

MARTIN-LÖF'S TYPE THEORY AS AN OPEN-ENDED FRAMEWORK*

YASUYUKI TSUKADA

*NTT Communication Science Laboratories
3-1 Morinosato-Wakamiya, Atsugi, Kanagawa, 243-0198 Japan*

Received 3 December 1998

Revised 22 April 1999

Communicated by M. Sato and Y. Toyama

ABSTRACT

This paper treats Martin-Löf's type theory as an open-ended framework composed of (i) flexibly extensible languages into which various forms of objects and types can be incorporated, (ii) their uniform, effectively given semantics, and (iii) persistently valid inference rules. The class of *expression systems* is introduced here to define an open-ended body of languages underlying the theory. Each expression system consists of two parts: the computational part is a structured lazy evaluation system with a bisimulation-like program equivalence; the structural part is a system of strictly positive inductive definitions for type constructors in terms of partial equivalence relations. Types and their objects are uniformly and inductively constructed from a given expression system as a *type system*, which can provide a semantics of the theory. Building on these concepts, this paper presents two main results. First, all the inference rules of the theory are *sound*; that is, they remain valid in every type system built from an extension of an initial expression system. This result gives a characterization of the class of types that can be introduced into the theory. Second, each type system is *complete* with respect to the underlying bisimulation-like program equivalence. This result provides a useful form of type-free equational reasoning in the theory.

Keywords: Martin-Löf's type theory, open-endedness, structured lazy evaluation systems, inductively defined types, bisimulation-like equivalences, partial equivalence relations.

1. Introduction

The study presented in this paper was motivated by a few questions arising from the following statements^a by Martin-Löf [33]:

Table 1 displays the primitive forms of expression used in the theory of types. New primitive forms of expression may of course be added when there is need of them.

*This paper is a full version of [48]. An expansion of the material in [47] has also been incorporated into the present paper.

^aParaphrased.

Table 1. Primitive forms of expression used in the theory of types.

Canonical	Noncanonical
$\Pi(A, B), \lambda(b)$	<code>apply(c, a)</code>
$\Sigma(A, B), \langle a, b \rangle$	<code>split(c, d)</code>
$A + B, \text{inl}(a), \text{inr}(b)$	<code>when(c, d, e)</code>
$\text{Eq}(A, a, b), \text{eq}$	<code>judge(c, d)</code>
\mathbf{N}_0	<code>case₀(c)</code>
$\mathbf{N}_1, 0_1$	<code>case₁(c, d₀)</code>
$\mathbf{N}_2, 0_2, 1_2$	<code>case₂(c, d₀, d₁)</code>
...	...
$\mathbf{N}, 0, \text{succ}(a)$	<code>natrec(c, d, e)</code>
$\text{List}(A), \text{nil}, \text{cons}(a, b)$	<code>listrec(c, d, e)</code>
$\mathbf{W}(A, B), \text{sup}(a, b)$	<code>wrec(c, d)</code>
$\mathbf{U}_0, \mathbf{U}_1, \mathbf{U}_2, \dots$	

The first statement is clear enough, but the second raises the following questions:

- (Q1) What are new primitive forms of expression?
- (Q2) When can they be validly added to the theory?
- (Q3) Why does the theory work when they are added?

This paper provides answers to these questions by presenting a mathematical interpretation of the “open-endedness” property in Martin-Löf’s Intuitionistic Type Theory (ITT) [33, 34].^b This semantical property, which was presented only implicitly in [33, 34] but which played a vital role in the design of the theory, is studied here from the mathematical viewpoint.

1.1. Aspects of Open-Endedness in ITT

In his semantical explanation [33, 34] of ITT, Martin-Löf explained the meaning of the selected basic concepts, *types* and their *objects*, by purely semantical means [35] in terms of the “primitive” notion, *methods*. For example, a judgement of the form “ A type,” which asserts that A is a type, means that the method A has a *canonical type* as its value, where the canonical type denoted by A is defined by prescribing how a *canonical object* of the type is formed as well as how two *equal canonical objects* of the type are formed. The only limitation on this prescription is that the equality relation in each canonical type must be reflexive, symmetric, and transitive. Another kind of judgement, of the form “ $a \in A$,” presupposes that A is a type and asserts that a is an object of type A . This means that the method a has as its value a canonical object of the canonical type denoted by A . In this semantical explanation there is no intensional restriction on the method a .

From this explanation the following statements can be inferred:

- if we have the judgement “ \mathbf{N} type,” then the type \mathbf{N} of natural numbers con-

^bAmong the many versions of ITT, the one treated in this paper is often called Martin-Löf’s polymorphic type theory (with extensional equality types). (See [38] for details.) Its universes are formulated here *à la* Russell rather than *à la* Tarski [34].

sists not only of $0, \text{succ}(0), \text{succ}(\text{succ}(0)), \dots$ but also of possible future objects such as $10+10, 10^{10}, 10000!$, and “a natural number expressed by using some real-valued functions introduced later”;

- if we have a judgement “ $f \in \mathbf{N} \rightarrow \mathbf{N}$,” then the object f is applicable to every possible future object of \mathbf{N} ;
- if we have a hypothetical judgement “ $C(x)$ type ($x \in \mathbf{N} \rightarrow \mathbf{N}$),” then the family C of types over $\mathbf{N} \rightarrow \mathbf{N}$ is defined in such a way as to anticipate the introduction of new objects of $\mathbf{N} \rightarrow \mathbf{N}$;
- if we have a hypothetical judgement “ $F(X) \in \mathbf{U}_0 (X \in \mathbf{U}_0)$,” then the function^c F on the first universe \mathbf{U}_0 is defined so that new, previously unimagined types may be introduced later.

The first three statements can be viewed as aspects of the *open-endedness of objects*; the last can be viewed as an aspect of the *open-endedness of types*. These are not explicitly presented in Martin-Löf’s papers but are implicit in his semantical explanation.

A similar kind of explanation can also be found in the Brouwer-Heyting-Kolmogorov interpretation of fundamental logical connectives; indeed, *proofs* of the propositions built from such connectives are semantically explained in terms of the primitive notion, *methods*. (See [45], for example.) Consequently, the *open-endedness of proofs* follows.

In brief, the open-endedness property in ITT is what enables ITT to incorporate new objects and types. This interesting property, however, must be expressed in a more specific and clearly understandable form so that it may be used to design new systems for programming and mathematics. One approach to this expression is to imagine that there is a certain series of languages underlying the theory and to define open-endedness to be the property that the inference rules of the theory remain valid under extensions of an initial underlying language. This approach leads us to an informal and general definition of open-endedness.

1.2. Informal Definition of Open-Endedness

In general, a *theory*—that is, a framework composed of languages, their semantics, and inference rules—is called *open-ended* provided that when a programmer or a mathematician using the theory works in an initial language L_0 whose semantics $M(L_0)$ is given by a mapping M , the following hold for every possible sequence $L_0 \sqsubseteq L_1 \sqsubseteq L_2 \sqsubseteq \dots$ of extended languages:

- *semantical anticipation*: M also provides each L_i ($i \geq 1$) with its semantics $M(L_i)$ effectively;
- *validity persistence*: each inference rule in L_0 is valid not only in $M(L_0)$ but also in every $M(L_i)$ ($i \geq 1$).

Two remarks about this definition should be made here: they explain the significance and nontriviality of the open-endedness property.

^cThe “identity” $X.X$ is an example of such a function. The structural recursion on the universe, however, cannot be allowed because it implies that the universe is closed. (See Section 5.)

Remark 1.1 The present paper is primarily concerned with a basic theory for “interactive” proof-development systems. Since these systems need to support the dynamic and partial reasoning processes of human thought, it should be possible for newly invented concepts, such as novel proof techniques and original data types, to be introduced into the current language of such a theory. Instead of “adding” them from outside the language, we could of course restrict ourselves to “defining” them within the language. The flexibility inherent in being able to add them, however, is more important than the stability resulting from being able to introduce them only by defining them. This is because such definitions are sometimes impossible; and, even when they are possible, they often force us to handle very delicate and complicated encodings. It is essential that each concept be introduced in its most appropriate form. This paper, therefore, considers a theory whose languages can be extended by “adding” new concepts instead of only by “defining” them. The semantical anticipation and validity persistence properties are indispensable if such a theory is to be sensible. Indeed, ITT, the successful basis of such interactive proof-development systems as those described in [13, 38], is proved here to be open-ended.

Remark 1.2 The present paper considers a “semantics” to be a “term model” since its concern is for a particular kind of semantics whose domain becomes larger and larger as the interpreted language extends. Combined with the postulation that languages can be extended by “additions,” this consideration implies that the semantical anticipation and validity persistence properties are not trivial.

1.3. Overview of This Paper

This paper presents a mathematical interpretation of the open-endedness property in ITT; that is, it formally demonstrates semantical anticipation and validity persistence with regard to ITT.

First, the class of *expression systems* is introduced to define an open-ended body of languages underlying the theory. Each expression system consists of two parts. The “computational” part is an untyped programming language called an *evaluation system*. It specifies a set of *canonical operators*, a set of *noncanonical operators*, and a set of *evaluation rules*. These in turn generate the set of *canonical terms* (which are “values” of programs), the set of *noncanonical terms* (which are “programs” to be evaluated), and the *evaluation relation* between two terms. In other words, evaluation systems determine the operational semantics for expression systems. The “structural” part of an expression system is a *prescription system*. It grants to special canonical operators the privileges of *type constructors*, and it assigns to each of the type constructors the following:

- a *kind*—every type constructor in ITT may be regarded as a map that assigns a new canonical type to each bundle of “families of types” and associated “functions,” and the concept of kind is abstracted from the domains of such maps;

- an *equality prescription*, which can serve as a *strictly positive inductive definition* of the equality relation in each canonical type formed by the type constructor—the class of such prescriptions is systematically defined by using the kind of the type constructor. (Details are presented in Section 2.)

Second, the concept of *extension* of an expression system is defined in such a way as to preserve the meaning of the original expression system. The binary relation induced by the concept of extension turns out to be a partial ordering of expression systems. (Details and examples are presented in Section 3.)

Assume that a language underlying the theory is specified as an expression system L . Then *types* and their *objects* can be uniformly and inductively—that is, effectively—constructed from L as a particular form of term model, the *type system* \mathcal{M}_L . This type system is a partial mapping from the set of terms to the set of partial equivalence relations in the set of terms and can provide a semantics of ITT. Indeed, statements of the following forms:

$$T \text{ type, } T = T', \ t \in T, \ \text{and } t = t' \in T$$

will respectively be interpreted as

$$T \in \text{dom}(\mathcal{M}_L), \ \mathcal{M}_L(T) = \mathcal{M}_L(T'), \ (t, t) \in \mathcal{M}_L(T), \ \text{and } (t, t') \in \mathcal{M}_L(T).$$

The construction of \mathcal{M}_L is based on the concept of kind. The validity of the inductive definition of \mathcal{M}_L is guaranteed by this “kind-based” construction and is intrinsically due to *predicativity* in ITT. (Details are presented in Section 4.)

Building on these concepts, this paper presents two main results: “soundness” and “completeness.”

1.3.1. “Soundness”

The present paper shows that the inference rules given by Martin-Löf [33, 34] are *sound*; that is, they remain valid in every type system built from an extension of the initial expression system containing all the type constructors presented explicitly in [33, 34]. Thus, it completes the mathematical interpretation of the open-endedness property in ITT. (Details are presented in Section 5.)

The soundness result provides a characterization of the class of types that can be introduced into the theory. Specifically, it stipulates that the class should consist of types representable in some type system built from an extension of the initial expression system. We can use this criterion to determine whether specific new types can be validly added.

The three questions—(Q1), (Q2), and (Q3)—asked at the beginning of this section can then be answered as follows:

- (A1) Terms of expression systems are primitive forms of expression.
- (A2) They can be validly added to the theory whenever they are terms of some extension of the initial expression system.
- (A3) The theory works because its inference rules are built up to meet every series of type systems indexed by a sequence of expression systems that are extended successively from the initial expression system.

1.3.2. “Completeness”

The construction of type systems from given expression systems has so far been used to show the soundness of the inference rules of the theory. This construction, however, has an interesting character in itself. This is the second subject of the present paper.

Each evaluation system C —the computational part of an expression system—is an untyped programming language and gives rise to a natural concept of a bisimulation-like program equivalence \sim_C , which formulates an observational indistinguishability between two programs. Specifically, the equivalence \sim_C can be characterized as follows: if two terms of C are related by \sim_C and the evaluation of one of them terminates, then so does that of the other; furthermore, the two resulting values have the same outermost form and the corresponding components are also related by \sim_C . This equivalence can be regarded as a generalization of Abramsky’s *applicative bisimulation* [1]. For \sim_C to be useful, however, equals should be substitutable for equals. Indeed, it can be shown that \sim_C is a *congruence*, that is, $s \sim_C s'$ implies $\rho(s) \sim_C \rho(s')$ for each operator ρ .

The present paper shows a fundamental relationship between these congruences and type systems. That is, given an expression system L with an evaluation system C as its computational part, the type system \mathcal{M}_L is *complete* with respect to the congruence \sim_C : if T is a type in \mathcal{M}_L and $T \sim_C T'$, then T and T' are equal types in \mathcal{M}_L , and if t is an object of type T in \mathcal{M}_L and $t \sim_C t'$, then t and t' are equal objects of T in \mathcal{M}_L .

The completeness result provides a useful form of type-free equational reasoning. Specifically, the following form of reasoning can be allowed in ITT: “ T' type” and “ $T = T'$ ” can be inferred from “ T type,” provided that $T \sim_C T'$; moreover, “ $t' \in T$ ” and “ $t = t' \in T$ ” can be inferred from “ $t \in T$,” provided that $t \sim_C t'$. In particular, because the congruence \sim_C contains the redex-contractum relation underlying in ITT, we can freely replace redexes with contracta (and vice versa) when reasoning about objects and their types.

Consider, for example, the two types

$$(\mathbf{N} \rightarrow \mathbf{Bool}) \rightarrow \mathbf{U}_0 \quad \text{and} \quad \text{apply}(\lambda(x.x \rightarrow \mathbf{U}_0), \text{apply}(\lambda(x.x \rightarrow \mathbf{Bool}), \mathbf{N})).$$

Their equality is intuitively apparent because the former can be obtained from the latter by contracting the β -redexes. Formally, however, ITT [33, 34] has no general form of its computational inference rules that guarantees this kind of equality. Instead, to prove equality within the framework of ordinary logical inference rules of introduction or elimination form, we would have to find types for the redexes and show that these types are appropriate. The appropriate types for $\lambda(x.x \rightarrow \mathbf{U}_0)$, $\lambda(x.x \rightarrow \mathbf{Bool})$, and \mathbf{N} should be $\mathbf{U}_0 \rightarrow \mathbf{U}_1$, $\mathbf{U}_0 \rightarrow \mathbf{U}_0$, and \mathbf{U}_0 , respectively. The equational reasoning presented in this paper for a large class of inductively defined types generally eliminates this inefficiency and is valuable in practice, as supported by such experimental studies as the one described in [13].

Furthermore, this type-free equational reasoning can be generalized from the open-endedness viewpoint. The generalized form, which is the highlight of the

present paper, is useful in cases where the underlying languages are continually being extended by adding new programming concepts. (Details and examples are presented in Section 6.)

1.4. Related Work

There are two views of ITT: “untyped” and “typed.”

1.4.1. “Untyped” approach

Computational languages underlying ITT can, in general, be untyped. Martin-Löf has promoted this view and, in particular, has proposed that lazy evaluation plays a fundamental role [33, 34]. For example, $u \equiv \lambda(z.\text{apply}(z, z))$ and $\text{succ}(0) + \text{succ}(\text{apply}(u, u))$ are considered to have u itself and $\text{succ}(0 + \text{succ}(\text{apply}(u, u)))$, respectively, as their values. A difficulty arising from this view is summarized by the following remark^d made by Martin-Löf [33]:

What is significant is that Frege’s principle, that the value of an expression depends only on the values of its parts, is irretrievably lost. To make the language work in spite of this loss has been one of the most serious difficulties in the design of the theory of types.

The “untyped” view has also been promoted and developed by Allen, by Howe, and by Aczel, Carlisle, and Mendler. The present paper also deals with this view.

