg \mathsf{if}\,\{\mathsf{B} \mid \mathsf{1}\}\,(a)\ \mathsf{type}$$

and, correspondingly, that

$$a : \mathsf{B} \gg \mathsf{if}\,\{\mathsf{tt} \mid \langle\,\rangle\}\,(a) \in \mathsf{if}\,\{\mathsf{B} \mid \mathsf{1}\}\,(a).$$

In each case the meaning of the judgment involves evaluation, after substitution, of a conditional expression that evaluates to a canonical type or canonical element of that type.

Formal type theory, by dint of drawing an *a priori* distinction between types and terms, requires special provisions to allow types to be computed. One approach is to rely on *universes*, which are considered in **??**. A universe is a type whose elements of types, and so it is possible to compute types non-trivially by computing elements of some universe. Despite the name, a universe can never be universal; in particular, the universe cannot be construed as an element of itself. The fallback is to consider an infinitude of universes so that although no universe can contain every type, nevertheless every type is an element of some universe.

Another approach is to enrich the formalism with what are known as *large eliminations forms* that duplicate the elimination forms for the positive types at the type level. Thus, for example, the syntax of types is extended with the type expression $\mathsf{If}\,\{A_1 \mid A_2\}\,(M)$ such that the formal typing judgments

$$a : \mathsf{B} \vdash \mathsf{If}\,\{\mathsf{B} \mid \mathsf{1}\}\,(a)$$

and

$$a : \mathsf{B} \vdash \mathsf{if}\,[a.\mathsf{If}\,\{\mathsf{B} \mid 1\}\,(a)]\,\{\mathsf{tt} \mid \langle\,\rangle\}\,(a) : \mathsf{If}\,\{\mathsf{B} \mid 1\}\,(a)$$

would both be derivable, provided that the inversion principles for booleans would be definitional equivalences. Large eliminations apply to any type at all, and there is no need for universes to compute types from positive data.

5.1 Semantics

The syntax generalizes that of the negative fragment by permitting function and product types for which the second type *depends* on the first via a specified bound variable.

Definition 5.1 (Untyped Syntax)**.** *The untyped syntax of dependent types includes these forms of expression:*

$$E ::= a : E_1 \to E_2 \mid \lambda\,(a.E) \mid \mathsf{ap}\,(E_1\,,E_2) \mid a : E_1 \times E_2 \mid \langle E_1\,,E_2\rangle \mid E \cdot 1 \mid E \cdot 2$$

Thus, one may consider the type $a : \mathsf{N} \to \mathsf{S}\,(a)$ of functions defined on the (as yet to be defined) type N of natural numbers that result in a sequene whose length is determined by its argument. Similarly, the type $a : \mathsf{N} \times \mathsf{S}\,(a)$ is the type of pairs of a natural number and a sequence of length given by that number.

The dynamics is the same as for the negative types (see Figure 3.1), which will not be repeated here. But note that both $a : E_1 \to E_2$ and $a : E_1 \times E_2$ are both values, regardless of the forms of E_1 or E_2.

Proposition 5.1 (Dependent Type Constructors)**.** *There exists a semantic type system with the following characteristics:*

1. *The judgment $A \doteq A'$ holds iff $A \Downarrow A_0$, $A' \Downarrow A_0'$, and one of the following conditions holds:*
   (a) *A_0 is $a : A_1 \to A_2$, A_0' is $a : A_1' \to A_2'$, $A_1 \doteq A_1'$, and $a : A_1 \gg A_2 \doteq A_2'$.*
   (b) *A_0 is $a : A_1 \times A_2$, A_0' is $a : A_1' \times A_2'$, $A_1 \doteq A_1'$, and $a : A_1 \gg A_2 \doteq A_2'$.*
2. *Given that A type, the judgment $M \doteq M' \in A$ holds iff $A \Downarrow A_0$ and either*
   (a) *A_0 is $a : A_1 \to A_2$, $M \Downarrow \lambda\,(a.N)$, $M' \Downarrow \lambda\,(a.N')$, and $a : A_1 \gg N \doteq N' \in A_2$, or*
   (b) *A_0 is $a : A_1 \times A_2$, $M \Downarrow \langle M_1\,,M_2\rangle$, $M' \Downarrow \langle M_1'\,,M_2'\rangle$, $M_1 \doteq M_1' \in A_1$, and $M_2 \doteq M_2' \in A_2\,[M_1\,/\,a]$.*

As an example, the following judgments are true in the type system specified by Proposition 5.1.

