On The Type Structure of Standard ML

ROBERT HARPER
Carnegie-Mellon University
and
JOHN C. MITCHELL
Stanford University

Standard ML is a useful programming language with a polymorphic type system and a flexible module facility. One notable feature of the core expression language of ML is that it is *implicitly typed*: no explicit type information need be supplied by the programmer. In contrast, the module language of ML is *explicitly typed*; in particular, the types of parameters in parametric modules must be supplied by the programmer. We study the type structure of Standard ML by giving an explicitly-typed, polymorphic function calculus that captures many of the essential aspects of both the core and module language. In this setting, implicitly-typed core language expressions are regarded as a convenient short-hand for an explicitly-typed counterpart in our function calculus. In contrast to the Girard-Reynolds polymorphic calculus, our function calculus is *predicative*: the type system may be built up by induction on type levels. We show that, in a precise sense, the language becomes inconsistent if restrictions imposed by type levels are relaxed. More specifically, we prove that the important programming features of ML cannot be added to any impredicative language, such as the Girard-Reynolds calculus, without implicitly assuming a type of all types.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classifications—*applicative languages, ML*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*abstract data types, modules, packages*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*denotational semantics, operational semantics*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*type structure*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*lambda calculus and related systems*

General Terms: Languages, Theory

1. INTRODUCTION

Various forms of typed λ-calculus have become popular as theoretical models of programming languages. One motivation for studying these elemental

A preliminary version of this paper appeared in *Proceedings of the 15th ACM Symposium on Principles of Programming Languages*, 1988 under the title "The Essence of ML." The work of R. Harper was supported in part by the Science and Engineering Research Council of the United Kingdom, and in part by the Defense Advanced research Projects Agency, monitored by the Office of Naval Research under contract N00014-84-K-0415, ARPA Order No. 5404. The work of J. C. Mitchell was supported in part by an NSF PYI Award, matching funds from Digital Equipment Corporation, the Powell Foundation, and Zerox Corporation; NSF grant CCR-8814921 and the Wallace F. and Lucille M. Davis Faculty Scholarship.
Authors' addresses: R. Harper, Carnegie Mellon University, Pittsburgh, PA 15213; J. C. Mitchell, Stanford University, Stanford, CA 94305.

languages is that they provide some insight into programming languages with similar typing features. For example, the Girard-Reynolds second-order λ-calculus seems useful for analyzing languages with polymorphic functions or abstract data type declarations [15, 53, 47]. The richer type systems proposed by Martin-Löf [32], Constable [10], and Huet and Coquand [12] also provide formal logics for reasoning about programs. This general line of research has a different flavor from the original Scott-Strachey approach to programming language semantics, since the metalanguage of type theory reflects the type structure of the object languages studied. However, the long-term goals are the same: a precise understanding of programming language constructs and a sound mathematical basis for reasoning formally or informally about programs.

In "The Essence of Algol" [54], Reynolds presents a study of Algol-60 in the denotational style, contending that "Algol may be obtained from the simple imperative language by imposing a procedure mechanism based on a fully typed, call-by-name lambda calculus." In addition to testing the Scott-Strachey approach for programming language analysis, Reynolds' study gave an important picture of Algol as the composition of several independent constituents. Using the framework of type theory, we propose an analogous case study of the programming language Standard ML [19, 40], only the first steps of which are completed here. In this paper, we will describe a typed λ-calculus that encompasses many of the essential features of Standard ML and use this to analyze some potential extensions of the language. We have chosen Standard ML as the basis for this analysis because it is sufficiently well-developed to be interesting and useful as a "real" programming language, and sufficiently well-designed to support detailed analysis. In a sequel to this paper, we use the categorical techniques developed by Moggi [48] to refine the calculus presented here in a manner that clearly identifies the compile-time/run-time distinction in Standard ML [21].

Standard ML is an updated version of the programming "metalanguage" of the LCF system [17], comprising a core expression language with polymorphic functions [38] and a module language for defining interdependent program units [28]. The core language is designed around an automatic type inference algorithm that performs compile-time checking of "untyped" expressions. The module language is designed to support the organization of programs into separately-compilable units, and involves a moderate amount of explicit type information.

The main focus of this paper will be the type system of Standard ML. To simplify the presentation, we will omit exceptions and references; what is left is still quite interesting. The two main areas of investigation will be the discrepancy between implicitly- and explicitly typed frameworks, and the importance of separating the types into two distinct universes. With respect to the first point, we will argue that the implicitly-typed core language is profitably viewed as a short-hand for an explicitly-typed language. This simplifies the semantics of the language, since only well-typed expressions must have meaning, and allows us to study the implicitly-typed expression language within the same framework as the module language. It is worth noting that although the semantics is simplified, there seems to be no significant loss of generality in taking this point of view. We will see that Milner's type inference model, as described in Milner [37], and the ideal model of MacQueen et al. [30] may be viewed as models of our explicitly-typed core calculus.