This paper’s formulation of the open-endedness property in ITT is a natural generalization of Allen’s non-type-theoretical (but mathematically comprehensible) reinterpretation [6] of ITT. Indeed, Allen’s reinterpretation, which is closely related to such realizability-like interpretations as those by Beeson [11], by Smith [41], and by Harper [20], is essentially the same as the single construction of the “minimal” type system from the initial expression system.

The formulation presented in this paper can also be viewed as a structural extension of Howe’s result of computational open-endedness [24] in that Howe’s can, in fact, be obtained by restricting the present formulation to allowing only extensions of the computational part of the initial expression system. Introducing the concept of a prescription system as the extensible structural part of an expression system, the present paper also treats the open-endedness of types.

The concept of an evaluation system—a generalization of an untyped λ -calculus—was introduced by Howe. In [23, 24, 10, 25], he used the approach of Plotkin [39] and Kahn [26] to extract a simple but fairly general operational structure from Martin-Löf’s informal description of the evaluation of terms [33, 34] and thereby developed the concept of a *structured lazy evaluation system*, which is essentially the same as that of an evaluation system used in the present paper. The syntax of evaluation systems has its origin in the work of Aczel [2]. *Combinatory reduction systems* [29] and *expression reduction systems* [28] are also developments

^dParaphrased.

from Aczel’s work. We are also indebted to Howe for the definition of bisimulation-like program equivalences and for the proof that they are congruences.

The completeness result—that is, \mathcal{M}_L is complete with respect to \sim_C —was first sketched in [23]; however, it was shown only for some restricted class of L containing a specific collection of type constructors. In other words, a proof sketch was given based on a case-by-case analysis of several basic type constructors. Clarification was not given for how each new type incorporated into ITT should be defined to make \mathcal{M}_L still complete with respect to \sim_C . The present paper demonstrates that this completeness result can be extended to a general class of inductively defined types. This is proved “uniformly” (instead of “on a case-by-case basis”) based on the uniform construction of types. In particular, it is shown that the “sequentiality” structure, which is found in the evaluation rules of each evaluation system and which is essential in proving \sim_C to be a congruence, also plays an important role in the inductive definitions of types so that \mathcal{M}_L is complete with respect to \sim_C .

Howe also constructed a classical model of ITT: into the computational part of the initial expression system, he introduced as new noncanonical operators a large collection of “oracles” for all functions in a certain set-theoretical universe, thereby creating a classical model in which schemas for the law of the excluded middle are valid [24]. (From a foundational viewpoint, this result corresponds to the well-known fact that the Brouwer-Heyting-Kolmogorov interpretation serves as a classical semantics when methods are understood to mean set-theoretical functions. From a practical viewpoint, this result may be applied to program development using classical objects.) Howe’s construction based on computational open-endedness is also applicable to the present formulation.

Aczel, Carlisle, and Mendler [37, 5] proposed the concept of an open-ended *framework*, thereby initiating a systematic approach to analyzing a series of extended languages and their semantics: the target example was the Logical Theory of Constructions (LTC) [41].

While ITT regards the concepts of type and elementhood as fundamental, LTC treats the concepts of proposition and truth as fundamental. There is, however, a close relationship between the two theories: from a syntactical viewpoint, they each can in principle be translated into the other, as shown by Smith [41] and by Aczel *et al.* [5]; from a semantical viewpoint, Aczel’s construction of Frege structures [3], which provide models of LTC, is closely related to Allen’s reinterpretation of ITT [6, 7]. Concerning the “untyped” view, the two theories may be regarded as based on essentially the same class of untyped computational languages. Indeed, the concept of a computation triple, which was presented by Aczel *et al.* [5] for LTC, is closely related to the concept of an evaluation rule, which was formulated by Howe [23, 24, 10, 25] for ITT and is also used in the present paper.

The approach of [37, 5] for LTC, however, considered a specific series of languages. Other possible language extensions were not discussed; nor has an effective procedure for expanding a semantics as the interpreted language extends been presented. The formulation presented here for ITT considers language extensions in general and also provides an effective procedure for expanding a semantics.

In the present paper, the class of equality prescriptions for type constructors is formulated on the basis of the theory of inductive definitions. Within the framework of “untyped” languages, Martin-Löf [32], Feferman [18, 19], Hayashi and Nakano [22], Tatsuta [43, 44], Kobayashi and Tatsuta [30], Hayashi and Kobayashi [21], and Kameyama [27] developed their theories of inductive definitions for intuitionistic predicate logic. Each of these theories, appropriately modified, would be an alternative basis for the study presented in this paper.

1.4.2. “Typed” approach

The “typed” view of ITT has also been embodied in several formulations of ITT, where a certain form of dependently typed λ -calculus has been used as the kernel of rules for types and objects. The present paper can also be compared with the work dealing with this view. In particular, it is noteworthy that the role the mechanism for inductive definitions developed in this paper plays in the “untyped” approach is similar to the role Dybjer’s inductive definitions play in the “typed” approach.

In [15, 16], Dybjer treated a more rigid variant of ITT,^e which is based on a dependently typed λ -calculus, and presented a useful general schema for introducing new inductively defined (families of) types. This schema is related to the formalization of inductive definitions by Backhouse, Chisholm, Malcolm, and Saaman [8] and by Coquand and Paulin [14]. In a recent paper [17], Dybjer and Setzer have shown an internalization of this schema.

Inductive definitions described in this paper may be more complex than Dybjer’s, since they involve untyped computational languages and refer directly to lazy evaluation relations. The present “untyped” approach, however, provides several advantages. One is that this approach can overcome the practical expressiveness problem that results from using elimination operators to codify recursive functions. (See Section 3.) Moreover, this view enables us to type programs expressed as fixed-points provided that they can be considered total functions. (See [42], for example.) And type-free equational reasoning presented in this paper may also be listed as one of the valuable features of the “untyped” approach.

Another point is that Dybjer gave a rather direct set-theoretical semantics. For example, a function type $A \rightarrow B$ was interpreted as the set of all set-theoretical functions from the set denoted by A to the set denoted by B . In other words, the domain of his interpretation was fixed and “full” from the beginning. The present paper, in contrast, treats a special kind of semantics whose domain expands naturally as the interpreted language extends. Note that a “full” semantics like Dybjer’s can, of course, be obtained by using Howe’s method of constructing a classical model of ITT [24].

2. Expression Systems

An *expression system*, a formulation of a language underlying ITT, is denoted

^eFor example, Dybjer did not consider such statements as $\text{split}(\text{apply}(\mathbf{t}, \mathbf{t}), xy.y) = 0 \in \mathbf{N}$, where $\mathbf{t} \equiv \lambda(z.(\text{apply}(z, z), 0))$. The present paper, however, takes the “untyped” view of ITT and thereby gives languages general enough for such statements to be considered valid.

by a pair $L = (C, S)$: the computational part C , called an evaluation system, determines the operational semantics for L ; the structural part S , called a prescription system, refers to C and specifies a system of equality prescriptions for type constructors.

2.1. Computational Part: Evaluation Systems

An *evaluation system* [23, 24, 10, 25]—a generalization of an untyped λ -calculus—is denoted by a quadruple $C = (K, N, \alpha, R)$: K , N , and α determine the terms of C , and R stipulates how they should be evaluated. Also presented here are two basic examples of evaluation systems: $C_\lambda = (K_\lambda, N_\lambda, \alpha_\lambda, R_\lambda)$ is the lazy λ -calculus, that is, the untyped λ -calculus with lazy evaluation; $C_0 = (K_0, N_0, \alpha_0, R_0)$ is the “initial” evaluation system containing all the operators presented explicitly in [33, 34]. (Although the version of ITT treated in the present paper is essentially the same as that in [33, 34], the notation used to describe it is mainly that in [38].)

2.1.1. Canonical operators, noncanonical operators, and their arities

A set of *operators* is specified as the union of two disjoint sets: a set K of *canonical* operators and a set N of *noncanonical* operators. A function α from $K \cup N$ to $\{(k_1, \dots, k_n) \mid n, k_i \geq 0\}$ provides each operator ρ with its *arity* $\alpha(\rho)$, which specifies the number and binding structure of the operator’s arguments.

Example 2.1 The syntax of C_λ is determined by $K_\lambda = \{\lambda\}$, $N_\lambda = \{\mathbf{apply}\}$, $\alpha_\lambda(\lambda) = (1)$, and $\alpha_\lambda(\mathbf{apply}) = (0, 0)$. And the constituents K_0 , N_0 , and α_0 of the initial evaluation system C_0 are as follows:

$$K_0 = \left\{ \begin{array}{l} \mathbf{N}_n, m_n \ (n \geq 0 \text{ and } m \in \{0, 1, \dots, n-1\}), \mathbf{N}, 0, \mathbf{succ}, \mathbf{List}, \mathbf{nil}, \mathbf{cons}, \\ \mathbf{Eq}, \mathbf{eq}, +, \mathbf{inl}, \mathbf{inr}, \mathbf{\Pi}, \lambda, \mathbf{\Sigma}, \langle \cdot, \cdot \rangle, \mathbf{W}, \mathbf{sup}, \mathbf{U}_n \ (n \geq 0) \end{array} \right\},$$

$$N_0 = \{ \mathbf{case}_n \ (n \geq 0), \mathbf{natrec}, \mathbf{listrec}, \mathbf{judge}, \mathbf{when}, \mathbf{apply}, \mathbf{eapply}, \mathbf{split}, \mathbf{wrec} \},$$

$$\alpha_0(\rho) = \left\{ \begin{array}{ll} () & \text{if } \rho \text{ is } \mathbf{N}_n, m_n, \mathbf{N}, 0, \mathbf{nil}, \mathbf{eq}, \mathbf{U}_n, \\ (0) & \text{if } \rho \text{ is } \mathbf{succ}, \mathbf{List}, \mathbf{inl}, \mathbf{inr}, \\ (1) & \text{if } \rho \text{ is } \lambda, \\ (0, 0) & \text{if } \rho \text{ is } \mathbf{cons}, \mathbf{apply}, \mathbf{eapply}, \langle \cdot, \cdot \rangle, \mathbf{judge}, +, \mathbf{sup}, \\ (0, 1) & \text{if } \rho \text{ is } \mathbf{\Pi}, \mathbf{\Sigma}, \mathbf{W}, \\ (0, 2) & \text{if } \rho \text{ is } \mathbf{split}, \\ (0, 3) & \text{if } \rho \text{ is } \mathbf{natrec}, \\ (0, 0, 0) & \text{if } \rho \text{ is } \mathbf{Eq}, \\ (0, 0, 2) & \text{if } \rho \text{ is } \mathbf{natrec}, \\ (0, 0, 3) & \text{if } \rho \text{ is } \mathbf{listrec}, \\ (0, 1, 1) & \text{if } \rho \text{ is } \mathbf{when}, \\ (0, 0, \underbrace{\dots}_n, 0) & \text{if } \rho \text{ is } \mathbf{case}_n. \end{array} \right.$$

The language described below provides a systematic notation for expressions involving binding structure. The reader can simply write, for example, $\lambda(b)$, $a(f, g)$,

and $\text{natrec}(c, d, e)$ instead of $\lambda x.b(x)$, $a(f/x, g/y)$, and $\text{natrec}(c, d, xy.e(x, y))$.

For each $m \geq 0$, fix an infinite set of variables called the *variables of arity m* . A variable of arity m is called a *variable* if $m = 0$; otherwise it is called a *second-order variable*. The *term schemas of arity m* are inductively defined as follows:

- (i) a variable of arity m is a term schema of arity m ;
- (ii) if ρ is an operator with arity (k_1, \dots, k_n) and if c_1, \dots, c_n are term schemas of the respective arities k_1, \dots, k_n , then $\rho(c_1, \dots, c_n)$ is a term schema of arity 0;
- (iii) if b is a term schema of arity n and if all a_1, \dots, a_n are term schemas of arity 0, then $b(a_1, \dots, a_n)$ is a term schema of arity 0;
- (iv) if b is a term schema of arity 0 and if \bar{x} is a sequence of m distinct variables, then $\bar{x}.b$ is a term schema of arity m .

In particular, the *term schemas* are the term schemas of arity 0. The *terms of arity m* are the term schemas of arity m containing no second-order variables; in particular, the *terms* are the terms of arity 0. A *simple term schema* is a term schema of the form $\rho(\bar{c})$, where ρ is an operator and \bar{c} is a list of distinct variables of appropriate arities. A term or a simple term schema of the form $\rho(\bar{c})$ is *canonical* if ρ is a canonical operator; otherwise it is *noncanonical*.

Binding structure can be added by specifying that in a term schema $x_1, \dots, x_m.b$ of arity m , each x_i binds in b . (This means that each occurrence of a second-order variable is always free.) The set of free variables and second-order variables in a term schema a of each arity is denoted by $\mathbb{V}(a)$. Term schemas of each arity are assumed to be identical up to the renaming of bound variables. Moreover, if $x_1, \dots, x_m.b$ is a term schema of arity m and if a_1, \dots, a_m is a list of term schemas, then $(x_1, \dots, x_m.b)(a_1, \dots, a_m)$ is identified with the result of substituting in the term schema b the term schemas a_1, \dots, a_m for free variables x_1, \dots, x_m .

The set of closed terms of arity m is denoted by \mathbb{T}_C^m . In particular, the set of closed terms is denoted by \mathbb{T}_C . Each closed term is regarded as a “program” to be evaluated if it is noncanonical; otherwise it is viewed as a “value” of a program. A *valuation* is a map that assigns an arbitrary element of \mathbb{T}_C^m to each variable of arity m . The domain of a valuation can of course be extended so that every term schema of arity m can be assigned an element of \mathbb{T}_C^m .

In this paper, a, b, c, \dots are usually syntactical variables that vary through term schemas of appropriate arities; s, t, u, \dots are usually syntactical variables that vary through closed terms of appropriate arities; and x, y, z, \dots are usually syntactical variables that vary through variables. These notational conveniences, however, are not adhered to strictly.

2.1.2. Evaluation rules

Let H be a set of variables of any arities. An expression of the form

$$a_1 \Downarrow b_1 \ \& \ \dots \ \& \ a_n \Downarrow b_n$$

($n \geq 1$) is *sequential in H* if it satisfies the following conditions:

- (i) each a_i ($1 \leq i \leq n$) is a term schema such that $\mathbb{V}(a_i) \subset H \cup \bigcup_{j < i} \mathbb{V}(b_j)$;
- (ii) each b_i ($1 \leq i \leq n$) is a canonical simple term schema or a variable such that $\mathbb{V}(b_i)$ and $H \cup \bigcup_{j < i} \mathbb{V}(b_j)$ are disjoint.

The concept of sequentiality plays a key role throughout the present paper.

Consider a set R of *evaluation rules*, each having the form

$$a \Downarrow b \blacktriangleleft a_1 \Downarrow b_1 \& \cdots \& a_n \Downarrow b_n$$

($n \geq 1$)^f and satisfying the following conditions:

- (i) a is a noncanonical simple term schema and b is a variable;
- (ii) $a_1 \Downarrow b_1 \& \cdots \& a_n \Downarrow b_n$ is sequential in $\mathbb{V}(a)$ and b_n is the same as b .

Example 2.2 The evaluation rule of the lazy λ -calculus C_λ is given by $R_\lambda = \{\text{apply}(c, a) \Downarrow e \blacktriangleleft c \Downarrow \lambda(b) \& b(a) \Downarrow e\}$. And the constituent R_0 of the initial evaluation system C_0 is

$$\left\{ \begin{array}{l} \text{case}_n(c, c_0, \dots, c_{n-1}) \Downarrow d \blacktriangleleft c \Downarrow m_n \& c_m \Downarrow d \\ \quad (n \geq 0 \text{ and } m \in \{0, 1, \dots, n-1\}), \\ \text{natrec}(c, d, e) \Downarrow f \blacktriangleleft c \Downarrow 0 \& d \Downarrow f, \\ \text{natrec}(c, d, e) \Downarrow f \blacktriangleleft c \Downarrow \text{succ}(a) \& e(a, \text{natrec}(a, d, e)) \Downarrow f, \\ \text{listrec}(c, d, e) \Downarrow f \blacktriangleleft c \Downarrow \text{nil} \& d \Downarrow f, \\ \text{listrec}(c, d, e) \Downarrow f \blacktriangleleft c \Downarrow \text{cons}(a, b) \& e(a, b, \text{listrec}(b, d, e)) \Downarrow f, \\ \text{judge}(c, d) \Downarrow e \blacktriangleleft c \Downarrow \text{eq} \& d \Downarrow e, \\ \text{when}(c, d, e) \Downarrow f \blacktriangleleft c \Downarrow \text{inl}(a) \& d(a) \Downarrow f, \\ \text{when}(c, d, e) \Downarrow f \blacktriangleleft c \Downarrow \text{inr}(b) \& e(b) \Downarrow f, \\ \text{apply}(c, a) \Downarrow e \blacktriangleleft c \Downarrow \lambda(b) \& b(a) \Downarrow e, \\ \text{eapply}(c, a) \Downarrow e \blacktriangleleft c \Downarrow \lambda(b) \& a \Downarrow d \& b(d) \Downarrow e, \\ \text{split}(c, d) \Downarrow e \blacktriangleleft c \Downarrow \langle a, b \rangle \& d(a, b) \Downarrow e, \\ \text{wrec}(c, d) \Downarrow e \blacktriangleleft c \Downarrow \text{sup}(a, b) \& b \Downarrow \lambda(g) \\ \quad \& d(a, \lambda(g), \lambda(x.\text{wrec}(g(x), d))) \Downarrow e \end{array} \right\}.$$

The operator `eapply`, which was not presented in [33, 34], is used here for “eager” functional application.