1. $\lambda\,(a.\mathsf{if}\,\{\mathsf{tt} \mid \langle\,\rangle\}\,(a)) \in a : \mathsf{B} \to \mathsf{if}\,\{\mathsf{B} \mid 1\}\,(a)$.
2. $\langle \mathsf{tt}\,,\mathsf{ff}\rangle \in a : \mathsf{B} \times \mathsf{if}\,\{\mathsf{B} \mid 1\}\,(a)$.

Lemma 5.2 (Formation Principles)**.** *If $\Gamma \gg A_1 \doteq A_1'$ and $\Gamma\,, a : A_1 \gg A_2 \doteq A_2'$, then*

1. *$\Gamma \gg a : A_1 \to A_2 \doteq a : A_1' \to A_2'$, and*
2. *$\Gamma \gg a : A_1 \times A_2 \doteq a : A_1' \times A_2'$.*

Lemma 5.3 (Introduction Principles).

1. *If $\Gamma, a : A_1 \gg M \doteq M' \in A_2$, then $\Gamma \gg \lambda(a.M) \doteq \lambda(a.M') \in a : A_1 \to A_2$.*
2. *If $\Gamma \gg M_1 \doteq M_1' \in A_1$ and $\Gamma \gg M_2 \doteq M_2' \in A_2[M_1/a]$, then $\Gamma \gg \langle M_1, M_2\rangle \doteq \langle M_1', M_2'\rangle \in a : A_1 \times A_2$.*

Lemma 5.4 (Elimination Principles).

1. *If $\Gamma \gg M \doteq M' \in a : A_1 \to A_2$ and $\Gamma \gg M_1 \doteq M_1' \in A_1$, then*
$$\Gamma \gg \mathsf{ap}(M, M_1) \doteq \mathsf{ap}(M', M_1') \in A_2[M_1/a].$$
2. *If $\Gamma \gg M \doteq M' \in a : A_1 \times A_2$, then*
$$\Gamma \gg M \cdot 1 \doteq M' \cdot 1 \in A_1 \quad \text{and} \quad \Gamma \gg M \cdot 2 \doteq M' \cdot 2 \in A_2[M \cdot 1/a].$$

Lemma 5.5 (Computation Principles).

1. *If $\Gamma, a_1 : A_1 \gg M_2 \in A_2$ and $\Gamma \gg M_1 \in A_1$, then*
$$\Gamma \gg \mathsf{ap}(\lambda(a_1.M_2), M_1) \doteq M_2[M_1/a_1] \in A_2[M_1/a].$$
2. *If $\Gamma \gg M_1 \in A_1$ and $\Gamma \gg M_2 \in A_2[M_1/a]$, then*
$$\Gamma \gg \langle M_1, M_2\rangle \cdot 1 \doteq M_1 \in A_1 \quad \text{and} \quad \Gamma \gg \langle M_1, M_2\rangle \cdot 2 \doteq M_2 \in A_2[M_1/a].$$

Lemma 5.6 (Unicity Principles).

1. *If $\Gamma \gg M \in a : A_1 \to A_2$, then $\Gamma \gg M \doteq \lambda(a.\mathsf{ap}(M, a)) \in a : A_1 \to A_2$.*
2. *If $\Gamma \gg M \in a : A_1 \times A_2$, then $\Gamma \gg M \doteq \langle M \cdot 1, M \cdot 2\rangle \in a : A_1 \times A_2$.*

The justification of Proposition 5.1 is similar to that of Proposition 3.2, with the additional complication that the product and function types are dependent. The key to understanding the construction is to consider that the type A_1, and all instances $A_2[M_1/a]$ of the family $a.A_2$ by elements $M_1 \in A_1$ of the index type, precede the types $a : A_1 \to A_2$ and $a : A_1 \times A_2$ in that the definitions of the latter two types are given in terms of these prior types. To be sure, the precedence ordering may be transfinite (a type may have infinitely many predecessors), but it is nevertheless well-founded (the preceding types are structurallly smaller than the dependent arrow and product types.) Given that the precedence ordering is well-founded, it follows that the dependent function and product types are well-defined and satisfy the conditions given by Proposition 5.1.

The construction of the required type system is very similar to the analogous construction given in Chapter 3. The only difference is in the definition of the operator T on candidate type systems given in Definition 3.3, wherein the clauses for function and binary product types are replaced by their dependent forms. In the following definition ϕ ranges over binary relations on closed terms, and Φ ranges over partial functions mapping closed terms to binary relations on closed terms.