An important feature of the analysis is that our type system is stratified into levels, or *universes*, in the style of Martin-Löf's type theory [32], and in keeping with the suggestions of MacQueen [28]. As in Martin-Löf's theory, our universes result in a *predicative* language, which means that the types may be ranked in such a way that every value occurs with higher rank than any values on which its existence or behavior is predicated. (For example, functions always occur at a higher rank than their arguments.) The universe distinctions are faithful to the separation of *monotypes* from *polytypes* in Milner's earlier work [37, 13], and allow us to show that implicit ML typing is syntactically equivalent to our explicit typing rules. The predicative universes also distinguish our calculus from both the implicit polymorphic typing of Mitchell [42], MacQueen et al. [30] and Cartwright [9] and the explicitly-typed polymorphic calculus of Girard [15, 16] and Reynolds [53]. In particular, the pure ML calculus without recursion has classical set-theoretic models, while the Girard-Reynolds calculus does not [56].

Some studies of ML typing (e.g., Cartwright [9], Mitchell [42] and MacQueen et al. [30]) have suggested, in effect, that the restrictions imposed by universes might be relaxed to allow the full second-order polymorphism of the Girard-Reynolds calculus [15, 16, 53]. However, these studies were generally based on consideration of the ML core language alone, and did not take modules into account. We will adopt the view of modules proposed by MacQueen, in which the main constructs are reduced to the Σ and Π types (the so-called "dependent" types) of Martin-Löf's type theory [29]. Using the typed λ-calculus with these constructs, we are able to show that universes play an important role.

Our examination of universes involves close study of a restricted subset of the language. In the fragment of Standard ML, without recursion or recursively-defined types, every expression evaluates to a *normal form*, regardless of the order of evaluation. (The fact that no evaluator could continue indefinitely is called the *strong normalization property*.) This is what one would naturally expect, since no construct explicitly provides unbounded search or recursion. However, we will show that if the distinction between universes is removed, it becomes possible to define a type of all types. It follows from previous work on *type*: *type*, specifically, Coquand [11], Girard [15], Howe [24] and Meyer and Reinhold [36], that there exist recursion-free programs that cannot be evaluated to a normal form by any evaluation strategy. As argued by Meyer and Reinhold [36], this alters the character of the language dramatically. In addition, Cardelli [6] argues that taking *type*: *type* has significant practical disadvantages because it eliminates the distinction between "compile time" and "run time" values. In particular, it is no longer possible to determine, without evaluating arbitrary expressions, whether a given expression denotes a type. This stands in the way of efficient compile-time type checking. Therefore, we believe that the separation of types into universes is essential to ML.

In this paper, we will not be concerned with evaluation order. The main reason is that for the fragment of ML without recursion or generative constructs, full evaluation in any order produces the same result. Consequently, our analysis of universes applies to both eager and lazy dialects of ML, and any similar language based on any other evaluation strategy. In fairness, we should emphasize that the relevance of *type*: *type* to programming remains a topic for further research. While it seems undesirable for a language to provide two distinct methods of recursion, one directly and one indirectly via *type*: *type*, we do not have clear-cut evidence that this is truly pathological. However, in further study of *type*: *type*, many subtle and important issues remain to be investigated. For example, we suspect that any study of representation independence [55, 45, 43] or full abstraction [52, 61] would be complicated dramatically by a type of all types.

The next section contains a short summary of the usual type inference rules for the core language of ML. In Section 3, an alternative, explicitly-typed core language is given. The two approaches are proved equivalent in Section 4, and the semantics of the core language is discussed in Section 5. Sections 6 through 9.3 consider a full calculus encompassing the module language. A review of modules is given in Section 6, followed by a reduction to Σ and Π types in Section 7. Section 8 considers the importance of universe distinctions and *type*: *type*. In Section 9 we give a brief overview of the type-theoretic treatment of various type declarations, and of the "sharing" constraints of MacQueen's module language. Concluding remarks appear in Section 10. All type systems are defined formally in tables at the end of the paper.

2. IMPLICITLY TYPED ML

Many studies of ML have focused on the type inference algorithm for the core expression language [37, 13, 30, 64] This algorithm allows the ML programmer to write, for

$$\text{let } id(\mathrm{x}) = \mathrm{x} \text{ in} \ldots$$

automatically inferring the fact that the function *id* is a function from type t to t, for any t. Milner's seminal paper [37] describes the type inference algorithm and proposes a semantic framework for justifying its behavior. In Milner's semantics, an untyped expression denotes some element of an untyped value space, and a type denotes a subset of this space. Types are therefore viewed as predicates expressing properties of untyped terms. One consequence of this view is that a given term can be assigned a variety of types; the type inference algorithm allows the programmer to enjoy the flexibility afforded by this semantics.

The syntactic part of Milner's analysis is refined in Damas and Milner [13] where an inference system for assigning types to expressions is given. The type inference rules are proved sound by showing that if it is possible to infer that expression e has type σ, then the untyped meaning of e belongs to the set denoted by σ. The type inference algorithm is then treated as a decision procedure for the inference system.

Milner's semantic analysis is elaborated in MacQueen et al. [30] and Cartwright [9], where the meanings of polymorphic types are clarified and recursive types are given semantics (see also Mitchell [42]). In Milner's model, the sets denoted by type expressions do not include a special error value of the domain, called *wrong*. Consequently, the soundness of ML typing is often summarized by the slogan *well-typed expressions cannot go wrong* [37].