Let R^+ be $R \cup \{a \Downarrow a \mid a \text{ is a canonical simple term schema}\}$. The *evaluation relation* $\Downarrow_C \subset \mathbb{T}_C \times \mathbb{T}_C$ is defined by the following inductive definition: if V is a valuation, if $a \Downarrow b \blacktriangleleft a_1 \Downarrow b_1 \& \cdots \& a_n \Downarrow b_n$ is in R^+ , and if $V(a_i) \Downarrow_C V(b_i)$ for every i , then $V(a) \Downarrow_C V(b)$. It immediately follows that the evaluation considered here is *lazy* in the following sense:

- (i) if $t \Downarrow_C s$, then s is canonical;
- (ii) if s is a closed canonical term, then $s \Downarrow_C u$ if and only if $s \equiv u$.

2.1.3. Determinism

^fThe number n of premises of an evaluation rule can, in general, be an arbitrary ordinal number. This generalization is essential to the construction of a classical model of ITT [24].

In this paper, hereafter, we assume that every evaluation system C is *deterministic*; that is, $t \Downarrow_C s$ and $t \Downarrow_C u$ imply $s \equiv u$ for every t , s , and u . The initial evaluation system C_0 , for example, is deterministic; induction on elements in \Downarrow_{C_0} shows that for every t and s such that $t \Downarrow_{C_0} s$, $t \Downarrow_{C_0} u$ implies $s \equiv u$ for every u .

The determinism of the computational languages underlying ITT has been assumed either explicitly or implicitly in several studies [6, 7, 24, 20, 47] as well as in the original account of the theory by Martin-Löf himself [33, 34]. This is because ITT could fall into inconsistency if nondeterminism were introduced into ITT's computational languages and we required ITT's judgemental equality to be "extensional" in the sense that $a = b \in A$ (or $A = B$) means that the "values" of a and b (or A and B) are equal. Consider, for example, what would happen if McCarthy's `amb`-operator [36] were introduced with the following evaluation rules: $\text{amb}(c, d) \Downarrow e \blacktriangleleft c \Downarrow e$ and $\text{amb}(c, d) \Downarrow e \blacktriangleleft d \Downarrow e$. The statements $\text{amb}(0, \text{succ}(0)) = 0 \in \mathbb{N}$ and $\text{amb}(0, \text{succ}(0)) = \text{succ}(0) \in \mathbb{N}$ would be considered valid because $\text{amb}(0, \text{succ}(0))$ has 0 and $\text{succ}(0)$ as its values; however, the validity of these statements would imply by symmetricity and transitivity that $0 = \text{succ}(0) \in \mathbb{N}$.

2.1.4. Bisimulation-like equivalences

The most explicit benefit from following Howe's definition of evaluation systems is proof of the fact that a certain bisimulation-like program equivalence becomes a congruence [23, 24, 25]. This congruence is essential in presenting several results in the following sections.

A notational convenience is introduced here: if ϕ is a binary relation in \mathbb{T}_C and if \bar{s} and \bar{s}' are lists s_1, \dots, s_n and s'_1, \dots, s'_n of closed terms of the respective arities k_1, \dots, k_n , then $\phi(\bar{s}, \bar{s}')$ means that $\phi(s_i(u_1, \dots, u_{k_i}), s'_i(u_1, \dots, u_{k_i}))$ for every i and for every list u_1, \dots, u_{k_i} of closed terms.

For each binary relation ϕ in \mathbb{T}_C , define $[\phi]_C$ as follows: for each t and t' in \mathbb{T}_C , $[\phi]_C(t, t')$ means that whenever $t \Downarrow_C \theta(\bar{s})$ for some canonical $\theta(\bar{s})$ in \mathbb{T}_C , there exists $\theta(\bar{s}')$ such that $t' \Downarrow_C \theta(\bar{s}')$ and $\phi(\bar{s}, \bar{s}')$. Let \leq_C be the largest binary relation ϕ in \mathbb{T}_C such that $\phi \subset [\phi]_C$. Since $[\cdot]_C$ is monotone in the sense that $\phi \subset \phi'$ implies $[\phi]_C \subset [\phi']_C$, the theory of coinductive definitions [4, 9] guarantees the existence of such \leq_C . It readily follows that \leq_C is reflexive and transitive. The sequentiality in evaluation rules plays a vital role in proving the following theorem.

Theorem 2.3 ([23, 24, 25]) *The preorder \leq_C is a precongruence; that is, $\bar{s} \leq_C \bar{s}'$ implies $\rho(\bar{s}) \leq_C \rho(\bar{s}')$ for each operator ρ . More generally, $t \leq_C t'$ and $\bar{s} \leq_C \bar{s}'$ imply $t(\bar{s}) \leq_C t'(\bar{s}')$.[§]*

Proof. (Informal Sketch) Define the *precongruence candidate* $\widehat{\leq}_C$ for \leq_C as follows: $t \widehat{\leq}_C t'$ means that t' can be obtained from t via one bottom-up pass of replacements of subterms by terms that are larger under \leq_C . Then, by definition, $\widehat{\leq}_C$ is operator-respecting and contains \leq_C . The sequentiality in evaluation rules

[§]In fact, Howe proved this theorem without assuming the determinism of C .

is used to show that $\widehat{\leq}_C \subset [\widehat{\leq}_C]_C$. This implies $\widehat{\leq}_C \subset \leq_C$ by coinduction principle. Hence, $\widehat{\leq}_C = \leq_C$. \square

The bisimulation-like equivalence \sim_C , which is featured in this paper, is defined as $\leq_C \cap \geq_C$. It can be viewed as an observational indistinguishability between two terms of C . By Theorem 2.3, the equivalence \sim_C becomes a *congruence*; that is, $t \sim_C t'$ and $\bar{s} \sim_C \bar{s}'$ imply $t(\bar{s}) \sim_C t'(\bar{s}')$. It should be noted that, by virtue of the determinism of C , if $t \Downarrow_C s$, then $t \sim_C t'$ if and only if $s \sim_C t'$.

2.2. Structural Part: Prescription Systems

According to Martin-Löf, a canonical type is defined by prescribing how a canonical object of the type is formed as well as how two equal canonical objects of the type are formed [33, 34]. A mathematical interpretation of such “prescription” is presented below.

The following is a sketch of a standard approach using partial equivalence relations (PERs). For each evaluation system C , let \mathbb{P}_C be the set of evaluation-preserving PERs in \mathbb{T}_C . In other words, \mathbb{P}_C is the set of symmetric and transitive $\phi \subset \mathbb{T}_C \times \mathbb{T}_C$ such that $\forall t \forall t'. (\phi(t, t') \Leftrightarrow \exists s. (t \Downarrow_C s \wedge \phi(s, t')))$. Consider the case for Π -like type constructors having families of types as their arguments. For each type constructor Δ , the *equality prescription for Δ* is the operation $[\Delta]$ that associates each C with a function $[\Delta]_C$ that assigns a member of \mathbb{P}_C to each pair of χ and ψ such that $\text{Fam}_C(\chi, \psi)$. Here $\text{Fam}_C(\chi, \psi)$ means that $\chi \in \mathbb{P}_C$ and $\psi \in \{s \mid \chi(s, s)\} \rightarrow \mathbb{P}_C$ such that $\forall s \forall s'. (\chi(s, s') \Rightarrow \psi(s) = \psi(s'))$. Given a set of equality prescriptions and an arbitrary C , we can use Allen’s methods [6] to inductively construct a semantics of ITT over C . Specifically, if ϕ_A and ϕ_B being $\text{Fam}_C(\phi_A, \phi_B)$ are assigned to terms A and B of the respective arities 0 and 1 at some stage in the construction, then at the next stage $[\Delta]_C(\phi_A, \phi_B)$ is assigned to $\Delta(A, B)$ as its denotation.

The approach outlined above can be extended to cover all the type constructors presented explicitly in [33, 34]. Indeed, the equality prescriptions for the basic type constructors \mathbb{N}_n , \mathbb{N} , List , Eq , $+$, Π , Σ , and \mathbb{W} can be inductively defined as follows. (Also see similar type definitions in the literature [11, 41, 6, 5].)

$$\begin{aligned}
& [\mathbb{N}_n]_C(t, t') \\
& \Leftrightarrow t \Downarrow_C 0_n \wedge t' \Downarrow_C 0_n \\
& \quad \vee \dots \vee \\
& \quad t \Downarrow_C n-1_n \wedge t' \Downarrow_C n-1_n, \\
& [\mathbb{N}]_C(t, t') \\
& \Leftrightarrow t \Downarrow_C 0 \wedge t' \Downarrow_C 0 \\
& \quad \vee \\
& \quad \exists s \exists s'. t \Downarrow_C \text{succ}(s) \wedge t' \Downarrow_C \text{succ}(s') \wedge [\mathbb{N}]_C(s, s'), \\
& [\text{List}]_C(\chi)(t, t') \\
& \Leftrightarrow t \Downarrow_C \text{nil} \wedge t' \Downarrow_C \text{nil} \\
& \quad \vee \\
& \quad \exists s u \exists s' u'. t \Downarrow_C \text{cons}(s, u) \wedge t' \Downarrow_C \text{cons}(s', u') \\
& \quad \quad \wedge \chi(s, s') \wedge [\text{List}]_C(u, u'),
\end{aligned}$$

$$\begin{aligned}
& \llbracket \text{Eq} \rrbracket_C(\chi, f_1, f_2)(t, t') \\
& \quad \Leftrightarrow t \downarrow_C \text{eq} \wedge t' \downarrow_C \text{eq} \wedge \chi(f_1, f_2), \\
& \llbracket + \rrbracket_C(\chi_1, \chi_2)(t, t') \\
& \quad \Leftrightarrow \exists a \exists a'. t \downarrow_C \text{inl}(a) \wedge t' \downarrow_C \text{inl}(a') \wedge \chi_1(a, a') \\
& \quad \vee \\
& \quad \exists b \exists b'. t \downarrow_C \text{inl}(b) \wedge t' \downarrow_C \text{inl}(b') \wedge \chi_2(b, b'), \\
& \llbracket \Pi \rrbracket_C(\chi, \psi)(t, t') \\
& \quad \Leftrightarrow \exists w \exists w'. t \downarrow_C \lambda(w) \wedge t' \downarrow_C \lambda(w') \\
& \quad \quad \wedge \forall s \forall s'. \chi(s, s') \Rightarrow \psi(s)(w(s), w'(s')), \\
& \llbracket \Sigma \rrbracket_C(\chi, \psi)(t, t') \\
& \quad \Leftrightarrow \exists s u \exists s' u'. t \downarrow_C \langle s, u \rangle \wedge t' \downarrow_C \langle s', u' \rangle \\
& \quad \quad \wedge \chi(s, s') \\
& \quad \quad \wedge \chi(s, s') \Rightarrow \psi(s)(u, u'), \\
& \llbracket \mathbb{W} \rrbracket_C(\chi, \psi)(t, t') \\
& \quad \Leftrightarrow \exists s u \exists s' u'. t \downarrow_C \text{sup}(s, u) \wedge t' \downarrow_C \text{sup}(s', u') \\
& \quad \quad \wedge \exists w \exists w'. u \downarrow_C \lambda(w) \wedge u' \downarrow_C \lambda(w') \\
& \quad \quad \quad \wedge \chi(s, s') \\
& \quad \quad \quad \wedge \chi(s, s') \Rightarrow \forall v \forall v'. \psi(s)(v, v') \\
& \quad \quad \quad \Rightarrow \llbracket \mathbb{W} \rrbracket_C(\chi, \psi)(w(v), w'(v')).
\end{aligned}$$

This paper extracts a general pattern from these definitions of specific equality prescriptions and presents a certain schema for treating them uniformly. This schema is original in that it is derived by focusing on the role that the concept of sequentiality plays in the inductive definitions of equality prescriptions.

Let $C = (K, N, \alpha, R)$ be an arbitrary evaluation system. A *prescription system* on C is denoted by a triple $S = (D, \kappa, \epsilon)$: D fixes a set of special canonical operators called type constructors; for each type constructor Δ , $\kappa(\Delta)$ specifies the number and structure of arguments with which Δ can form canonical types; and the equality prescription $\llbracket \Delta \rrbracket$ for Δ is generated from $\epsilon(\Delta)$. Also presented here is a basic example of a prescription system: the “initial” prescription system $S_0 = (D_0, \kappa_0, \epsilon_0)$ is defined on the initial evaluation system C_0 and contains all the type constructors presented explicitly in [33, 34].

2.2.1. Type constructors and their kinds

In general, every type constructor in ITT may be regarded as a map that assigns a new canonical type to each bundle of “families of types” and associated “functions.” Consider, for example, the following formation rule:

$$\frac{
\begin{array}{l}
A \text{ type} \\
B(x) \text{ type } (x \in A) \\
b_1(x) \in B(x) \text{ } (x \in A) \\
b_2(x) \in B(x) \text{ } (x \in A) \quad D \text{ type} \\
C(x, y) \text{ type } (x \in A, y \in B(x)) \quad d \in D
\end{array}
}{
\Delta(A, B, b_1, b_2, C, D, d) \text{ type.}
}$$

The type constructor Δ can be regarded as a map that assigns a new canonical type to each bundle having two components such that (i) the first component of

the bundle has three layers of families of types whose second layer is accompanied by two unary functions and (ii) the second component consists of a single type accompanied by a 0-ary function. From the domains of such maps, the concept of “kind” can be abstracted. For example, the kind of Δ will be given as $12\Box\Box31\Box$.

A set of *type constructors* is specified as $D \subset K$. Let \mathbb{K} be the set

$$\left(\bigcup_{m \geq 1} \{1\} \{\Box\}^* \{2\} \{\Box\}^* \{3\} \{\Box\}^* \cdots \{m\} \{\Box\}^* \right)^*$$

of strings. (When S and S' are sets of strings, the concatenation of S and S' is denoted by SS' and the Kleene closure of S is denoted by S^* .) A function κ from D to \mathbb{K} provides each $\Delta \in D$ with its *kind* $\kappa(\Delta) \in \mathbb{K}$.