Definition 5.2. *The operator T on candidate type systems is defined by the equation*

$$T(\tau) = \textsc{One} \cup \textsc{Sig}(\tau) \cup \textsc{Pi}(\tau),$$

where

$$\begin{aligned}
\textsc{One} &= \{ (1, 1, \mathbf{1}) \} \\
\textsc{Sig}(\tau) &= \{ (a : A_1 \times A_2, a : A_1' \times A_2', \mathbf{\Sigma}(\phi_1, \Phi_2)) \mid \tau(A_1, A_1', \phi_1) \wedge \tau(A_2[M_1/a], A_2'[M_1/a], \Phi_2(M_1)) \} \\
\textsc{Pi}(\tau) &= \{ (a : A_1 \rightarrow A_2, a : A_1' \rightarrow A_2', \mathbf{\Pi}(\phi_1, \Phi_2)) \mid \tau(A_1, A_1', \phi_1) \wedge \tau(A_2[M_1/a], A_2'[M_1/a], \Phi_2(M_1)) \}
\end{aligned}$$

with

$$\begin{aligned}
\mathbf{1} &= \{ (\langle\rangle, \langle\rangle) \} \\
\mathbf{\Sigma}(\phi_1, \Phi_2) &= \{ (\langle M_1 , M_2 \rangle, \langle M_1' , M_2' \rangle) \mid \phi_1(M_1, M_1') \wedge \Phi_2(M_1)(M_2, M_2') \} \\
\mathbf{\Pi}(\phi_1, \Phi_2) &= \{ (\lambda\,(a.M_2), \lambda\,(a.M_2')) \mid \phi_1(M_1, M_1') \supset \Phi_2(M_1)(M_2[M_1/a], M_2'[M_1'/a]) \}
\end{aligned}$$

In the clauses for the dependent types the partial function Φ_2 is implicitly defined by the specified conditions. It need not be defined for all M_1, and is defined by arbitrary choice for certain arguments, because candidate type systems need not be functional. But if τ is a proper type system, then there is at most one choice of $\Phi_2(M_1)$ for each M_1, which is all that matters. When $\tau \models A_1$ type and $\tau \models a : A_1 \gg A_2$ type, then $\tau \models M_1 \doteq M_1' \in A_1$ implies that both $\Phi_2(M_1)$ and $\Phi_2(M_1')$ exist and are the same relation.

It is easy to check that the operator T given by Definition 5.2 is monotone, and thus has a least fixed point (with respect to inclusion of candidate type systems). The fixed point clearly satisfies the conditions stated in Proposition 5.1.

5.2 Formalisms

The syntax of formal dependent types is given by the following grammar:

$$\begin{aligned}
A &::= \Sigma\,(a : A_1 , A_2) \mid \Pi\,(a : A_1 , A_2) \\
M &::= \lambda[A](a.M) \mid \mathtt{ap}\,(M_1 , M_2) \mid \langle M_1 , M_2 \rangle \mid M \cdot 1 \mid M \cdot 2
\end{aligned}$$

The syntax of terms replicates that for product and arrow types given in Chapter 3. There is no need to distinguish the two, because the product and arrow types are simply the corresponding dependent types with degenerate dependency,

$$A_1 \times A_2 \triangleq \Sigma\,(\_ : A_1 , A_2) \quad \textit{and} \quad A_1 \rightarrow A_2 \triangleq \Pi\,(\_ : A_1 , A_2),$$

wherein the underscore indicates a term variable that does not occur within A_2.

The rules defining formal dependent type theory are given in Figure 5.1 and Figure 5.2. One may also consider unicity principles corresponding to those given in Figure 3.3, albeit in their dependently typed forms. Doing so ensures that the type constructors enjoy their full universal properties, but it also complicates the proof of decidability of definitional equivalence.