Although there are quite a few constructs in the core expression language of ML, the behavior of the type checker may be understood by considering the fragment presented in Damas and Milner [13], which we will call *Core*-ML. The syntax of Core-ML is given by

$$e ::= x \mid ee \mid \lambda x.e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e,$$

where x may be any identifier. The let expression form is taken as primitive because it has a typing rule that is not derivable from the others. We will review the type inference algorithm for *Core*-ML briefly, so that we may later compare *Core*-ML with an explicitly-typed calculus.

We will use two classes of type expressions, *monomorphic* and *polymorphic*. In earlier work, monomorphic expressions have been the type expressions without variables, and the polymorphic expressions have had their type variables implicitly bound. Following Damas and Milner, we will make the same intuitive distinction in a slightly different way. We begin with some infinite set of type variables, an arbitrary collection of base types, and define the *monomorphic* type expressions (or *monotypes*) by

$$\tau ::= t \mid \beta \mid \tau \rightarrow \tau$$

where β may be any base type and t any type variable. In a monomorphic type expression τ, a type variable t stands for some unknown, monomorphic type. The *polymorphic* type expressions (also called *type schemes* or *polytypes*) are defined by

$$\sigma ::= \tau \mid \forall t.\sigma.$$

Intuitively, the elements of type $\forall t.\sigma$ a have type σ for every possible value of the type variable t (which will generally occur in σ). The constructor $\forall$ binds t in σ, so that $\forall t.\sigma = \forall s.[s/t]\sigma$, where $[s/t]\sigma$ denotes substitution of s for free occurrences of t in σ, with renaming of bound variables to avoid capture as usual. The purpose of the universal quantifier is to distinguish between *generic* type variables, which may be replaced by any monotype, and *ordinary* type variables, which denote a specific, but indeterminate, monotype. Note that every monotype is regarded as a polytype. This is merely a technical convenience, for one could equally well introduce an explicit "lifting" of monotypes, writing, say, $\Uparrow \tau$ for τ regarded as a polytype.

In the Damas-Milner system, the assertion that a *Core*-ML expression e has type σ is written $e: \sigma$. Since the type of an expression will generally depend on the types given to free ordinary variables, we will use typing statements that incorporate such assumptions. A *type assignment*, or *context*, Γ, is a finite set of *bindings* of the form $x: \sigma$, with no variable x occurring twice. It is useful to think of a context Γ as a partial function from variables to types and write $\Gamma(x)$ for the unique σ with $x: \sigma$ in Γ (if such a binding exists). We will also use the notation $\mathrm{Dom}(\Gamma)$ for the set of expression variables occurring in Γ. If Γ_1 and Γ_2 are type assignments with disjoint domains, we write Γ_1, Γ_2 for their union, $\Gamma_1 \cup \Gamma_2$. A special case is that we write $\Gamma, x: \sigma$ for $\Gamma \cup \{x: \sigma\}$, assuming $x \notin \mathrm{Dom}(\Gamma)$. A *typing* is a triple of the form $\Gamma \triangleright e: \sigma$, which may be read, "the expression e has type scheme σ in context Γ."

The Damas-Milner type assignment system is given in Table I. We write $\vdash_{\mathrm{DM}} \Gamma \triangleright e: \sigma$ if $\Gamma \triangleright e: \sigma$ is derivable in this system, and say that an expression e is *typable* if there is a context Γ and type scheme σ such that $\vdash_{\mathrm{DM}} \Gamma \triangleright e: \sigma$. We say e is Γ-typable if $\vdash_{\mathrm{DM}} \Gamma, \Gamma' \triangleright e: \sigma$ for some σ and some Γ' disjoint from Γ containing only monotypes. The reason for considering Γ-typings is that the types of let-bound variables in a program (closed expression) will be determined by their declarations. Therefore, in type-checking a term, polymorphic types of variables are determined by context but monotypes must be inferred.

A type σ is a *substitution instance* of σ' iff there is a substitution S of monotypes for type variables such that $S(\sigma') = \sigma$, where equality is modulo renaming of bound variables. A monotype τ is a *generic instance* of a polytype $\sigma = \forall t_1 \dots t_n . \tau'$, written $\sigma \sqsubseteq \tau$, iff there is a substitution S of monotypes for $t_1 \dots t_n$ such that $S(\tau') = \tau$. For polytypes, we say $\sigma \sqsubseteq \sigma'$ if every generic instance of σ' is also a generic instance of σ. Syntactically, this means that there is an α-variant $\forall s_1 \dots s_k . \tau'$ of σ' such that no $s_i (1 \le i \le k)$ occurs free in σ and $\sigma \sqsubseteq \tau'$; see Damas and Milner [13] and Mitchell [42] for further discussion, and Mitchell [42] for an interpretation of generic instantiation as semantic containment. When $\sigma \sqsubseteq \sigma'$, we say σ is *more general than* σ'. It is worth mentioning that the generic instance relation is preserved by substitution, i.e., if $\sigma \sqsubseteq \sigma'$, then $S(\sigma) \sqsubseteq S(\sigma')$ for any substitution S.