Example 2.4 The constituents D_0 and κ_0 of the initial prescription system S_0 are as follows:

$$D_0 = \{\mathbf{N}_n \ (n \geq 0), \mathbf{N}, \text{List}, \text{Eq}, +, \Pi, \Sigma, \mathbf{W}\},$$

$$\kappa_0(\Delta) = \begin{cases} \varepsilon & \text{if } \Delta \text{ is } \mathbf{N}_n, \mathbf{N}, \\ 1 & \text{if } \Delta \text{ is List}, \\ 1\Box\Box & \text{if } \Delta \text{ is Eq}, \\ 11 & \text{if } \Delta \text{ is } +, \\ 12 & \text{if } \Delta \text{ is } \Pi, \Sigma, \mathbf{W}. \end{cases}$$

The following are basic operations associated with kinds: If $\xi \in \mathbb{K}$ has the form

$$\underbrace{1\Box\cdots\Box}_{k_{1,1}\Box\text{'s}} \cdots \underbrace{\Box\cdots\Box}_{k_{1,m_1}\Box\text{'s}} \cdots \underbrace{1\Box\cdots\Box}_{k_{n,1}\Box\text{'s}} \cdots \underbrace{\Box\cdots\Box}_{k_{n,m_n}\Box\text{'s}},$$

then

$$|\xi| = n,$$

$$\xi_i = m_i$$

for each i such that $1 \leq i \leq n$,

$$\xi_{i,j} = k_{i,j}$$

for each i and j such that $1 \leq i \leq n$ and $1 \leq j \leq m_i$, and $\beta(\xi)$ is

$$\underbrace{(0, \dots, 0)}_{k_{1,1}0\text{'s}}, \dots, \underbrace{m_1-1, m_1-1, \dots, m_1-1}_{k_{1,m_1}m_1-1\text{'s}}, \dots, \underbrace{0, 0, \dots, 0}_{k_{n,1}0\text{'s}}, \dots, \underbrace{m_n-1, m_n-1, \dots, m_n-1}_{k_{n,m_n}m_n-1\text{'s}}.$$

In terms of these operations, every type constructor Δ can be regarded as a map that assigns a new canonical type to each bundle of the following form:

- the bundle consists of $|\kappa(\Delta)|$ components;
- for each i such that $1 \leq i \leq |\kappa(\Delta)|$, the i -th component of the bundle has $\kappa(\Delta)_i$ layers of families of types;
- for each i and j such that $1 \leq i \leq |\kappa(\Delta)|$ and $1 \leq j \leq \kappa(\Delta)_i$, the j -th layer family of types in the i -th component of the bundle is accompanied by $\kappa(\Delta)_{i,j}$ functions.

For the sake of concise presentation, the concept of a bundle is introduced formally. For each $\xi \in \mathbb{K}$, a ξ -*bundle* is a sequence of the form

$$(T_{i,j}, t_{i,j,k})_{1 \leq i \leq |\xi|, 1 \leq j \leq \xi_i, 1 \leq k \leq \xi_{i,j}}$$

such that all $T_{i,j}, t_{i,j,k}$ are closed terms of arity $j-1$. Each type constructor Δ will take a $\kappa(\Delta)$ -bundle Γ as its arguments in order to constitute a canonical type $\Delta(\Gamma)$. To guarantee this, we impose on κ the condition that $\beta(\kappa(\Delta))$ should be equal to $\alpha(\Delta)$ for every $\Delta \in D$.

Each ξ -bundle is interpreted as a ξ -*structure*, which is a sequence

$$(\phi_{i,j}, \eta_{i,j,k})_{1 \leq i \leq |\xi|, 1 \leq j \leq \xi_i, 1 \leq k \leq \xi_{i,j}}$$

such that

$$\begin{aligned} \phi_{i,j} &\in \prod_{s_1 \in |\phi_{i,1}|} \cdots \prod_{s_{j-1} \in |\phi_{i,j-1}(s_1, s_2, \dots, s_{j-2})|} \mathbb{P}_C \quad \text{and} \\ \eta_{i,j,k} &\in \prod_{s_1 \in |\phi_{i,1}|} \cdots \prod_{s_{j-1} \in |\phi_{i,j-1}(s_1, s_2, \dots, s_{j-2})|} |\phi_{i,j}(s_1, s_2, \dots, s_{j-1})|. \end{aligned}$$

(For each $\phi \in \mathbb{P}_C$, the set $\{s \mid \phi(s, s)\}$ is denoted by $|\phi|$.) Moreover, each ξ -structure must be *extensional* [33, 34] in the sense that for every i, j , and k ,

$$\begin{aligned} \phi_{i,j}(s_1, \dots, s_{j-1}) &= \phi_{i,j}(s'_1, \dots, s'_{j-1}) \quad \text{and} \\ \phi_{i,j}(s_1, \dots, s_{j-1}) &(\eta_{i,j,k}(s_1, \dots, s_{j-1}), \eta_{i,j,k}(s'_1, \dots, s'_{j-1})) \end{aligned}$$

whenever $\phi_{i,1}(s_1, s'_1), \dots, \phi_{i,j-1}(s_1, s_2, \dots, s_{j-2})(s_{j-1}, s'_{j-1})$.

2.2.2. Equality prescriptions for type constructors

Consider the following definition of an equality prescription:

$$\begin{aligned} \llbracket \mathbf{N}_* \rrbracket_C(t, t') & \\ \Leftrightarrow \exists n m \exists n' m'. & t \Downarrow_C [n, m] \wedge t' \Downarrow_C [n', m'] \\ & \wedge \exists a \exists a'. n \Downarrow_C \text{succ}(a) \wedge n' \Downarrow_C \text{succ}(a') \\ & \wedge m \Downarrow_C 0 \wedge m' \Downarrow_C 0 \\ & \wedge a \Downarrow_C 0 \wedge a' \Downarrow_C 0 \\ & \vee \\ & \exists l \exists l'. a \Downarrow_C \text{succ}(l) \wedge a' \Downarrow_C \text{succ}(l') \\ & \wedge \llbracket \mathbf{N}_* \rrbracket_C([\text{succ}(l), 0], [\text{succ}(l'), 0]) \\ & \vee \\ & \exists b \exists b'. m \Downarrow_C \text{succ}(b) \wedge m' \Downarrow_C \text{succ}(b') \\ & \wedge \llbracket \mathbf{N}_* \rrbracket_C([a, b], [a', b']). \end{aligned}$$

This is the inductive definition of the equality prescription for \mathbf{N}_* , which is an integrated form of the finite types $\mathbf{N}_n (n \geq 0)$. Intuitively, $[n, m] \in \mathbf{N}_*$ is equivalent to $m_n \in \mathbf{N}_n$. This example is an adaptation of the material that was treated by Dybjer [15] as an example of an inductively defined family of types.

An observation about this example is that the definition of $\llbracket \mathbf{N}_* \rrbracket$ can be decomposed into two parts. First, the following three expressions are implicitly used in the definition of $\llbracket \mathbf{N}_* \rrbracket$ to determine the “operational” property of each t and t' :

$$\begin{aligned} q_1 &\equiv c \Downarrow [n, m] \ \& \ n \Downarrow \text{succ}(a) \ \& \ m \Downarrow 0 \ \& \ a \Downarrow 0; \\ q_2 &\equiv c \Downarrow [n, m] \ \& \ n \Downarrow \text{succ}(a) \ \& \ m \Downarrow 0 \ \& \ a \Downarrow \text{succ}(l); \\ q_3 &\equiv c \Downarrow [n, m] \ \& \ n \Downarrow \text{succ}(a) \ \& \ m \Downarrow \text{succ}(b). \end{aligned}$$

Second, the following three formulae corresponding to q_1 , q_2 , and q_3 are used to determine the “logical” property of each t and t' :

$$\begin{aligned} A_{q_1}[X, n, m, a, n', m', a'] &\equiv \top; \\ A_{q_2}[X, n, m, a, l, n', m', a', l'] &\equiv X([\text{succ}(l), 0], [\text{succ}(l'), 0]); \\ A_{q_3}[X, n, m, a, b, n', m', a', b'] &\equiv X([a, b], [a', b']). \end{aligned}$$

From these, $\llbracket \mathbf{N}_* \rrbracket$ can be immediately reconstructed.

This observation leads to the definition of the constituent ϵ of a prescription system $S = (D, \kappa, \epsilon)$: ϵ is a function that assigns to each $\Delta \in D$ a pair $(Q, \{A_q\}_{q \in Q})$, where Q is called a “selector” and A_q a “ $\kappa(\Delta)$ -formula” for q .

Selectors. A *selector* is a set Q of expressions, sequential in $\{a_1\}$, of the form

$$a_1 \Downarrow b_1 \& \cdots \& a_n \Downarrow b_n$$

($n \geq 1$). (This definition implies that a_1 should be a variable.) Also, assume that two members, $a_1 \Downarrow b_1 \& \cdots \& a_n \Downarrow b_n$ and $a'_1 \Downarrow b'_1 \& \cdots \& a'_{n'} \Downarrow b'_{n'}$, of Q are identical unless there exists i ($1 \leq i \leq n$ and $1 \leq i \leq n'$) such that (i) $a_1, b_1, a_2, b_2, \dots, a_i$ are respectively the same as $a'_1, b'_1, a'_2, b'_2, \dots, a'_i$ and (ii) b_i and b'_i are simple term schemas having different canonical operators. This condition means that Q should constitute a tree with a special form. The above expressions q_1 , q_2 , and q_3 , for example, make up a selector.

When $q \in Q$ is of the form $a_1 \Downarrow b_1 \& \cdots \& a_n \Downarrow b_n$, the set $\bigcup_i \mathbb{V}(b_i)$ is denoted by $\mathbb{V}(q)$. A distinct set obtained from $\mathbb{V}(q)$ by renaming each element in it is fixed and denoted by $\mathbb{V}'(q)$; $\#_q$ is an arity-preserving bijective map from $\mathbb{V}'(q)$ to $\mathbb{V}(q)$. If t is a closed term and V is a valuation, then the t -interpretation ${}^t\llbracket q \rrbracket_C^V$ of q under V is that $t \equiv V(a_1)$ and

$$V(a_1) \Downarrow_C V(b_1) \wedge \cdots \wedge V(a_n) \Downarrow_C V(b_n).$$

ξ -formulae. For each $\xi \in \mathbb{K}$, fix a set of special variables $f_{i,j,k}$ of arity $j - 1$ such that $1 \leq i \leq |\xi|$, $1 \leq j \leq \xi_i$, and $1 \leq k \leq \xi_{i,j}$. These variables are treated here as “function symbols” of the corresponding arities. Then the ξ -terms are inductively defined as follows:

- (i) a term schema not containing any $f_{i,j,k}$ is a ξ -term;
- (ii) if a_1, \dots, a_{j-1} are ξ -terms, then $f_{i,j,k}(a_1, \dots, a_{j-1})$ is a ξ -term.

Also fix a binary predicate symbol X and a set of $(j + 1)$ -ary predicate symbols $P_{i,j}$ such that $1 \leq i \leq |\xi|$ and $1 \leq j \leq \xi_i$. Then the ξ -formulae are inductively defined as follows:

- (i) if a and a' are ξ -terms, then $X(a, a')$ is a ξ -formula;
- (ii) if $a_1, \dots, a_{j-1}, a, a'$ are ξ -terms, then $P_{i,j}(a_1, \dots, a_{j-1}, a, a')$ is a ξ -formula;
- (iii) if A and B are ξ -formulae, then \top , \perp , $A \wedge B$, $A \vee B$, and $A \Rightarrow B$ are also ξ -formulae;
- (iv) if v is a variable and if A is a ξ -formula, then $\forall v.A$ and $\exists v.A$ are ξ -formulae.

Let V be a valuation, let

$$\Phi = (\phi_{i,j}, \eta_{i,j,k})_{1 \leq i \leq |\xi|, 1 \leq j \leq \xi_i, 1 \leq k \leq \xi_{i,j}}$$

be a ξ -structure, and let ϕ be a binary relation in \mathbb{T}_C . Then the *interpretation* $\llbracket a \rrbracket_C^V(\Phi)$ of a ξ -term a under V and Φ is inductively defined as follows:

$$\begin{aligned} \llbracket e \rrbracket_C^V(\Phi) &= V(e) \quad \text{if } e \text{ does not contain any } f_{i,j,k}; \\ \llbracket f_{i,j,k}(a_1, \dots, a_{j-1}) \rrbracket_C^V(\Phi) &= \eta_{i,j,k}(\llbracket a_1 \rrbracket_C^V(\Phi), \dots, \llbracket a_{j-1} \rrbracket_C^V(\Phi)). \end{aligned}$$

Thus, to make $\llbracket a \rrbracket_C^V(\Phi)$ *well-defined*, in every application of the second clause of the above definition, $\llbracket a_1 \rrbracket_C^V(\Phi), \dots, \llbracket a_{j-1} \rrbracket_C^V(\Phi)$ must be in the domain of $\eta_{i,j,k}$. And the *interpretation* $\llbracket A \rrbracket_C^V(\Phi, \phi)$ of a ξ -formula A under V , Φ , and ϕ is inductively defined as follows:

$$\begin{aligned} \llbracket X(a, a') \rrbracket_C^V(\Phi, \phi) &\Leftrightarrow \phi(\llbracket a \rrbracket_C^V(\Phi), \llbracket a' \rrbracket_C^V(\Phi)); \\ \llbracket P_{i,j}(a_1, \dots, a_{j-1}, a, a') \rrbracket_C^V(\Phi, \phi) &\Leftrightarrow \phi_{i,j}(\llbracket a_1 \rrbracket_C^V(\Phi), \dots, \llbracket a_{j-1} \rrbracket_C^V(\Phi), \\ &\quad \llbracket a \rrbracket_C^V(\Phi), \llbracket a' \rrbracket_C^V(\Phi)); \\ \llbracket \top \rrbracket_C^V(\Phi, \phi) &\Leftrightarrow \top; \\ \llbracket \perp \rrbracket_C^V(\Phi, \phi) &\Leftrightarrow \perp; \\ \llbracket A \wedge B \rrbracket_C^V(\Phi, \phi) &\Leftrightarrow \llbracket A \rrbracket_C^V(\Phi, \phi) \wedge \llbracket B \rrbracket_C^V(\Phi, \phi); \\ \llbracket A \vee B \rrbracket_C^V(\Phi, \phi) &\Leftrightarrow \llbracket A \rrbracket_C^V(\Phi, \phi) \vee \llbracket B \rrbracket_C^V(\Phi, \phi); \\ \llbracket A \Rightarrow B \rrbracket_C^V(\Phi, \phi) &\Leftrightarrow \llbracket A \rrbracket_C^V(\Phi, \phi) \Rightarrow \llbracket B \rrbracket_C^V(\Phi, \phi); \\ \llbracket \forall v. A \rrbracket_C^V(\Phi, \phi) &\Leftrightarrow \llbracket A \rrbracket_{C'}^{V'}(\Phi, \phi) \text{ for every } V' \\ &\quad \text{such that } V'(h) = V(h) \text{ whenever } h \neq v; \\ \llbracket \exists v. A \rrbracket_C^V(\Phi, \phi) &\Leftrightarrow \llbracket A \rrbracket_{C'}^{V'}(\Phi, \phi) \text{ for some } V' \\ &\quad \text{such that } V'(h) = V(h) \text{ whenever } h \neq v. \end{aligned}$$

Thus, to make $\llbracket A \rrbracket_C^V(\Phi, \phi)$ *well-defined*, the interpretation of each ξ -term occurring in A must be well-defined; moreover, in every application of the second clause of the above definition, $\llbracket a_1 \rrbracket_C^V(\Phi), \dots, \llbracket a_{j-1} \rrbracket_C^V(\Phi)$ must be in the domain of $\phi_{i,j}$.

When A is a ξ -formula and q is in a selector Q , A is called a ξ -*formula for* q if its free variables and second-order variables are among those in $\{f_{i,j,k}\} \cup \mathbb{V}(q) \cup \mathbb{V}'(q)$. For example, A_{q_1} , A_{q_2} , and A_{q_3} are ε -formulae for q_1 , q_2 , and q_3 , respectively.

Example 2.5 It is straightforward to specify the constituent ϵ_0 of the initial prescription system S_0 by decomposing the definitions of $\llbracket \mathbf{N}_n \rrbracket$, $\llbracket \mathbf{N} \rrbracket$, $\llbracket \text{List} \rrbracket$, $\llbracket \text{Eq} \rrbracket$, $\llbracket + \rrbracket$, $\llbracket \Pi \rrbracket$, $\llbracket \Sigma \rrbracket$, and $\llbracket \mathbf{W} \rrbracket$ shown previously.

From a given $\epsilon(\Delta) = (Q, \{A_q\}_{q \in Q})$, the equality prescription for Δ can be immediately reconstructed. Indeed, for each $\kappa(\Delta)$ -structure Φ , $\llbracket \Delta \rrbracket_C(\Phi)$ is defined as the least binary relation ϕ in \mathbb{T}_C such that $\phi(t, t')$ is equivalent to

$$\exists q \in Q. \exists V \exists V'. \iota \llbracket q \rrbracket_C^V \wedge \iota' \llbracket q \rrbracket_C^{V'} \wedge \llbracket A_q \rrbracket_C^{V \oplus_q V'}(\Phi, \phi),$$

where $V \oplus_q V'$ is a valuation such that

$$(V \oplus_q V')(e) = \begin{cases} V(e) & \text{if } e \in \mathbb{V}(q), \\ V'(\#_q(e)) & \text{if } e \in \mathbb{V}'(q). \end{cases}$$

Actually, the conditions on Q guarantee that q and hence A_q (and also V and V' in substance) will be uniquely determined for each t and t' because the evaluation relation \Downarrow_C is deterministic. This can be rephrased as the slogan: “operational” determinism implies “logical” determinism.