Explain this.

$$\text{TF-}\Sigma \quad \frac{\Gamma \vdash A_1 \qquad \Gamma , a : A_1 \vdash A_2}{\Gamma \vdash \Sigma\,(a : A_1 , A_2)}$$

$$\text{TQ-}\Sigma \quad \frac{\Gamma \vdash A_1 \equiv A_1' \qquad \Gamma , a : A_1 \vdash A_2 \equiv A_2'}{\Gamma \vdash \Sigma\,(a : A_1 , A_2) \equiv \Sigma\,(a : A_1' , A_2')}$$

$$\text{MF-}\Sigma\text{-I} \quad \frac{\Gamma \vdash M_1 : A_1 \qquad \Gamma \vdash M_2 : A_2\,[M_1 / a]}{\Gamma \vdash \langle M_1 , M_2 \rangle : \Sigma\,(a : A_1 , A_2)}$$

$$\text{MQ-}\Sigma\text{-I} \quad \frac{\Gamma \vdash M_1 \equiv M_1' : A_1 \qquad \Gamma \vdash M_2 \equiv M_2' : A_2\,[M_1 / a]}{\Gamma \vdash \langle M_1 , M_2 \rangle \equiv \langle M_1' , M_2' \rangle : \Sigma\,(a : A_1 , A_2)}$$

$$\text{MF-}\Sigma\text{-E-L} \quad \frac{\Gamma \vdash M : \Sigma\,(a : A_1 , A_2)}{\Gamma \vdash M \cdot 1 : A_1}$$

$$\text{MF-}\Sigma\text{-E-R} \quad \frac{\Gamma \vdash M : \Sigma\,(a : A_1 , A_2)}{\Gamma \vdash M \cdot 2 : A_2\,[M \cdot 1 / a]}$$

$$\text{MQ-}\Sigma\text{-E-L} \quad \frac{\Gamma \vdash M \equiv M' : \Sigma\,(a : A_1 , A_2)}{\Gamma \vdash M \cdot 1 \equiv M' \cdot 1 : A_1}$$

$$\text{MQ-}\Sigma\text{-E-R} \quad \frac{\Gamma \vdash M \equiv M' : \Sigma\,(a : A_1 , A_2)}{\Gamma \vdash M \cdot 2 \equiv M' \cdot 2 : A_2\,[M \cdot 1 / a]}$$

$$\text{MQ-}\Sigma\text{-C-L} \quad \frac{\Gamma \vdash M_1 : A_1 \qquad \Gamma \vdash M_2 : A_2\,[M_1 / a]}{\Gamma \vdash \langle M_1 , M_2 \rangle \cdot 1 \equiv M_1 : A_1}$$

$$\text{MQ-}\Sigma\text{-C-R} \quad \frac{\Gamma \vdash M_1 : A_1 \qquad \Gamma \vdash M_2 : A_2\,[M_1 / a]}{\Gamma \vdash \langle M_1 , M_2 \rangle \cdot 2 \equiv M_2 : A_2\,[M_1 / a]}$$

Figure 5.1: Formal Sigma Types

$$\textsc{tf-}\Pi\quad \frac{\Gamma \vdash A_1 \qquad \Gamma\,,a : A_1 \vdash A_2}{\Gamma \vdash \Pi\,(a : A_1\,, A_2)}$$

$$\textsc{tq-}\Pi\quad \frac{\Gamma \vdash A_1 \equiv A_1' \qquad \Gamma\,,a : A_1 \vdash A_2 \equiv A_2'}{\Gamma \vdash \Pi\,(a : A_1\,, A_2) \equiv \Pi\,(a : A_1'\,, A_2')}$$

$$\textsc{mf-}\Pi\textsc{-i}\quad \frac{\Gamma \vdash A_1 \qquad \Gamma\,,a : A_1 \vdash M_2 : A_2}{\Gamma \vdash \lambda[A_1](a.M_2) : \Pi\,(a : A_1\,, A_2)}$$

$$\textsc{mq-}\Pi\textsc{-i}\quad \frac{\Gamma \vdash A_1 \equiv A_1' \qquad \Gamma\,,a : A_1 \vdash M_2 \equiv M_2' : A_2}{\Gamma \vdash \lambda[A_1](a.M_2) \equiv \lambda[A_1'](a.M_2') : \Pi\,(a : A_1\,, A_2)}$$

$$\textsc{mf-}\Pi\textsc{-e}\quad \frac{\Gamma \vdash M : \Pi\,(a : A_1\,, A_2) \qquad \Gamma \vdash M_1 : A_1}{\Gamma \vdash \mathsf{ap}\,(M\,, M_1) : A_2\,[M_1\,/\,a]}$$

$$\textsc{mq-}\Pi\textsc{-e}\quad \frac{\Gamma \vdash M \equiv M' : \Pi\,(a : A_1\,, A_2) \qquad \Gamma \vdash M_1 \equiv M_1' : A_1}{\Gamma \vdash \mathsf{ap}\,(M\,, M_1) \equiv \mathsf{ap}\,(M'\,, M_1') : A_2\,[M_1\,/\,a]}$$

$$\textsc{mq-}\Pi\textsc{-c}\quad \frac{\Gamma \vdash A_1 \qquad \Gamma\,,a : A_1 \vdash M_2 : A_2}{\Gamma \vdash \mathsf{ap}\,(\lambda[A_1](a.M_2)\,, M_1) \equiv M_2\,[M_1\,/\,a] : A_2\,[M_1\,/\,a]}$$