The following technical lemma summarizes some useful properties of the Damas-Milner system.

LEMMA 2.1 1. *If* $\vdash_{\mathrm{DM}} \Gamma \triangleright e: \sigma$ *then* $\vdash_{\mathrm{DM}} \Gamma' \triangleright e: \sigma$ *whenever* $\Gamma'(x) \sqsubseteq \Gamma(x)$ *for all* x *free in* e. 2. *If* $\vdash_{\mathrm{DM}} \Gamma \triangleright e: \sigma$ *and* S *is any substitution, then* $\vdash_{\mathrm{DM}} S(\Gamma) \triangleright e: S(\sigma)$.

An important property of the Damas-Milner system is that for every type assignment Γ, every Γ-typable expression has a "simplest" Γ-typing. A typing $\Gamma, \Gamma' \triangleright e: \sigma$ is Γ*-principal for expression* e if Γ' contains only monotypes, $\vdash_{\mathrm{DM}} \Gamma, \Gamma' \triangleright e: \sigma$ and, whenever $\vdash_{\mathrm{DM}} \Gamma, \Gamma'' \triangleright e: \sigma'$ for Γ'' containing only monotypes, we have a $\sigma \sqsubseteq \sigma'$ and $\Gamma'' \supseteq S\Gamma'$ for some substitution S of monotypes for type variables. In other words, the Γ-principle typing for e must be derivable, and it must give the most general type subject to the simplest association of monotypes to term variables not contained in Γ.

Table I. Damas-Milner Type Assignment

VAR	$\Gamma \rhd x : \sigma$	$(\Gamma(x) = \sigma)$
GEN	$\dfrac{\Gamma \rhd e : \sigma}{\Gamma \rhd e : \forall t.\sigma}$	*(t not free in Γ)*
SPEC	$\dfrac{\Gamma \rhd e : \sigma}{\Gamma \rhd e : \sigma'}$	$(\sigma \sqsubseteq \sigma')$
ABS	$\dfrac{\Gamma, x{:}\tau \rhd e' : \tau'}{\Gamma \rhd \lambda x.e' : \tau \rightarrow \tau'}$	$(x \notin \mathrm{Dom}(\Gamma))$
APP	$\dfrac{\Gamma \rhd e : \tau' \rightarrow \tau \quad \Gamma \rhd e' : \tau'}{\Gamma \rhd ee' : \tau}$	
LET	$\dfrac{\Gamma \rhd e : \sigma \quad \Gamma, x{:}\sigma \rhd e' : \tau'}{\Gamma \rhd \mathtt{let}\, x = e\, \mathtt{in}\, e' : \tau'}$	$(x \notin \mathrm{Dom}(\Gamma))$

(Variations and further details may be found in Damas and Milner [13] and Mitchell [44].)

THEOREM 2.2 (DAMAS-MILNER). *If an untyped Core-ML expression e is Γ-typable, then there exists a Γ-principal typing for e. Furthermore, there is an algorithm which, given e and Γ, computes the Γ-principle typing if it exists, and fails otherwise.*

The algorithm described in this theorem works by recursively computing the principal typings of subterms. If a variable x is let-bound, as in let $x = e$ in e', the principal type for e is computed and then added to the type assignment when typing e'. In general, the type of a let-bound variable may be a polytype, with one or more type quantifiers. When a λ-bound variable is encountered, as in $\lambda x.e$, it is not possible to determine the appropriate type for x before typing the function body e. However, we only want a monotype for a λ-bound variable. Therefore, the algorithm is designed to determine the most general monotypes for free variables as well as the types of terms.

For the important special case of closed *Core*-ML expressions, Theorem 2.2 implies that every closed term has a single most general type. It is shown in Kanellakis et al. [25] and Kfoury et al. [26] that any algorithm which decides whether an untyped *Core*-ML term has a type necessarily requires exponential time for infinitely many terms. It follows that computing the principal type of a *Core*-ML term requires exponential time.

3. EXPLICITLY TYPED ML

In contrast to the Milner-style analyses of ML, we will view ML programs as being *explicitly* typed in the sense that a given term has at most one type in any given context (modulo type equality). To achieve this, we will modify the syntax of terms to include explicit type information. In particular, a type is assigned to the bound variable of a λ-abstraction at the point where it is bound, and the type abstraction and instantiation associated with polymorphism are made explicit. We view the untyped concrete syntax of ML as a convenient shorthand for an explicitly-typed abstract syntax, with the type inference algorithm bridging the gap. The main reason for taking this position is that the implicitly-typed approach does not scale up to the full language. For example, ML includes type constraints, type definitions, and an explicitly-typed modules language.

In addition, we choose to view ML as an explicitly-typed language because it provides a better basis for studying equational properties of the language such as representation independence and full abstraction. (For discussion of these topics, see Reynolds [55] and Stoughton [61].) Viewing ML as an explicitly-typed language also leads to technical simplifications in the semantics of the language. Without explicit typing, the semantics would become somewhat more complicated, since we would need a "universal domain"-like interpretation for untyped lambda abstraction, type abstraction, and type constructors like $\rightarrow$ and Π. Moreover, a semantics for an implicitly-typed language would entail identifying any two expressions that are equal as untyped terms. It is worth remarking that there is no semantic loss of generality in focusing on the explicitly-typed language, since models of the implicitly-typed system, such as Milner's original domain interpretation [37] and related structures [30], give rise to models of the explicitly-typed system in a natural way. (See Section 5.)