Three additional conditions must be imposed on each ξ -formula A_q . (In fact, all the examples shown in this section satisfy these conditions.) First, each A_q should be *contextual* in the sense that $\llbracket A_q \rrbracket_C^V(\Phi, \phi)$ is well-defined for every V , Φ , and ϕ .

Second, to guarantee the validity of the inductive definitions, every occurrence of X in each A_q should be *strictly positive*, that is, not in the left scope of any \Rightarrow . As a result, $\phi \subset \phi'$ always implies $\llbracket A_q \rrbracket_C^V(\Phi, \phi) \subset \llbracket A_q \rrbracket_C^V(\Phi, \phi')$.

Third, each A_q should be *balanced* in the sense that

- (i) $\llbracket A_q \rrbracket_C^{V \oplus_q V'}(\Phi, \phi_{\text{Sy}})$ implies $\llbracket A_q \rrbracket_C^{V' \oplus_q V}(\Phi, \phi)$ and
- (ii) $\llbracket A_q \rrbracket_C^{V \oplus_q V'}(\Phi, \phi_{\text{Tr}})$ and $\llbracket A_q \rrbracket_C^{V' \oplus_q V''}(\Phi, \phi)$ imply $\llbracket A_q \rrbracket_C^{V \oplus_q V''}(\Phi, \phi)$

for every V , V' , V'' , Φ , and ϕ . Here ϕ_{Sy} and ϕ_{Tr} are defined respectively as $\phi_{\text{Sy}}(t, t') \Leftrightarrow \phi(t', t)$ and $\phi_{\text{Tr}}(t, t') \Leftrightarrow \forall t''. (\phi(t', t'') \Rightarrow \phi(t, t''))$. This condition, combined with “operational” (and hence “logical”) determinism, guarantees that each $\llbracket \Delta \rrbracket_C$ reconstructed from $\epsilon(\Delta)$ will be a function from the set of $\kappa(\Delta)$ -structures to \mathbb{P}_C . Indeed, if a $\kappa(\Delta)$ -structure Φ is given and if φ is ϕ_{Sy} or ϕ_{Tr} where ϕ is $\llbracket \Delta \rrbracket_C(\Phi)$, then

$$\exists q \in Q. \exists V \exists V'. {}^t \llbracket q \rrbracket_C^V \wedge {}^{t'} \llbracket q \rrbracket_C^{V'} \wedge \llbracket A_q \rrbracket_C^{V \oplus_q V'}(\Phi, \varphi)$$

implies $\varphi(t, t')$ for every t and t' . Hence $\phi \subset \phi_{\text{Sy}}$ and $\phi \subset \phi_{\text{Tr}}$.

3. Extensions of Expression Systems

Let $L = ((K, N, \alpha, R), (D, \kappa, \epsilon))$ and $L' = ((K', N', \alpha', R'), (D', \kappa', \epsilon'))$ be arbitrary expression systems. If the following conditions hold, L' is called an *extension* of L :

- $K \subset K'$;
- $N \subset N'$;
- $\alpha = \alpha' \upharpoonright K \cup N$;
- $R = \{r \in R' \mid o(r) \in N\}$,
where $o(r) = \rho$ if r has the form $\rho(\bar{c}) \Downarrow b \blacktriangleleft a_1 \Downarrow b_1 \& \cdots \& a_n \Downarrow b_n$;
- $D \subset D'$;
- $\kappa = \kappa' \upharpoonright D$;
- $\epsilon = \epsilon' \upharpoonright D$;

where $f \upharpoonright Z$ denotes the restriction of a function f to its subdomain Z . It is easy to see that the binary relation induced by the concept of extension is a partial ordering of expression systems. From a computational viewpoint, the following lemma holds.

Lemma 3.1 (Computational Conservativity) *If $L' = (C', S')$ is an extension of $L = (C, S)$, then $t \Downarrow_{C'} s$ and $t \in \mathbb{T}_C$ imply $t \Downarrow_C s$ for every t and s .*

Proof. Assume that $C' = (K', N', \alpha', R')$ and $C = (K, N, \alpha, R)$. The lemma is proved by induction on elements in $\Downarrow_{C'}$. Since it is trivial when t is canonical, assume

that for an evaluation rule $r \in R'$ of the form $\rho(\bar{c}) \Downarrow b \blacktriangleleft a_1 \Downarrow b_1 \& \cdots \& a_n \Downarrow b_n$ and for a valuation V , two terms t and s can respectively be written as $V(\rho(\bar{c}))$ and $V(b)$. Because $t \in \mathbb{T}_C$, ρ must be in N ; hence $r \in R$. Therefore it suffices to prove that $V(a_i) \Downarrow_C V(b_i)$ for every i . This follows immediately from the induction hypothesis and the sequentiality condition on r . \square

3.1. Examples of Computational Extensions

Along the lines presented above, richer languages can be obtained by building extensions of the “initial” expression system $L_0 = (C_0, S_0)$. The generality of the “untyped” approach used here can be illustrated by a few examples of extensions whose evaluation rules fall outside the traditional pattern for elimination forms.

Example 3.2 The expression system $L_{\text{even}} = ((K_{\text{even}}, N_{\text{even}}, \alpha_{\text{even}}, R_{\text{even}}), S_0)$ is the extension of L_0 with the *even*-operator. Let K_{even} and N_{even} be K_0 and $N_0 \cup \{\text{even}\}$, respectively. The arity of each operator is given by

$$\alpha_{\text{even}}(\rho) = \begin{cases} (0, 0, 0) & \text{if } \rho \text{ is even,} \\ \alpha_0(\rho) & \text{otherwise.} \end{cases}$$

The set R_{even} of evaluation rules is

$$R_0 \cup \left\{ \begin{array}{l} \text{even}(c, d, e) \Downarrow f \blacktriangleleft c \Downarrow 0 \& d \Downarrow f, \\ \text{even}(c, d, e) \Downarrow f \blacktriangleleft c \Downarrow \text{succ}(a) \& a \Downarrow 0 \& e \Downarrow f, \\ \text{even}(c, d, e) \Downarrow f \blacktriangleleft c \Downarrow \text{succ}(a) \& a \Downarrow \text{succ}(b) \& \text{even}(b, d, e) \Downarrow f \end{array} \right\}.$$

Having three evaluation rules, the *even*-operator calculates by recursion whether its argument number is even. This direct, brief characterization of the operator should be compared with the operator’s traditional definition in terms of such elimination operators as “natrec.” This example demonstrates that the present approach can overcome the practical expressiveness problem resulting from using elimination operators to codify recursive functions.

Example 3.3 The expression system $L_{\text{por}} = ((K_{\text{por}}, N_{\text{por}}, \alpha_{\text{por}}, R_{\text{por}}), S_0)$ is the extension of L_0 with a “parallel or” operator. Let K_{por} and N_{por} be K_0 and $N_0 \cup \{\text{por}\}$, respectively. The arity of each operator is given by

$$\alpha_{\text{por}}(\rho) = \begin{cases} (0, 0, 0, 0) & \text{if } \rho \text{ is por,} \\ \alpha_0(\rho) & \text{otherwise.} \end{cases}$$

The set R_{por} of evaluation rules is

$$R_0 \cup \left\{ \begin{array}{l} \text{por}(c, d, e, f) \Downarrow g \blacktriangleleft c \Downarrow \text{true} \& e \Downarrow g, \\ \text{por}(c, d, e, f) \Downarrow g \blacktriangleleft d \Downarrow \text{true} \& e \Downarrow g, \\ \text{por}(c, d, e, f) \Downarrow g \blacktriangleleft c \Downarrow \text{false} \& d \Downarrow \text{false} \& f \Downarrow g \end{array} \right\}.$$

(Note that *true*, *false*, and *Bool* are definitionally equal to 0_2 , 1_2 , and N_2 , respectively.)

Example 3.4 The expression system $L_{\text{loop}} = ((K_{\text{loop}}, N_{\text{loop}}, \alpha_{\text{loop}}, R_{\text{loop}}), S_0)$ is the extension of L_0 with the **loop**-operator. Let K_{loop} and N_{loop} be K_0 and $N_0 \cup \{\text{loop}\}$, respectively. The arity of each operator is given by

$$\alpha_{\text{loop}}(\rho) = \begin{cases} (0, 2) & \text{if } \rho \text{ is loop,} \\ \alpha_0(\rho) & \text{otherwise.} \end{cases}$$

The set R_{loop} of evaluation rules is

$$R_0 \cup \{ \text{loop}(c, d) \Downarrow e \blacktriangleleft d(c, \lambda(v.\text{loop}(v, d))) \Downarrow e \}.$$

Let $k(c)$ be definitionally equal to $\text{loop}(c, xy.\text{cons}(x, \text{apply}(y, \text{succ}(x))))$. Then for each closed term n denoting a natural number, $k(n)$ can informally be regarded as a method of generating the infinite list of increasing numbers starting with n . By virtue of the laziness of the evaluation considered here, however, it has $\text{cons}(n, \text{apply}(\lambda(v.k(v)), \text{succ}(n)))$ as its unique value. The statement $\text{car}(k(x)) = x \in \mathbf{N}$ ($x \in \mathbf{N}$), where $\text{car}(c)$ is definitionally equal to $\text{listrec}(c, \text{nil}, xyz.x)$, will hence be valid in the semantics for L_{loop} presented in Section 4.

3.2. Examples of Structural Extensions

By building extensions of the “initial” expression system L_0 , a large class of inductively defined types can also be incorporated successively. Several examples follow.

Example 3.5 The expression system $L_{\mathbf{N}_*} = ((K_{\mathbf{N}_*}, N_{\mathbf{N}_*}, \alpha_{\mathbf{N}_*}, R_{\mathbf{N}_*}), (D_{\mathbf{N}_*}, \kappa_{\mathbf{N}_*}, \epsilon_{\mathbf{N}_*}))$ is the extension of L_0 with an integrated form of the finite types \mathbf{N}_n ($n \geq 0$). This was partially described in Section 2.2.2 and is elaborated here. Let $K_{\mathbf{N}_*}$ and $N_{\mathbf{N}_*}$ be $K_0 \cup \{\mathbf{N}_*, [\cdot, \cdot]\}$ and $N_0 \cup \{\text{diagrec}\}$, respectively. The arity of each operator is given by

$$\alpha_{\mathbf{N}_*}(\rho) = \begin{cases} () & \text{if } \rho \text{ is } \mathbf{N}_*, \\ (0, 0) & \text{if } \rho \text{ is } [\cdot, \cdot], \\ (0, 0, 2, 3) & \text{if } \rho \text{ is diagrec,} \\ \alpha_0(\rho) & \text{otherwise.} \end{cases}$$

Let $D_{\mathbf{N}_*}$ be $D_0 \cup \{\mathbf{N}_*\}$. The kind of each type constructor is given by

$$\kappa_{\mathbf{N}_*}(\Delta) = \begin{cases} \varepsilon & \text{if } \Delta \text{ is } \mathbf{N}_*, \\ \kappa_0(\Delta) & \text{otherwise.} \end{cases}$$

As already mentioned in Section 2.2.2, $\epsilon_{\mathbf{N}_*}(\Delta)$ is a pair of

$$\left\{ \begin{array}{l} q_1 \equiv c \Downarrow [n, m] \ \& \ n \Downarrow \text{succ}(a) \ \& \ m \Downarrow 0 \ \& \ a \Downarrow 0, \\ q_2 \equiv c \Downarrow [n, m] \ \& \ n \Downarrow \text{succ}(a) \ \& \ m \Downarrow 0 \ \& \ a \Downarrow \text{succ}(l), \\ q_3 \equiv c \Downarrow [n, m] \ \& \ n \Downarrow \text{succ}(a) \ \& \ m \Downarrow \text{succ}(b) \end{array} \right\}$$

and

$$\left\{ \begin{array}{l} A_{q_1}[X, n, m, a, n', m', a'] \equiv \top, \\ A_{q_2}[X, n, m, a, l, n', m', a', l'] \equiv X([\text{succ}(l), 0], [\text{succ}(l'), 0]), \\ A_{q_3}[X, n, m, a, b, n', m', a', b'] \equiv X([a, b], [a', b']) \end{array} \right\}$$

if Δ is \mathbf{N}_* ; otherwise it is $\epsilon_0(\Delta)$. Using the selector $\{q_1, q_2, q_3\}$ above gives the set $R_{\mathbf{N}_*}$ of evaluation rules as

$$R_0 \cup \left\{ \begin{array}{l} \text{diagrec}(c, d, e, f) \Downarrow g \blacktriangleleft q_1 \ \& \ d \Downarrow g, \\ \text{diagrec}(c, d, e, f) \Downarrow g \blacktriangleleft q_2 \ \& \ e(l, \text{diagrec}([\text{succ}(l), 0], d, e, f)) \Downarrow g, \\ \text{diagrec}(c, d, e, f) \Downarrow g \blacktriangleleft q_3 \ \& \ f(a, b, \text{diagrec}([a, b], d, e, f)) \Downarrow g \end{array} \right\}.$$

Example 3.6 The expression system $L_{\sqcap} = ((K_{\sqcap}, N_{\sqcap}, \alpha_{\sqcap}, R_{\sqcap}), (D_{\sqcap}, \kappa_{\sqcap}, \epsilon_{\sqcap}))$ is the extension of L_0 with “intersection” types. Let K_{\sqcap} and N_{\sqcap} be $K_0 \cup \{\sqcap, \text{both}\}$ and $N_0 \cup \{\text{bothpeel}\}$, respectively. The arity of each operator is given by

$$\alpha_{\sqcap}(\rho) = \begin{cases} (0, 0) & \text{if } \rho \text{ is } \sqcap, \\ (0) & \text{if } \rho \text{ is } \text{both}, \\ (0, 1) & \text{if } \rho \text{ is } \text{bothpeel}, \\ \alpha_0(\rho) & \text{otherwise.} \end{cases}$$

The set R_{\sqcap} of evaluation rules is

$$R_0 \cup \{\text{bothpeel}(c, d) \Downarrow e \blacktriangleleft c \Downarrow \text{both}(a) \ \& \ d(a) \Downarrow e\}.$$

Let D_{\sqcap} be $D_0 \cup \{\sqcap\}$. The kind of each type constructor is given by

$$\kappa_{\sqcap}(\Delta) = \begin{cases} 11 & \text{if } \Delta \text{ is } \sqcap, \\ \kappa_0(\Delta) & \text{otherwise.} \end{cases}$$

Finally, $\epsilon_{\sqcap}(\Delta)$ is

$$(\{c \Downarrow \text{both}(a)\}, \{P_{1,1}(a, a') \wedge P_{2,1}(a, a')\})$$

if Δ is \sqcap ; otherwise it is $\epsilon_0(\Delta)$. Since the type theory treated in this paper is a purely polymorphic version of ITT, the introduction of “intersection” types will be of interest even though it precludes a direct, classical set-theoretic interpretation. (Consider, for example, $\mathbf{N} \rightarrow \mathbf{N}$ and $(\mathbf{N} \rightarrow \mathbf{N}) \rightarrow (\mathbf{N} \rightarrow \mathbf{N})$. When they are viewed as ordinary sets, their intersection is empty. But when they are viewed as types, the intersection defined in this paper contains the polymorphic identity $\lambda(x.x)$.) The extension of L_0 with “union” types would be obtained in a similar fashion; in particular, the equality prescription for “union” would be

$$(\{c \Downarrow \text{either}(a)\}, \{P_{1,1}(a, a') \vee P_{2,1}(a, a')\}).$$

However, because the formula included fails to be balanced, the extension is no longer an expression system.