Figure 5.2: Formal Pi Types

5.3 Interpretation

Definition 5.3. *The* erasure *of a typed expression to an untyped expression is defined as follows:*

$$|\Pi\,(a : A\,,B)| = a : |A| \to |B|$$
$$|\Sigma\,(a : A\,,B)| = a : |A| \times |B|$$

(The clauses for terms are those given in Definition 3.5.)

Theorem 5.7.

1. *If* Γ *ctx, then* $|\Gamma|$ *ctx.*
2. *If* $\Gamma \vdash A$*, then* $|\Gamma| \gg |A|$ *type.*
3. *If* $\Gamma \vdash A \equiv A'$*, then* $|\Gamma| \gg |A| \doteq |A'|$.
4. *If* $\Gamma \vdash M : A$*, then* $|\Gamma| \gg |M| \in |A|$.
5. *If* $\Gamma \vdash M \equiv M' : A$*, then* $|\Gamma| \gg |M| \doteq |M'| \in |A|$.

Proof. By appeal to Theorem 5.2, Theorem 5.3, Theorem 5.5, and Theorem 5.6. □

5.4 Exercises

1. Argue that "large eliminations" for negative types are not needed, but what if the negatives are formulated positively?

Chapter 6

Inductive Types

The positive types considered in Chapter 4 are instances of the more general concept of *inductive types*. Roughly speaking, the characteristic feature of inductive types is that they are generated by a specified collection of *constructors* that build elements out of other elements the type itself. For example, a type of binary trees might be generated by a constructor for the empty tree taking no arguments (equivalently, an argument of unit type) and a constructor for a binary node taking as arguments two (other) binary trees—or, rather, a pair of binary trees, which is to say an element of the product of the type of binary trees with itself. Which compound types are permitted as argument types for constructors determines the class of inductive types that may be defined. The most basic forms permit finite products of the inductive type with itself and other specified types. Richer forms permit, for example, finite sequences as arguments, or, more generally, any family of elements of the inductive type indexed by a fixed type. What *cannot* be permitted are sequences indexed by the inductive type itself, for then the elements of a type are determined in part by what elements are not in the type, opening the door to paradox by considering a value whose construction relies on the non-existence of the very thing it constructs.

The most basic inductive type is that of the natural numbers, which is treated separately for both illustrative and technical purposes. A rather general class of inductive types is given by the class of *well-founded trees*, or *W-types*. But there is no end to the possible variations one may consider so as to increase the class of permitted inductive types.

Appendix A

Answers to the Exercises

Appendix B

Background

1. Metavariable conventions. Color conventions. Color neutrality for generic operations.
2. Abstract binding trees. Identification convention. Capture-avoiding substitution, free variables. Color correspondence between variables.
   (a) The binding and scope conventions are implicit, using abstractors $a.E$ to bind a variable a within expression E.
   (b) Abstract binding trees are implicitly identified up to consistent renaming of bound variables.
   (c) Free variables fv.
   (d) Capture-avoiding substitution, written $E' [E/a]$, is total modulo renaming.
3. Inductive definitions by axioms and rules. Admissible and derivable rules.
4. Structural operational semantics. cf. λ-terms or TM indices.
5. Tarski fixed point theorem for relations. Where does it live?

A *substitution* γ is a finite function assigning closed expressions to variables. It is often convenient to display γ as $[E_1/x_1\,, \ldots, E_n/x_n]$, where $\gamma(x_i)$ is E_i for each $1 \leq i \leq n$ with $n \geq 0$. If E is an open expression, the action of γ on E, written $E\,\gamma$, is the result of simultaneously substituting
$gamma(x_i)$ for x_i in E. When γ is displayed as above, the action is written $E\,[E_1/x_1, \ldots, E_n/x_n]$, which specializes the notation for single substitution when $n = 1$.