We therefore introduce an explicitly-typed function calculus, called *Core*-XML, for *core explicit ML*. This calculus is essentially equivalent to *Core*-ML, the implicitly-typed language presented above. The types of *Core*-XML fall into two classes, corresponding to the monomorphic and polymorphic types of *Core*-ML. To introduce some useful terminology, we will say that τ is a *type of the first universe*, and write $\tau : U_1$, if τ is built up from base types and type variables using the function-space constructor $\rightarrow$. This means that $\tau : U_1$ iff τ is a monomorphic type expression of *Core*-ML.

The polymorphic type expressions of *Core*-ML quantify over the monomorphic types. In *Core*-XML this corresponds to universal quantification over the first universe, and so it is natural to regard these types as being of a "higher" second universe. We will say that σ is a *type of the second universe*, and write $\sigma : U_2$, if σ has the form $\Pi t_1 : U_1 \ldots \Pi t_n : U_1 . \tau$, where $\tau : U_1$. Thus U_2 consists of exactly the *Core*-ML polymorphic types, except that we will write Π instead of $\forall$, and the universe of each type variable is written explicitly. This is to allow a smooth generalization to full XML with type variables ranging over both universes, a step we will take in Section 7. As a matter of convenience, we follow Damas and Milner and consider every monomorphic type to be a polymorphic type, and hence we effectively have $U_1 \subseteq U_2$ identification, and introduce to U_2.

The presentation of *Core*-XML is simplified by adopting the meta-variable conventions used in the previous section, so that $\tau, \tau_1, \ldots$ will always be U_1 types, and $\sigma, \sigma_1, \ldots$ will be U_2 types. We will not explicitly declare type variables in contexts. Rather, we assume at the outset that all type variables $r, s, t, \ldots$ denote elements of U_1.

The unchecked *preterms* of explicitly typed *Core*-XML are given by the grammar,

$$M ::= x \mid MN \mid \lambda x\colon \tau . M \mid M[\tau] \mid \lambda t. M \mid \mathsf{let}\ x\colon \sigma = M\ \mathsf{in}\ N,$$

where metavariables M and N range over preterms. In this grammar, x may be any term variable, MN is the application of function M to argument N, $\lambda x\colon \tau . M$ denotes the function defined by treating M as a function of variable x, $M[\tau]$ is the application of polymorphic function M to type argument τ, and $\lambda t. M$ is the polymorphic function obtained by treating M as a function of type variable t. Following Damas-Milner, we retain let as a primitive construct in *Core*-XML since its typing rule is not derivable from the other rules. The value of expression let $x\colon \sigma = M$ in N is the value of N when x is given the value of M. In the full XML system, let will be definable in terms of abstraction and application, and therefore will be eliminated.

The type checking rules of the language are listed in Table II. To distinguish implicit *Core*-ML typing from explicit *Core*-XML typing, we will write $\vdash_X \Gamma \rhd \mathrm{M}\colon \sigma$ if the typing $\Gamma \rhd \mathrm{M}\colon \sigma$ is derivable from the *Core* XML typing rules. The essential difference between $\vdash_{DM}$ and $\vdash_X$ is that the GEN and SPEC rules of the implicit system are replaced by rules for explicit type abstraction and type application. A preterm M is a *term* of *Core*-XML if $\vdash_X \Gamma \rhd M\colon \sigma$ for some Γ and σ.

The difference between *Core*-XML and the Girard-Reynolds polymorphic λ-calculus [5, 53, 15] lies in the distinction between universes U_1 and U_2. Rule TAPP of *Core*-XML only allows a type application $\Gamma \rhd \mathrm{M}[\tau]\colon [\tau/t]\sigma$ when τ is a type of the first universe U_1. However, in the Girard-Reynolds calculus, there is no universe distinction, and we can apply a term of polymorphic type to any type. One consequence of the universe distinction is that *Core*-XML has classical set-theoretic models, while the Girard-Reynolds calculus does not [56].

The language *Core*-XML is closely related to several explicitly-typed function calculi, in particular Martin-Löf's intensional type theory [31], the AUTOMATH languages [14], the Calculus of Constructions [12], and the type theory of LF [18]. Since *Core*-XML is based on a predicative notion of universe, it is most closely related to Martin-Löf's early type theories, except that we do not, at this stage, take $U_1\colon U_2$. It is worth remarking that Martin-Löf's later type theory [32, 33], and the NuPRL type theory [10], are type assignment systems, and hence are more closely related to the type system defined by Damas and Milner for the study of ML.