Example 3.7 The expression system $L_{\{\cdot\}} = ((K_{\{\cdot\}}, N_{\{\cdot\}}, \alpha_{\{\cdot\}}, R_{\{\cdot\}}), (D_{\{\cdot\}}, \kappa_{\{\cdot\}}, \epsilon_{\{\cdot\}}))$ is the extension of L_0 with “subset” types. Let $K_{\{\cdot\}}$ and $N_{\{\cdot\}}$ be $K_0 \cup \{\{\cdot\}, \text{sub}\}$ and $N_0 \cup \{\text{subpeel}\}$, respectively. The arity of each operator is given by

$$\alpha_{\{\cdot\}}(\rho) = \begin{cases} (0, 1) & \text{if } \rho \text{ is } \{\cdot\}, \\ (0) & \text{if } \rho \text{ is } \text{sub}, \\ (0, 1) & \text{if } \rho \text{ is } \text{subpeel}, \\ \alpha_0(\rho) & \text{otherwise.} \end{cases}$$

The set $R_{\{\cdot\}}$ of evaluation rules is

$$R_0 \cup \{\text{subpeel}(c, d) \Downarrow e \blacktriangleleft c \Downarrow \text{sub}(a) \ \& \ d(a) \Downarrow e\}.$$

Let $D_{\{\cdot\}}$ be $D_0 \cup \{\{\cdot\}\}$. The kind of each type constructor is given by

$$\kappa_{\cap}(\Delta) = \begin{cases} 12 & \text{if } \Delta \text{ is } \{\cdot\}, \\ \kappa_0(\Delta) & \text{otherwise.} \end{cases}$$

Finally, $\epsilon_{\{\cdot\}}(\Delta)$ is

$$\left(\{c \Downarrow \text{sub}(a)\}, \left\{ \begin{array}{l} P_{1,1}(a, a') \\ \wedge P_{1,1}(a, a') \Rightarrow (\exists b. P_{1,2}(a, b) \wedge \exists b'. P_{1,2}(a', b', b')) \end{array} \right\} \right)$$

if Δ is $\{\cdot\}$; otherwise it is $\epsilon_0(\Delta)$. For more advanced treatment of “subset” types, see [38, 46].

It should be noted that an occurrence of the pattern $A \wedge (A \Rightarrow B)$ can be found in the description of $\epsilon_{\{\cdot\}}(\{\cdot\})$. (Also recall the descriptions of $\epsilon_0(\Sigma)$ and $\epsilon_0(W)$.) In [3], the abbreviation $A \wedge \Rightarrow B$ is used for $A \wedge (A \Rightarrow B)$. From a foundational viewpoint, $A \wedge (A \Rightarrow B)$ is different from $A \wedge B$ in that it can become a proposition on a weaker condition. This difference would be important if the present mechanism of inductively defining types were formalized in an extended variant of LTC that treats as fundamental the concept of proposition as well as the concept of truth. (For details of LTC, see [41, 37, 5].)

4. Type Systems

This section explains how a semantics of ITT can be obtained when the underlying language is specified as an expression system; the semantics will be provided by a certain form of a term model, a “type system.”

Let $L = (C, S)$ be an arbitrary expression system, where $C = (K, N, \alpha, R)$ is an evaluation system and $S = (D, \kappa, \epsilon)$ is a prescription system on C . In obtaining a semantics of ITT we need to consider not only a set of ordinary type constructors but also the set $\{\mathbf{U}_n \mid n \geq 0\}$ of *universes*. Accordingly, the following additional conditions are imposed on L : (i) $\{\mathbf{U}_n \mid n \geq 0\} \subset K$; (ii) $\alpha(\mathbf{U}_n) = ()$ for each $n \geq 0$; (iii) $D \subset K - \{\mathbf{U}_n \mid n \geq 0\}$.

4.1. Inductive Definitions of Type Systems

A *type system* τ over C is a subset of $\mathbb{T}_C \times \mathbb{T}_C \times \mathbb{P}_C$. Intuitively, $\tau(T, T', \phi)$ means that T and T' are equal types with ϕ representing their respective equalities. In particular, τ is *regular* if it satisfies the following conditions:

- (i) $\forall T \forall T' \forall \phi \forall \varphi. (\tau(T, T', \phi) \wedge \tau(T, T', \varphi) \Rightarrow \phi = \varphi)$;
- (ii) $\forall T \forall T' \forall \phi. (\tau(T, T', \phi) \Rightarrow \tau(T', T, \phi))$
 $\wedge \forall T \forall T' \forall T'' \forall \phi \forall \varphi. (\tau(T, T', \phi) \wedge \tau(T', T'', \varphi) \Rightarrow \phi = \varphi \wedge \tau(T, T'', \phi))$;
- (iii) $\forall T \forall T' \forall \phi. (\tau(T, T', \phi) \Leftrightarrow \exists S. (T \Downarrow_C S \wedge \tau(S, T', \phi)))$.

A regular type system can thus be regarded as a partial mapping from $\mathbb{T}_C \times \mathbb{T}_C$ to \mathbb{P}_C whose domain is in \mathbb{P}_C . The regularity of τ not only expresses the adequacy of τ but also is essential to proving the validity of each inference rule in the semantics provided by τ .

A type system σ over C is called a *base* if $\sigma(T, T', \phi)$ implies $T \Downarrow_C \mathbf{U}_m$ and $T' \Downarrow_C \mathbf{U}_m$ for some m . For each base σ , define the unary operation \mathcal{I}_L^σ on type systems over C by

$$\mathcal{I}_L^\sigma(\tau) = \sigma \cup \mathcal{K}_L(\tau),$$

where $\mathcal{K}_L(\tau)(T, T', \phi)$ is defined as

$$T \Downarrow_C \Delta(\Gamma) \wedge T' \Downarrow_C \Delta(\Gamma') \wedge \phi = \llbracket \Delta \rrbracket_C(\Phi)$$

for some type constructor $\Delta \in D$ of kind $\xi = \kappa(\Delta)$, for some ξ -bundles

$$\Gamma \equiv (T_{i,j}, t_{i,j,k})_{1 \leq i \leq |\xi|, 1 \leq j \leq \xi_i, 1 \leq k \leq \xi_{i,j}} \quad \text{and} \quad \Gamma' \equiv (T'_{i,j}, t'_{i,j,k})_{1 \leq i \leq |\xi|, 1 \leq j \leq \xi_i, 1 \leq k \leq \xi_{i,j}},$$

and for some ξ -structure

$$\Phi = (\phi_{i,j}, \eta_{i,j,k})_{1 \leq i \leq |\xi|, 1 \leq j \leq \xi_i, 1 \leq k \leq \xi_{i,j}}$$

such that $\text{Bun}_C^\xi(\tau, \Gamma, \Gamma', \Phi)$. Here $\text{Bun}_C^\xi(\tau, \Gamma, \Gamma', \Phi)$ means that two ξ -bundles Γ and Γ' are equal in τ with Φ representing their respective denotations. It is precisely defined as follows: for every i, j , and k ,

$$\tau(T_{i,j}(s_1, \dots, s_{j-1}), T'_{i,j}(s'_1, \dots, s'_{j-1}), \phi_{i,j}(s_1, \dots, s_{j-1})) \quad \text{and} \\ t_{i,j,k}(s_1, \dots, s_{j-1}) \equiv \eta_{i,j,k}(s_1, \dots, s_{j-1})$$

whenever $\phi_{i,1}(s_1, s'_1), \dots, \phi_{i,j-1}(s_1, s_2, \dots, s_{j-2})(s_{j-1}, s'_{j-1})$.

Obviously, $\tau \subset \tau'$ implies $\mathcal{I}_L^\sigma(\tau) \subset \mathcal{I}_L^\sigma(\tau')$ for each σ ; therefore,

$$\mu_L^\sigma = \bigcap \{ \tau \mid \mathcal{I}_L^\sigma(\tau) \subset \tau \}$$

becomes the least fixed point of \mathcal{I}_L^σ . This definition can be used to define the type systems \mathcal{S}_L^n and \mathcal{M}_L^n for each $n \geq 0$ and also to define the type systems \mathcal{S}_L and \mathcal{M}_L :

$$\mathcal{S}_L^n = \{ (T, T', =_{\mathcal{M}_L^n}) \mid 0 \leq m < n \wedge T \Downarrow_C \mathbf{U}_m \wedge T' \Downarrow_C \mathbf{U}_m \\ \wedge \forall S \forall S'. (S =_{\mathcal{M}_L^n} S' \Leftrightarrow \exists \phi. (\mathcal{M}_L^m(S, S', \phi))) \}, \\ \mathcal{M}_L^n = \mu_L^{\mathcal{S}_L^n}, \quad \mathcal{S}_L = \bigcup_{n \geq 0} \mathcal{S}_L^n, \quad \text{and} \quad \mathcal{M}_L = \mu_L^{\mathcal{S}_L}.$$

Lemma 4.1 (Regularity of Type Systems) *For each expression system L , the type systems \mathcal{S}_L^n ($n \geq 0$), \mathcal{M}_L^n ($n \geq 0$), \mathcal{S}_L , and \mathcal{M}_L are regular.*

Proof. Assume that L is a pair consisting of $C = (K, N, \alpha, R)$ and $S = (D, \kappa, \epsilon)$. By virtue of the determinism of C , it is straightforward to show that (i) for each $n \geq 0$, if \mathcal{M}_L^m is regular for every m such that $0 \leq m < n$, then \mathcal{S}_L^n is also regular; and (ii) if \mathcal{M}_L^m is regular for every $m \geq 0$, then \mathcal{S}_L is also regular. Hence, it suffices to prove that if σ is a regular base, then μ_L^σ is also regular.

Define the two type systems τ_{fun} and τ_{sytr} over C as

$$\{(T, T', \phi) \mid \forall \varphi. (\mu_L^\sigma(T, T', \varphi) \Rightarrow \phi = \varphi)\} \quad \text{and} \\ \{(T, T', \phi) \mid \mu_L^\sigma(T', T, \phi) \wedge \forall T'' \forall \varphi. (\mu_L^\sigma(T', T'', \varphi) \Rightarrow \phi = \varphi \wedge \mu_L^\sigma(T, T'', \phi))\},$$

respectively. Then it can be proved that $\mathcal{I}_L^\sigma(\tau_{\text{fun}}) \subset \tau_{\text{fun}}$; therefore, $\mu_L^\sigma \subset \tau_{\text{fun}}$. It can also be proved that $\mathcal{I}_L^\sigma(\tau_{\text{sytr}}) \subset \tau_{\text{sytr}}$; therefore, $\mu_L^\sigma \subset \tau_{\text{sytr}}$. And μ_L^σ is evaluation-preserving since $\mathcal{I}_L^\sigma(\mu_L^\sigma) = \mu_L^\sigma$. \square

4.2. Semantics of ITT

A *statement* has one of the following forms:

$$\begin{aligned} & T(x_1, \dots, x_n) \text{ type } (x_1 \in T_1, \dots, x_n \in T_n(x_1, \dots, x_{n-1})); \\ & T(x_1, \dots, x_n) = T'(x_1, \dots, x_n) \quad (x_1 \in T_1, \dots, x_n \in T_n(x_1, \dots, x_{n-1})); \\ & t(x_1, \dots, x_n) \in T(x_1, \dots, x_n) \quad (x_1 \in T_1, \dots, x_n \in T_n(x_1, \dots, x_{n-1})); \\ & t(x_1, \dots, x_n) = t'(x_1, \dots, x_n) \in T(x_1, \dots, x_n) \quad (x_1 \in T_1, \dots, x_n \in T_n(x_1, \dots, x_{n-1})). \end{aligned}$$

Here, x_1, \dots, x_n are distinct variables and $T, T', t, t', T_1, \dots, T_n$ are closed terms of the respective arities $n, n, n, n, 0, \dots, n-1$. The sequence $x_1 \in T_1, \dots, x_n \in T_n(x_1, \dots, x_{n-1})$ is called the *hypothesis* of the statement.

The type system \mathcal{M}_L provides a semantics of ITT when the underlying language is specified as L . In fact, the property $\mathcal{M}_L \models \Theta$ of a statement Θ in L , which property is that Θ is *valid* in \mathcal{M}_L , can be defined as follows:

$$\begin{aligned} \mathcal{M}_L \models T(x_1, \dots, x_n) \text{ type } (x_1 \in T_1, \dots, x_n \in T_n(x_1, \dots, x_{n-1})) \\ \Leftrightarrow \exists (\phi_j)_{1 \leq j \leq n} \exists \phi. \text{Bun}_C^{1 \cdots n(n+1)}(\mathcal{M}_L, (T_j)_{1 \leq j \leq n}, T, \\ (T_j)_{1 \leq j \leq n}, T, \\ (\phi_j)_{1 \leq j \leq n}, \phi); \\ \mathcal{M}_L \models T(x_1, \dots, x_n) = T'(x_1, \dots, x_n) \quad (x_1 \in T_1, \dots, x_n \in T_n(x_1, \dots, x_{n-1})) \\ \Leftrightarrow \exists (\phi_j)_{1 \leq j \leq n} \exists \phi. \text{Bun}_C^{1 \cdots n(n+1)}(\mathcal{M}_L, (T_j)_{1 \leq j \leq n}, T, \\ (T_j)_{1 \leq j \leq n}, T', \\ (\phi_j)_{1 \leq j \leq n}, \phi); \\ \mathcal{M}_L \models t(x_1, \dots, x_n) \in T(x_1, \dots, x_n) \quad (x_1 \in T_1, \dots, x_n \in T_n(x_1, \dots, x_{n-1})) \\ \Leftrightarrow \exists (\phi_j)_{1 \leq j \leq n} \exists \phi \exists \eta. \text{Bun}_C^{1 \cdots n(n+1)\square}(\mathcal{M}_L, (T_j)_{1 \leq j \leq n}, T, t, \\ (T_j)_{1 \leq j \leq n}, T, t, \\ (\phi_j)_{1 \leq j \leq n}, \phi, \eta); \\ \mathcal{M}_L \models t(x_1, \dots, x_n) = t'(x_1, \dots, x_n) \in T(x_1, \dots, x_n) \quad (x_1 \in T_1, \dots, x_n \in T_n(x_1, \dots, x_{n-1})) \\ \Leftrightarrow \exists (\phi_j)_{1 \leq j \leq n} \exists \phi \exists \eta. \text{Bun}_C^{1 \cdots n(n+1)\square}(\mathcal{M}_L, (T_j)_{1 \leq j \leq n}, T, t, \\ (T_j)_{1 \leq j \leq n}, T, t', \\ (\phi_j)_{1 \leq j \leq n}, \phi, \eta). \end{aligned}$$

For a hypothesis-free statement, the definition is simply given as follows:

$$\begin{aligned} \mathcal{M}_L \models T \text{ type} & \Leftrightarrow \exists \phi. \mathcal{M}_L(T, T, \phi); \\ \mathcal{M}_L \models T = T' & \Leftrightarrow \exists \phi. \mathcal{M}_L(T, T', \phi); \\ \mathcal{M}_L \models t \in T & \Leftrightarrow \exists \phi. \mathcal{M}_L(T, T, \phi) \wedge \phi(t, t); \\ \mathcal{M}_L \models t = t' \in T & \Leftrightarrow \exists \phi. \mathcal{M}_L(T, T, \phi) \wedge \phi(t, t'). \end{aligned}$$

5. “Soundness”

5.1. Validity Persistence of Inference Rules

The system of inference rules given in [33, 34] can be formalized as the deduction system ITT^0 , which is built up in the language L_0 . When L is an arbitrary extension of L_0 , the property $\text{ITT}^0 \vdash_L \Theta$ of a statement Θ in L means that a derivation of Θ can be obtained in ITT^0 by using a valuation of L to get instances of inference rules. The following theorem uses Lemma 4.1 and shows that the inference rules given in [33, 34] are sound.

Theorem 5.1 (Validity Persistence of Inference Rules) *If L is an extension of L_0 , then $\text{ITT}^0 \vdash_L \Theta$ implies $\mathcal{M}_L \models \Theta$ for every statement Θ in L .*

Proof. Let $L = (C, S)$ be an arbitrary extension of L_0 . The theorem is proved by induction on the structure of the derivation of Θ . Consider only the case in which the derivation ends up with an instance

$$\frac{c = f \in \Pi(A, B) \quad a = d \in A}{\text{apply}(c, a) = \text{apply}(f, d) \in B(a)}$$

of the hypothesis-free second Π -elimination rule. The treatment of the other cases follows a similar pattern.

By the induction hypothesis, there exists ϕ such that $\mathcal{M}_L(\Pi(A, B), \Pi(A, B), \phi)$ and $\phi(c, f)$. Thus $\mathcal{K}_L(\mathcal{M}_L)(\Pi(A, B), \Pi(A, B), \phi)$ since $\mathcal{M}_L = \mathcal{I}_L^{S^L}(\mathcal{M}_L)$; so there exist ϕ_A and ϕ_B such that $\mathcal{M}_L(A, A, \phi_A)$,

$$\forall u \forall u'. (\phi_A(u, u') \Rightarrow \mathcal{M}_L(B(u), B(u'), \phi_B(u)),$$

and $\phi = \llbracket \Pi \rrbracket_C(\phi_A, \phi_B)$. Since $\phi(c, f)$, there also exist b and e such that $c \downarrow_C \lambda(b)$, $f \downarrow_C \lambda(e)$, and

$$\forall u \forall u'. (\phi_A(u, u') \Rightarrow \phi_B(u, b(u), e(u'))).$$

Again by the induction hypothesis, there exists ψ such that $\mathcal{M}_L(A, A, \psi)$ and $\psi(a, d)$. In view of the regularity of \mathcal{M}_L , which is guaranteed by Lemma 4.1, $\psi = \phi_A$; thus, $\phi_A(a, d)$. Hence $\mathcal{M}_L(B(a), B(d), \phi_B(a))$ and $\phi_B(a, b(a), e(d))$, which imply that

$$\mathcal{M}_L(B(a), B(a), \phi_B(a)) \quad \text{and} \quad \phi_B(a, b(a), e(d))$$

by virtue of the regularity of \mathcal{M}_L . Here $b(a)$ can be replaced with $\text{apply}(c, a)$ because they have the same value. For the same reason, $e(d)$ can be replaced with $\text{apply}(f, d)$. This proves that $\mathcal{M}_L \models \text{apply}(c, a) = \text{apply}(f, d) \in B(a)$. \square

Theorem 5.1 can be used to show the logical consistency of ITT^0 : if $\text{ITT}^0 \vdash_{L_0} t \in \mathbb{N}_0$ ($\equiv \perp$) was true for some t , then t would be an element of the empty set.

Corollary 5.2 *If L is an extension of L_0 and if L' is an extension of L , then $\text{ITT}^0 \vdash_L \Theta$ implies $\mathcal{M}_{L'} \models \Theta$ for every statement Θ in L .*

Corollary 5.2 enables each example of an aspect of open-endedness shown informally in Section 1.1 to be elaborated in terms of the formal concepts developed in this paper. For instance, the second example “if we have a judgement $f \in \mathbf{N} \rightarrow \mathbf{N}$, then the object f is applicable to every possible future object of \mathbf{N} ” can be given the following formal expression.

Corollary 5.3 *Let L be an extension of L_0 and f a closed term of L such that $\text{ITT}^0 \vdash_L f \in \mathbf{N} \rightarrow \mathbf{N}$. Then $\mathcal{M}_{L'} \models e \in \mathbf{N}$ implies $\mathcal{M}_{L'} \models \text{apply}(f, e) \in \mathbf{N}$ for every extension L' of L and for every closed term e of L' .*

The reader might try to prove the following stronger form of open-endedness: if L is an extension of L_0 and if f is a closed term of L such that $\mathcal{M}_L \models f \in \mathbf{N} \rightarrow \mathbf{N}$, then $\mathcal{M}_{L'} \models e \in \mathbf{N}$ implies $\mathcal{M}_{L'} \models \text{apply}(f, e) \in \mathbf{N}$ for every extension L' of L and for every closed term e of L' . Unfortunately, this will be refuted with the present semantics.

5.2. Monotonicity and Conservativity

Some observations on a series $\{\mathcal{M}_L\}$ of type systems might be helpful here. The mapping \mathcal{M} defined by $\mathcal{M}(L) = \mathcal{M}_L$ is called *monotonic* if $\mathcal{M}_L \models \Theta$ implies $\mathcal{M}_{L'} \models \Theta$ for every L , for every extension L' of L , and for every statement Θ in L . The mapping \mathcal{M} is said to be *conservative* if $\mathcal{M}_{L'} \models \Theta$ implies $\mathcal{M}_L \models \Theta$ for every L , for every extension L' of L , and for every statement Θ in L .

Claim 5.4 *\mathcal{M} is not monotonic.*

Take as L the extension L_{urec} of L_0 obtained by adding the urec_n -operator of arity

$$(0, 0, 0, 2, 4, 4, 4, 4, 4, \underbrace{0, \dots, 0}_n)$$

(for each $n \geq 0$) together with the following set of evaluation rules, which are used to express the structural recursion on the universes [38]:

$$\left\{ \begin{array}{l} \text{urec}_n(c, d_1, \dots, d_{8+n}) \Downarrow e \triangleleft c \Downarrow \mathbf{N}_k \ \& \ d_1 \Downarrow e \ (k \geq 0), \\ \text{urec}_n(c, d_1, \dots, d_{8+n}) \Downarrow e \triangleleft c \Downarrow \mathbf{N} \ \& \ d_2 \Downarrow e, \\ \text{urec}_n(c, d_1, \dots, d_{8+n}) \Downarrow e \triangleleft c \Downarrow \text{List}(A) \ \& \ d_3(A, \text{urec}_n(A, d_1, \dots, d_{8+n})) \Downarrow e, \\ \text{urec}_n(c, d_1, \dots, d_{8+n}) \Downarrow e \triangleleft c \Downarrow \text{Eq}(A, a, b) \\ \quad \& \ d_4(A, a, b, \text{urec}_n(A, d_1, \dots, d_{8+n})) \Downarrow e, \\ \text{urec}_n(c, d_1, \dots, d_{8+n}) \Downarrow e \triangleleft c \Downarrow A + B \\ \quad \& \ d_5(A, B, \text{urec}_n(A, d_1, \dots, d_{8+n}), \text{urec}_n(B, d_1, \dots, d_{8+n})) \Downarrow e, \\ \text{urec}_n(c, d_1, \dots, d_{8+n}) \Downarrow e \triangleleft c \Downarrow \Pi(A, B) \\ \quad \& \ d_6(A, \lambda(B), \text{urec}_n(A, d_1, \dots, d_{8+n}), \lambda(w.\text{urec}_n(B(w), d_1, \dots, d_{8+n}))) \Downarrow e, \\ \text{urec}_n(c, d_1, \dots, d_{8+n}) \Downarrow e \triangleleft c \Downarrow \Sigma(A, B) \\ \quad \& \ d_7(A, \lambda(B), \text{urec}_n(A, d_1, \dots, d_{8+n}), \lambda(w.\text{urec}_n(B(w), d_1, \dots, d_{8+n}))) \Downarrow e, \\ \text{urec}_n(c, d_1, \dots, d_{8+n}) \Downarrow e \triangleleft c \Downarrow W(A, B) \\ \quad \& \ d_8(A, \lambda(B), \text{urec}_n(A, d_1, \dots, d_{8+n}), \lambda(w.\text{urec}_n(B(w), d_1, \dots, d_{8+n}))) \Downarrow e, \\ \text{urec}_n(c, d_1, \dots, d_{8+n}) \Downarrow e \triangleleft c \Downarrow \mathbf{U}_k \ \& \ d_{9+k} \Downarrow e \ (k \in \{0, 1, \dots, n-1\}) \end{array} \right\}.$$

Take as L' an arbitrary extension of L_{urec} that contains a fresh type constructor. And take as Θ

$$\Pi(\mathbf{U}_0, v.\text{urec}_0(v, \mathbf{N}, \mathbf{N}, xy.\mathbf{N}, xyzw.\mathbf{N}, xyzw.\mathbf{N}, xyzw.\mathbf{N}, xyzw.\mathbf{N}, xyzw.\mathbf{N}, xyzw.\mathbf{N})) \in \mathbf{U}_1.$$

This is a counterexample of the monotonicity of \mathcal{M} .

Claim 5.5 \mathcal{M} is not conservative.

Take as L the extension $L_{\{\dagger\}}$ of L_0 with “subset” types, which is described in Example 3.7. Take as L' the extension of $L_{\{\dagger\}}$ obtained by adding the large collection of “oracles” introduced by Howe [24]. And take as Θ

$$\text{sub}(0) \in \{\mathbf{N} \mid x.\Pi(\mathbf{U}_0, A.\Pi(A, y.\mathbf{N}_0) + A)\} (\equiv \{x \in \mathbf{N} \mid \forall A \in \mathbf{U}_0. \neg A \vee A\}).$$

This is a counterexample of the conservativity of \mathcal{M} .

6. “Completeness”

6.1. Completeness of Type Systems

One of the theoretical goals of this paper is to relate the type system \mathcal{M}_L (which is denotational, involving the concept of types, and inductively defined) to the congruence \sim_C (which is operational, type-free, and coinductively defined). The following theorem shows that \mathcal{M}_L is *complete* with respect to \sim_C . The proof uses the sequentiality and strict positivity conditions on the inductive definitions of types.

Theorem 6.1 (Completeness of Type Systems) *Let $L = (C, S)$ be an expression system. If $\mathcal{M}_L \models T$ type and $T \sim_C T'$, then $\mathcal{M}_L \models T'$ type and $\mathcal{M}_L \models T = T'$. Moreover, if $\mathcal{M}_L \models t \in T$ and $t \sim_C t'$, then $\mathcal{M}_L \models t' \in T$ and $\mathcal{M}_L \models t = t' \in T$.*

Proof. Assume that $C = (K, N, \alpha, R)$ and $S = (D, \kappa, \epsilon)$. A type system τ over C is called *total* if

$$\forall T' \forall S'. (T \sim_C T' \wedge S \sim_C S' \Rightarrow \tau(T', S', \phi))$$

and

$$\forall t \forall s. (\phi(t, s) \Rightarrow \forall t' \forall s'. (t \sim_C t' \wedge s \sim_C s' \Rightarrow \phi(t', s')))$$

for every T, S , and ϕ such that $\tau(T, S, \phi)$. It will be demonstrated below that \mathcal{S}_L^n , \mathcal{M}_L^n , \mathcal{S}_L , and \mathcal{M}_L are total for every $n \geq 0$.

It is straightforward to show that (i) for each $n \geq 0$, if \mathcal{M}_L^m is total for every m such that $0 \leq m < n$, then \mathcal{S}_L^n is also total; and (ii) if \mathcal{M}_L^m is total for every $m \geq 0$, then \mathcal{S}_L is also total. Hence, it suffices to prove that if σ is total, then μ_L^σ is also total.

Define the type system τ_{tot} over C as

$$\{(T, S, \phi) \mid \forall T' \forall S'. (T \sim_C T' \wedge S \sim_C S' \Rightarrow \mu_L^\sigma(T', S', \phi)) \wedge \forall t \forall s. (\phi(t, s) \Rightarrow \forall t' \forall s'. (t \sim_C t' \wedge s \sim_C s' \Rightarrow \phi(t', s')))\}.$$

Because $\mathcal{I}_L^\sigma(\tau_{\text{tot}}) \subset \tau_{\text{tot}}$ implies $\mu_L^\sigma \subset \tau_{\text{tot}}$, it suffices to show that $\sigma \subset \tau_{\text{tot}}$ and $\mathcal{K}_L(\tau_{\text{tot}}) \subset \tau_{\text{tot}}$. As the former is immediate, we will only consider the latter.

Assume that $\mathcal{K}_L(\tau_{\text{tot}})(T, S, \phi)$. That is,

$$T \Downarrow_C \Delta(\Gamma), \quad S \Downarrow_C \Delta(\Upsilon), \quad \text{and} \quad \phi = \llbracket \Delta \rrbracket_C(\Phi)$$

for some $\Delta \in D$ of kind $\xi = \kappa(\Delta)$, for some ξ -bundles Γ and Υ , and for some ξ -structure Φ such that $\text{Bun}_C^\xi(\tau_{\text{tot}}, \Gamma, \Upsilon, \Phi)$.

If $T \sim_C T'$, then there exists Γ' such that $T' \Downarrow_C \Delta(\Gamma')$ and $\Gamma \sim_C \Gamma'$. And if $S \sim_C S'$, then there exists Υ' such that $S' \Downarrow_C \Delta(\Upsilon')$ and $\Upsilon \sim_C \Upsilon'$. Then it can be proved that $\text{Bun}_C^\xi(\mu_L^\sigma, \Gamma', \Upsilon', \Phi)$. This implies that $\mathcal{K}_L(\mu_L^\sigma)(T', S', \phi)$ and hence $\mu_L^\sigma(T', S', \phi)$.

Assume that $\epsilon(\Delta) = (Q, \{A_q\}_{q \in Q})$, and define $\varphi(t, s)$ as

$$\forall t' \forall s'. (t \sim_C t' \wedge s \sim_C s' \Rightarrow \phi(t', s')).$$

The following immediately implies $\phi \subset \varphi$ and thus completes the proof of Theorem 6.1:

$$\exists q \in Q. \exists V \exists V'. {}^t \llbracket q \rrbracket_C^V \wedge {}^{t'} \llbracket q \rrbracket_C^{V'} \wedge \llbracket A_q \rrbracket_C^{V \oplus_q V'}(\Phi, \varphi)$$

implies $\varphi(t, t')$ for every t and t' . By virtue of Theorem 2.3 and the sequentiality of each q , this is an immediate consequence of the following, which can be proved by induction on the construction of A_q : if $V(e) \sim_C W(e)$ for every $e \in \mathbb{V}(q)$ and if $V'(e) \sim_C W'(e)$ for every $e \in \mathbb{V}(q)$, then $\llbracket A_q \rrbracket_C^{V \oplus_q V'}(\Phi, \varphi)$ implies $\llbracket A_q \rrbracket_C^{W \oplus_q W'}(\Phi, \phi)$. Note that the strict positivity condition on A_q is used to prove the case for implication. \square

The converse of Theorem 6.1 says that \mathcal{M}_L is *correct* with respect to \sim_C . That is, $\mathcal{M}_L \models T = T'$ implies $T \sim_C T'$, and $\mathcal{M}_L \models t = t' \in T$ implies $t \sim_C t'$ for each expression system $L = (C, S)$. Unfortunately, this is not true because the definition of \sim_C does not take typehood into account. For example, the statement

$$\lambda(x.\text{succ}(x)) = \lambda(x.\text{natrec}(x, \text{succ}(0), xy.\text{succ}(y))) \in \Pi(\mathbf{N}, x.\mathbf{N}) \equiv \mathbf{N} \rightarrow \mathbf{N}$$

is valid in \mathcal{M}_{L_0} , but obviously

$$\lambda(x.\text{succ}(x)) \sim_{C_0} \lambda(x.\text{natrec}(x, \text{succ}(0), xy.\text{succ}(y)))$$

does not hold. A correct and complete model is called *fully abstract*. In [31], Loader presented the fully abstract model of a λ -calculus with inductively defined types. The calculus is, however, “typed” and hence strongly normalizing; moreover, it is quite simple in that it covers only the propositional fragment of inductively defined types. (That is, “dependent types” are not treated.)

6.2. Type-Free Equational Reasoning

As a corollary of Theorem 6.1, a form of type-free equational reasoning can be obtained. This can be generalized, however, in view of the open-endedness property

in ITT. For ITT to be sensible as an open-ended framework, each inference rule must be persistently valid with respect to language extension. (As already shown in Theorem 5.1, the inference rules given by Martin-Löf [33, 34] are basic examples of such rules.) In other words, ITT allows every form of inference rule as long as it remains valid in each semantics constructed from an extension of an initial underlying language. In view of this open-endedness property, a generalized form of type-free equational reasoning can be formulated.

Corollary 6.2 (Type-Free Equational Reasoning) *Let $L^* = (C^*, S^*)$ be an extension of an expression system $L = (C, S)$. Then the following form of reasoning can be allowed in ITT from the “validity persistence” viewpoint: statements “ T' type” and “ $T = T'$ ” in L^* can be inferred from “ T type” in L , provided that $T \sim_{C^*} T'$; moreover, statements “ $t' \in T$ ” and “ $t = t' \in T$ ” in L^* can be inferred from “ $t \in T$ ” in L , provided that $t \sim_{C^*} t'$.*

Proof. Consider only the latter part of the corollary and assume that $t \in T$ in the language L and $t \sim_{C^*} t'$. Let $L^{**} = (C^{**}, S^{**})$ be an arbitrary extension of L^* . Then $\mathcal{M}_{L^{**}} \models t \in T$ holds by the assumption. Hence if $t \sim_{C^{**}} t'$ also holds, Theorem 6.1 implies that $\mathcal{M}_{L^{**}} \models t' \in T$ and $\mathcal{M}_{L^{**}} \models t = t' \in T$.