Table II. *Core*-XML Type System

VAR	$\Gamma \rhd x : \sigma$	$(\Gamma(x) = \sigma)$
TABS	$\dfrac{\Gamma \rhd M : \sigma}{\Gamma \rhd \lambda t.M : \Pi t{:}U_1.\sigma}$	*(t not free in* Γ*)*
TAPP	$\dfrac{\Gamma \rhd M : \Pi t{:}U_1.\sigma}{\Gamma \rhd M[\tau] : [\tau/t]\sigma}$	
ABS	$\dfrac{\Gamma, x{:}\tau \rhd M' : \tau'}{\Gamma \rhd \lambda x{:}\tau.M' : \tau \rightarrow \tau'}$	$(x \notin \mathrm{Dom}(\Gamma))$
APP	$\dfrac{\Gamma \rhd M : \tau' \rightarrow \tau \quad \Gamma \rhd M' : \tau'}{\Gamma \rhd MM' : \tau}$	
LET	$\dfrac{\Gamma \rhd M : \sigma \quad \Gamma, x{:}\sigma \rhd M' : \tau'}{\Gamma \rhd \mathtt{let}\, x{:}\sigma = M \,\mathtt{in}\, M' : \tau'}$	$(x \notin \mathrm{Dom}(\Gamma))$

4. EQUIVALENCE OF EXPLICIT AND IMPLICIT SYSTEMS

In this section, we will show that implicitly-typed *Core*-ML and explicitly-typed *Core*-XML are essentially equivalent. A related correspondence between implicit ML typing and Girard-Reynolds typing restricted by "rank," which is similar to our universe restriction, was suggested earlier in [27, Section 7]. However, the statement of Theorem 7.1 in that paper is incorrect, since rank 2 typing of lambda terms allows us to type λ-abstractions polymorphically, whereas the typing rules of *Core*-ML do not. It is to avoid precisely this problem that we have included let in the syntax of *Core*-XML.

The *type erasure* M° of an explicitly typed term M is defined as follows:

$$M^\circ = \begin{cases} x & \text{if } M \equiv x \\ M_1^\circ M_2^\circ & \text{if } M \equiv M_1 M_2 \\ \lambda x.M_1^\circ & \text{if } M \equiv \lambda x{:}\,\tau.M_1 \\ M_1^\circ & \text{if } M \equiv \lambda t.M_1 \\ M_1^\circ & \text{if } M \equiv M_1[\,\sigma\,] \\ \text{let } x = M_1^\circ \text{ in } M_2^\circ & \text{if } M \equiv \text{let } x{:}\,\sigma = M_1 \text{ in } M_2. \end{cases}$$

THEOREM 4.1 *If* $\vdash_X \Gamma \rhd M{:}\,\sigma$, *then* $\vdash_{DM} \Gamma \rhd M^\circ{:}\,\sigma$.

PROOF. The proof is by induction on derivations. The only nontrivial case is the TAPP rule, where the theorem follows from the fact that $\forall t.\sigma \sqsubseteq [\tau/t]\sigma$ for any type τ. □

THEOREM 4.2 *If* $\vdash_{\mathrm{DM}} \Gamma \rhd e\colon \sigma$, *then there exists an explicitly typed term* M *such that* $M^\circ \equiv e$ *and* $\vdash_{\mathrm{X}} \Gamma \rhd M\colon \sigma$. *Furthermore,* M *can be computed efficiently from a proof of* $\Gamma \rhd e\colon \sigma$.

PROOF. We use induction on typing derivations.

VAR Take M to be x.

GEN By the induction hypothesis there exists an N such that $N^\circ \equiv e$ and $\vdash_{\mathrm{X}} \Gamma \rhd N\colon \sigma$. Since t does not occur free in Γ, we may apply TABS to obtain $\vdash_{\mathrm{X}} \Gamma \rhd \lambda t.N\colon \forall t.\sigma$. Take M to be $\lambda t.N$, and observe that $M^\circ \equiv N^\circ \equiv e$.

SPEC By the induction hypothesis there exists an N such that $N^\circ \equiv e$ and $\vdash_{\mathrm{X}} \Gamma \rhd N\colon \sigma$, with $\sigma \sqsubseteq \sigma'$. We know that σ has the form $\sigma \equiv \forall t_1 \dots t_n.\tau$, and similarly for σ'. Choose an α-variant $\forall s_1 \dots s_k.\tau'$ of σ' such that no $s_i (1 \le i \le k)$ occurs free in σ or Γ. Then there is a substitution S acting on $t_1 \dots t_n$ such that $S(\tau) = \tau'$. By n applications of TAPP followed by k applications of TABS, we obtain

$$\vdash_{\mathrm{X}} \Gamma \rhd \lambda s_1 \dots s_k.N\tau_1 \dots \tau_n\colon \forall s_1 \dots s_k.[\tau_1 \dots \tau_n/t_1 \dots t_n]\tau$$

where $\tau_i = S(t_i)$ $(1 \le i \le n)$. But by the choice of S, this is

$$\vdash_{\mathrm{X}} \Gamma \rhd \lambda s_1 \dots s_k.N\tau_1 \dots \tau_n\colon \sigma'.$$

Take $M \equiv \lambda s_1 \dots s_k.N\tau_1 \dots \tau_n$, and observe that $M^\circ \equiv N^\circ \equiv e$.