There remains the task of proving that $\leq_{C^*} C \leq_{C^{**}}$. To prove $\leq_{C^*} C \leq_{C^{**}}$, assume that $u \leq_{C^*} u'$ and $u \Downarrow_{C^{**}} \theta(\bar{s})$. Then $u \Downarrow_{C^*} \theta(\bar{s})$ by Lemma 3.1, so there exists $\theta(\bar{s}')$ such that $u' \Downarrow_{C^*} \theta(\bar{s}')$ and $\bar{s} \leq_{C^*} \bar{s}'$. \square

Theorem 6.1 and Corollary 6.2 are easily extensible to general cases for hypothetical statements. For example, “ $t'(x) \in T'(x) (x \in S')$ ” in L^* can be inferred from “ $t(x) \in T(x) (x \in S)$ ” in L , provided that $t \sim_{C^*} t'$, $T \sim_{C^*} T'$, and $S \sim_{C^*} S'$.

To illustrate how Corollary 6.2 can be applied in practice, a small example of program transformation is presented here. Assuming that the initial expression system L_0 is used, define the program `reverse(c)`, which calculates the reverse of a given list c , as follows:

$$\begin{aligned} \text{reverse}(c) &\equiv \text{listrec}(c, \text{nil}, \text{xyz.append}(z, \text{cons}(x, \text{nil}))); \\ \text{append}(c, d) &\equiv \text{listrec}(c, d, \text{xyz.eapply}(\lambda(w.\text{cons}(x, w)), z)). \end{aligned}$$

Here `append(c, d)` is defined using the “eager” version of functional application; its usual definition is `listrec(c, d, xyz.cons(x, z))`.

A problem here is that the program `reverse(c)` is rather inefficient. It can be transformed to a better one, `reverse1(c)`, with a “tail-recursive” form:

$$\begin{aligned} \text{reverse1}(c) &\equiv \text{apply}(\text{rev1}(c), \text{nil}); \\ \text{rev1}(c) &\equiv \text{listrec}(c, \lambda(w.w), \text{xyz.\lambda}(w.\text{apply}(z, \text{cons}(x, w)))). \end{aligned}$$

It is easy to show that these two programs—simple but inefficient `reverse(c)` and somewhat tricky but more efficient `reverse1(c)`—are observationally indistinguishable; that is, $\text{reverse}(c) \sim_{C_0} \text{reverse1}(c)$ holds for every closed term c of C_0 .

Assume that we have a program of the form

$$\dots \text{apply}(\lambda(w.\text{reverse}(w)), c) \dots$$

of type T . To obtain a better program, we only have to replace `reverse` with `reverse1`:

$$\dots \text{apply}(\lambda(w.\text{reverse1}(w)), c) \dots \in T.$$

The validity of this inference is guaranteed by Theorem 2.3 and Corollary 6.2. Also guaranteed is the equality of the two programs:

$$\dots \text{apply}(\lambda(w.\text{reverse}(w)), c) \dots = \dots \text{apply}(\lambda(w.\text{reverse1}(w)), c) \dots \in T.$$

A more natural and comprehensible way of introducing an efficient version of `reverse` is to simply add to C_0 a new program-forming operator together with the associated evaluation rules. Let K_{rev2} be K_0 and let N_{rev2} be $N_0 \cup \{\text{rev2}\}$. And $\alpha_{\text{rev2}}(\rho)$ is $(0, 0)$ if ρ is `rev2`; otherwise it is $\alpha_0(\rho)$. Let R_{rev2} be

$$R_0 \cup \left\{ \begin{array}{l} \text{rev2}(c, d) \Downarrow e \quad \blacktriangleleft \quad c \Downarrow \text{nil} \ \& \ d \Downarrow e, \\ \text{rev2}(c, d) \Downarrow e \quad \blacktriangleleft \quad c \Downarrow \text{cons}(a, b) \ \& \ \text{rev2}(b, \text{cons}(a, d)) \Downarrow e \end{array} \right\}.$$

Thus the evaluation system $C_{\text{rev2}} = (K_{\text{rev2}}, N_{\text{rev2}}, \alpha_{\text{rev2}}, R_{\text{rev2}})$ is obtained. The new program form `reverse2(c)`, which is as efficient as `reverse1(c)`, is given by

$$\text{reverse2}(c) \equiv \text{rev2}(c, \text{nil}).$$

Again it is easy to show that $\text{reverse}(c) \sim_{C_{\text{rev2}}} \text{reverse2}(c)$ holds for every closed term c of C_{rev2} . Although the underlying language is extended by addition, we can also use Corollary 6.2 to obtain a better program by simply replacing `reverse` with `reverse2`.

7. Concluding Remarks

This paper has treated ITT as an open-ended framework with the following three layers:

- (i) an underlying expression system L consisting of
 - (a) an evaluation system C with the program equivalence \sim_C and
 - (b) a prescription system S defining a collection of type constructors,
into which various forms of objects and types can be incorporated successively;
- (ii) the PER-based type system \mathcal{M}_L constructed from L ;
- (iii) inference rules to be interpreted in \mathcal{M}_L .

To guarantee this framework to be sensible, the present paper has shown that all the inference rules of the theory are sound; that is, they are valid in every \mathcal{M}_L whenever L is an extension of an initial expression system. Moreover, \sim_C and \mathcal{M}_L have been related by showing that \mathcal{M}_L is complete with respect to \sim_C . This result has been used to present a useful form of type-free equational reasoning.

Since the present framework deals with not a specific series of languages but instead language extensions in general, the resulting uniformity in type constructions can be used to prove a variety of results on types “uniformly” instead of “on a case-by-case basis,” as exemplified in Lemma 4.1 and Theorem 6.1.

Several improvements, of course, may be possible. First, the syntax of the underlying languages used in this paper is restricted for the sake of simplicity and hence has limited expressive power. For example, operators having higher-order arities, such as “funsplit” [38, 12], are not supported in the present syntax.

Second, some useful classes of types that could be introduced into the theory are not treated in the present framework. Two examples are “quotient” types [13] and unions of disjoint types, both of which are formed by “conditional” type-constructors. The problem is that, in the present formulation, the equality prescription for each type constructor is required to generate PERs for any argument structures, whereas the type constructor forming quotient types, for example, only works when one of its arguments represents an equivalence relation.

Third, the present formulation focuses on language extensions and associated semantics-expansions; possible extensions of inference rules are not discussed sufficiently. Instead, the present paper only explains a principle that the theory can allow every form of inference rule as long as it is persistently valid. A syntactical exposition of the class of inference rules that are validated by this principle, however, is particularly important for implementing the theory. An extension of Dybjer’s rules [15, 16], for example, would constitute the core of that class.

Fourth, the present mechanism of inductively defining types can be formalized in a single formal language: an extended variant of LTC might provide such a language. This possible formalization seems closely related to the work by Sato [40] and would generalize the results shown by Smith [41] and by Aczel *et al.* [5].

Finally, the practical value of the equational reasoning presented in this paper needs to be substantiated by further experiments within a variety of proof-development systems implementing ITT and its family. Two different approaches may be possible—to combine a foreign equational prover with ITT or to internalize \sim_C within an extension of ITT. This is important future work.

Acknowledgements

I am grateful to Hirofumi Katsuno, Mizuhito Ogawa, Akihiro Umemura, Zurab Khasidashvili, and Shigeki Goto for their encouragement and helpful advice. I also thank Masahiko Sato, Makoto Tatsuta, and Yuki Yoshi Kameyama for their valuable comments on an earlier version of this paper. Special thanks are also due to Masako Takahashi, whose constructive suggestions helped me improve the present paper. I also thank the two anonymous referees for their careful reading and constructive criticisms.

References

1. S. Abramsky, “The lazy lambda-calculus,” in *Research Topics in Functional Programming*, ed. D. A. Turner (Addison-Wesley, Reading, Massachusetts, 1989) pp. 65–116.
2. P. Aczel, “A general Church-Rosser theorem,” Technical Report, University of Manchester, 1978.
3. P. Aczel, “Frege structures and the notions of proposition, truth and set,” in *The*

- Kleene Symposium*, eds. J. Barwise, H. J. Keisler, and K. Kunen (North-Holland, Amsterdam, 1980) pp. 31–59.
4. P. Aczel, *Non-Well-Founded Sets* (CSLI Publications, Stanford, California, 1988).
 5. P. Aczel, D. P. Carlisle, and N. Mendler, “Two frameworks of theories and their implementation in Isabelle,” in *Logical Frameworks*, eds. G. Huet and G. Plotkin (Cambridge University Press, Cambridge, 1991) pp. 3–39.
 6. S. F. Allen, “A non-type-theoretic definition of Martin-Löf’s types,” in *Proc. 2nd IEEE Symp. on Logic in Computer Science*, 1987, pp. 215–221.
 7. S. F. Allen, “A non-type-theoretic semantics for type-theoretic language,” Ph. D. Thesis, Cornell University, 1987.
 8. R. Backhouse, P. Chisholm, G. Malcolm, and E. Saaman, “Do-it-yourself type theory,” *Formal Aspects of Computing* **1** (1989) 19–84.
 9. J. Barwise and L. S. Moss, *Vicious Circles: On the Mathematics of Non-Wellfounded Phenomena* (CSLI Publications, Stanford, California, 1996).
 10. D. A. Basin and D. J. Howe, “Some normalization properties of Martin-Löf’s type theory, and applications,” in *Proc. 1st Internat. Conf. on Theoretical Aspects of Computer Software, Lecture Notes in Computer Science*, Vol. 526 (Springer, Berlin, 1991) pp. 475–494.
 11. M. J. Beeson, “Recursive models for constructive set theories,” *Ann. Math. Logic* **23** (1982) 127–178.
 12. A. Bossi and S. Valentini, “An intuitionistic theory of types with assumptions of high-arity variables,” *Ann. Pure and Applied Logic* **57** (1992) 93–149.
 13. R. L. Constable *et al.*, *Implementing Mathematics with the Nuprl Proof Development System* (Prentice-Hall, Englewood Cliffs, New Jersey, 1986).
 14. T. Coquand and C. Paulin, “Inductively defined types,” in *Proc. Internat. Conf. on Computer Logic, Lecture Notes in Computer Science*, Vol. 417 (Springer, Berlin, 1988) pp. 50–66.
 15. P. Dybjer, “Inductive sets and families in Martin-Löf’s type theory and their set-theoretic semantics,” in *Logical Frameworks*, eds. G. Huet and G. Plotkin (Cambridge University Press, Cambridge, 1991) pp. 280–306.
 16. P. Dybjer, “Inductive families,” *Formal Aspects of Computing* **6** (1994) 440–465.
 17. P. Dybjer and A. Setzer, “A finite axiomatization of inductive-recursive definitions,” in *Proc. 4th Internat. Conf. on Typed Lambda Calculi and Applications, Lecture Notes in Computer Science*, Vol. 1581 (Springer, Berlin, 1999) pp. 129–146.
 18. S. Feferman, “A language and axioms for explicit mathematics,” in *Algebra and Logic, Lecture Notes in Mathematics*, Vol. 450 (Springer, Berlin, 1975) pp. 87–139.
 19. S. Feferman, “Polymorphic typed lambda-calculi in a type-free axiomatic framework,” in *Logic and Computation*, ed. W. Sieg, *Contemporary Mathematics*, Vol. 106 (American Mathematical Society, Providence, Rhode Island, 1990) pp. 101–136.
 20. R. Harper, “Constructing type systems over an operational semantics,” *J. Symbolic Comput.* **14** (1992) 71–84.
 21. S. Hayashi and S. Kobayashi, “A new formalization of Feferman’s system of functions and classes and its relation to Frege structure,” *Internat. J. Foundations Comput. Sci.* **6** (1995) 187–202.
 22. S. Hayashi and H. Nakano, *PX: A Computational Logic* (The MIT Press, Cambridge, Massachusetts, 1988).
 23. D. J. Howe, “Equality in lazy computation systems,” in *Proc. 4th IEEE Symp. on*

- Logic in Computer Science*, 1989, pp. 198–203.
24. D. J. Howe, “On computational open-endedness in Martin-Löf’s type theory,” in *Proc. 6th IEEE Symp. on Logic in Computer Science*, 1991, pp. 162–172.
 25. D. J. Howe, “Proving congruence of bisimulation in functional programming languages,” *Inform. and Comput.* **124** (1996) 103–112.
 26. G. Kahn, “Natural semantics,” in *Proc. Symp. on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science*, Vol. 247 (Springer, Berlin, 1987) pp. 22–39.
 27. Y. Kameyama, “A type-free theory of half-monotone inductive definitions,” *Internat. J. Foundations Comput. Sci.* **6** (1995) 203–234.
 28. Z. Khasidashvili, “The Church-Rosser theorem in orthogonal combinatory reduction systems,” INRIA Research Report 1825, INRIA, 1992.
 29. J. W. Klop, “Combinatory reduction systems,” *Mathematical Centre Tracts* 127, CWI, 1980.
 30. S. Kobayashi and M. Tatsuta, “Realizability interpretation of generalized inductive definitions,” *Theoret. Comput. Sci.* **131** (1994) 121–138.
 31. R. Loader, “Equational theories for inductive types,” *Ann. Pure and Applied Logic* **84** (1997) 175–217.
 32. P. Martin-Löf, “Hauptsatz for the intuitionistic theory of iterated inductive definitions,” in *Proc. 2nd Scandinavian Logic Symposium*, ed. J. E. Fenstad (North-Holland, Amsterdam, 1971) pp. 179–216.
 33. P. Martin-Löf, “Constructive mathematics and computer programming,” in *Logic, Methodology and Philosophy of Science VI*, eds. L. J. Cohen, J. Los, H. Pfeiffer, and K. P. Podewsky (North-Holland, Amsterdam, 1982) pp. 153–175.
 34. P. Martin-Löf, *Intuitionistic Type Theory* (Bibliopolis, Napoli, 1984).
 35. P. Martin-Löf, “Truth of a proposition, evidence of a judgement, validity of a proof,” *Synthese* **73** (1987) 407–420.
 36. J. McCarthy, “A basis for a mathematical theory of computations,” in *Computer Programming and Formal Systems*, eds. P. Braffort and D. Hirschberg (North-Holland, Amsterdam, 1963) pp. 33–70.
 37. P. F. Mendler and P. Aczel, “The notion of a framework and a framework for LTC,” in *Proc. 3rd IEEE Symp. on Logic in Computer Science*, 1988, pp. 392–399.
 38. B. Nordström, K. Petersson, and J. M. Smith, *Programming in Martin-Löf’s Type Theory: An Introduction* (Clarendon Press, Oxford, 1990).
 39. G. D. Plotkin, “A structural approach to operational semantics,” Report DAIMI FN-19, Aarhus University, 1981.
 40. M. Sato, “Adding proof objects and inductive definition mechanisms to Frege structures,” in *Proc. 1st Internat. Conf. on Theoretical Aspects of Computer Software, Lecture Notes in Computer Science*, Vol. 526 (Springer, Berlin, 1991) pp. 53–87.
 41. J. Smith, “An interpretation of Martin-Löf’s type theory in a type-free theory of propositions,” *J. Symbolic Logic* **49** (1984) 730–753.
 42. S. F. Smith, “Hybrid partial-total type theory,” *Internat. J. Foundations Comput. Sci.* **6** (1995) 235–263.
 43. M. Tatsuta, “Program synthesis using realizability,” *Theoret. Comput. Sci.* **90** (1991) 309–353.
 44. M. Tatsuta, “Monotone recursive definition of predicates and its realizability inter-

- pretation,” in *Proc. 1st Internat. Conf. on Theoretical Aspects of Computer Software, Lecture Notes in Computer Science*, Vol. 526 (Springer, Berlin, 1991) pp. 38–52.
45. A. S. Troelstra and D. van Dalen, *Constructivism in Mathematics: An Introduction, Volumes I and II* (North-Holland, Amsterdam, 1988).
 46. R. Turner, “Reading between the lines in constructive type theory,” *J. Logic and Comput.* **7** (1997) 229–250.
 47. Y. Tsukada, “Open-endedness of objects and types in Martin-Löf’s type theory,” in *Proc. Workshop on Type Theory and its Application to Computer Systems, RIMS Lecture Notes*, No. 851 (Research Institute for Mathematical Sciences, Kyoto, 1993) pp. 102–126.
 48. Y. Tsukada, “Type-free equational reasoning in the theory of inductively defined types,” in *Proc. 3rd Fuji Internat. Symp. on Functional and Logic Programming*, eds. M. Sato and Y. Toyama (World Scientific, Singapore, 1998) pp. 227–246.