ABS By induction we have $\vdash_{\mathrm{X}} \Gamma[x\colon \tau] \rhd N\colon \tau'$ for some N such that $N^\circ \equiv e$, and thus $\vdash_{\mathrm{X}} \Gamma \rhd \lambda x\colon \tau.N\colon \tau \to \tau'$. Take M to be $\lambda x\colon \tau.N$.

APP By induction we have $\vdash_{\mathrm{X}} \Gamma \rhd N\colon \tau' \to \tau$ and $\vdash_{\mathrm{X}} \Gamma \rhd P\colon \tau'$ for some N and P such that $N^\circ \equiv e$ and $P^\circ \equiv e'$. Take $M \equiv NP$, and observe that $M^\circ \equiv N^\circ P^\circ \equiv ee'$ and $\vdash_{\mathrm{X}} \Gamma \rhd M\colon \tau$, as desired.

LET By induction $\vdash_{\mathrm{X}} \Gamma \rhd N\colon \sigma$ and $\vdash_{\mathrm{X}} \Gamma[x\colon \sigma] \rhd P\colon \tau'$ with $N^\circ \equiv e$ and $P^\circ \equiv e'$. Take $M \equiv \mathsf{let}\ x\colon \sigma = N\ \mathsf{in}\ P$, and observe that $M^\circ \equiv \mathsf{let}\ x = N^\circ\ \mathsf{in}\ P^\circ \equiv \mathsf{let}\ x = e\ \mathsf{in}\ e'$, and $\vdash_{\mathrm{X}} \Gamma \rhd M\colon \tau'$. □

Theorem 2.2 (from Damas and Milner [13]) states that there is an algorithm which finds, for any Γ-typable expression e, a Γ-principal typing for e. It is a simple matter to modify this algorithm so that it also produces a derivation of the principal typing in the Damas-Milner system. Applying Theorem 4.2, we obtain an algorithm X that, given a Γ-typable expression e, yields an explicitly typed term M such that $\vdash_{\mathrm{X}} \Gamma, \Gamma' \rhd M\colon \sigma$ (and fails otherwise). Algorithm X inserts type labels on λ's and let's, type abstractions on all let-bound expressions, and type applications at all uses of identifiers whose type is of the form $\forall t_1 \dots t_n.\tau$. For example, the explicitly-typed term produced from

$$\mathsf{let} I = \lambda x.x\ \mathsf{in}\ II$$

is

$$\lambda t.\mathsf{let}\ I\colon \Pi t.t \to t = \lambda t.\lambda x\colon t.x\ \mathsf{in}\ I[t \to t](I[t]).$$

Note that the principal type of let $I = \lambda x.x$ in II is $\forall t.t \rightarrow t$, and $\prod t.t \rightarrow t$ is the type of the explicitly-typed term.

5. SEMANTICS OF CORE-XML

5.1 Introduction

The *Core*-XML language has a straightforward Henkin-style model theory that is similar to the semantics of second-order lambda calculus described in Bruce and Meyer [4], Bruce et al. [5] and Mitchell [42], except that we have two universes instead of one collection of types. Categorical semantics may also be developed along the lines of Moggi [48] and Harper et al. [21], which resemble the indexed-categorical frameworks of Seely [58, 59]. However, we will not discuss categorical semantics in this paper. We will summarize some basic ideas regarding Henkin-style models primarily to emphasize that there is a semantic connection between *Core*-ML and *Core*-XML, as well as a syntactic one. In particular, structures such as the so-called *ideal model* of Milner [37] and MacQueen et al. [30] provide models of *Core*-XML. A tangential reason to consider the semantics of *Core*-XML is that when U_1 and U_2 are isomorphic, we have a model of the impredicative Girard-Reynolds calculus (see Mitchell [44]). Therefore, the semantics of *Core*-XML may be considered more basic than the semantics of the Girard-Reynolds calculus.

An interesting choice in giving semantics to *Core*-XML lies in the interpretation of the containment $U_1 \subseteq U_2$. While it seems syntactically simpler to view every element of U_1 as an element of U_2, there may be some semantic advantages of interpreting $U_1 \subseteq U_2$ as meaning that U_1 may be embedded in U_2. With appropriate assumptions about the inclusion mapping from U_1 to U_2, this seems entirely workable, and leads to a more flexible model definition than literal set-theoretic interpretation of $U_1 \subseteq U_2$.

5.2. Model Definition

Since *Core*-XML has two collections of types, U_1 and U_2, with $U_1 \subseteq U_2$, a *Core*-XML model $\mathscr{A}$ will have two sets $U_1^{\mathscr{A}}$ and $U_2^{\mathscr{A}}$ with $U_1^{\mathscr{A}} \subseteq U_2^{\mathscr{A}}$. For each element $a \in U_2^{\mathscr{A}}$, we will also have a set Dom^a elements of type a. (Since $U_1^{\mathscr{A}} \subseteq U_2^{\mathscr{A}}$, this also gives us a set Dom^a for each $a \in U_1^{\mathscr{A}}$.) In addition, we need some way to interpret type expressions with $\rightarrow$ and $\forall$ as elements of U_1 or U_2, and some machinery to interpret function application. The following model definition is in the same spirit as Bruce et al. [5].

A *frame* $\mathscr{A}$ *for Core*-XML is a tuple

$$\mathscr{A} = \langle \mathscr{U}, \text{dom}, \{\Phi_{a,b}, \Psi_f\}, \mathscr{I} \rangle,$$

where

—$\mathscr{U} = \{U_1^{\mathscr{A}}, U_2^{\mathscr{A}}, [U_1^{\mathscr{A}} \rightarrow U_2^{\mathscr{A}}], \rightarrow^{\mathscr{A}}, \prod^{\mathscr{A}}\}$ specifies sets $U_1^{\mathscr{A}} \subseteq U_2^{\mathscr{A}}$, a set $[U_1^{\mathscr{A}} \rightarrow U_2^{\mathscr{A}}]$ of functions from $U_1^{\mathscr{A}}$ to $U_2^{\mathscr{A}}$, a binary operation $\rightarrow^{\mathscr{A}}$ on $U_1^{\mathscr{A}}$ and a map $\prod^{\mathscr{A}}$ from $[U_1^{\mathscr{A}} \rightarrow U_2^{\mathscr{A}}]$ to $U_2^{\mathscr{A}}$.

—dom $= \{Dom^a | a \in U_2^{\mathscr{A}}\}$ is a collection of sets indexed by types,

—$\{\Phi_{a,b}, \Psi_f\}$ is a collection of functions, with one $\Phi_{a,b}$ for every pair of types $a, b \in U_1^{\mathscr{A}}$ from the first universe, and one Ψ_f for every function $f \in [U_1^{\mathscr{A}} \to U_2^{\mathscr{A}}]$ mapping the first universe to the second.
Each $\phi_{a,b}$ must be a bijection

$$\Phi_{a,b}: DOM^{a \to b} \to [Dom^a \to Dom^b]$$

between $Dom^{a \to b}$ and some collection $[Dom^a \to Dom^b]$ of functions from Dom^a to Dom^b. Similarly, each Ψ_f must be a bijection

$$\Psi_f: Dom^{\Pi f} \to \left[\prod_{a \in U_1} \cdot Dom^{f(a)} \right]$$

between $Dom^{\Pi f}$ and some subset $[\Pi_{a \in U_1} \cdot \mathrm{Dom}^{f(a)}]$ of the cartesian product $\Pi_{a \in U_1} \cdot Dom^{f(a)}$.

—$\mathscr{I}$: *Constants* $\to \cup_{a \in U_2} Dom^a$ assigns a value to each constant symbol, with $\mathscr{I}(c) \in Dom_{[\tau]}$ if c is a constant of type τ.[1]

This concludes the definition.

If $\mathscr{A}$ is an *Core*-XML frame, then an $\mathscr{A}$-environment is a mapping

$$\eta: Variables \to \left(U_1^{\mathscr{A}} \cup \bigcup_{a \in U_2^{\mathscr{A}}} Dom^a \right)$$

such that for every type variable t, we have $\eta(t) \in U_1^{\mathscr{A}}$. The meaning $[\![\sigma]\!]\eta$ of any type expression σ in environment η is straightforward, as presented by Bruce et al. [5].

If Γ is a context, then η *satisfies* Γ, written $\eta \models \Gamma$, if $\eta(x) \in Dom^{[\sigma]}$ for every $x: \sigma \in \Gamma$. The meaning of a term $\vdash_X \Gamma \rhd M: \sigma$ in environment $\eta \models \Gamma$ is defined by induction as follows.

$$\begin{aligned}
[\![\Gamma \rhd x: \sigma]\!]\eta &= \eta(x) \\
[\![\Gamma \rhd MN: \tau]\!]\eta &= \Phi_{a,b}[\![\Gamma \rhd M: \sigma \to \tau]\!]\eta[\![\Gamma \rhd N: \sigma]\!]\eta, \\
&\quad \text{where } a = [\![\sigma]\!]\eta \text{ and } b = [\![\tau]\!]\eta \\
[\![\Gamma \rhd \lambda x: \sigma.M: \sigma \to \tau]\!]\eta &= \Phi_{a,b}^{-1}(f) \text{ where } f \text{ is the function} \\
&\quad f(d) = [\![\Gamma, x: \sigma \rhd M: \tau]\!]\eta[d/x] \text{ all } d \in Dom^a, \\
&\quad a = [\![\sigma]\!]\eta \text{ and } b = [\![\tau]\!]\eta \\
[\![\Gamma \rhd M\tau: [\tau/t]\sigma]\!]\eta &= \Psi_f [\![\Gamma \rhd M: \Pi t.\sigma]\!]\eta[\![\tau]\!]\eta, \\
&\quad \text{where } f(a) = [\![\sigma]\!]\eta[a/t] \text{ all } a \in U_1^{\mathscr{A}},
\end{aligned}$$

[1] Each constant of the language must be given a type. The type associated with any constant must be a closed type expression (without free variables), so that the semantic type of the constant symbol is independent of the environment.

$$[\![\Gamma \rhd \lambda t{:}\,U_1.M{:}\,\Pi t.\sigma]\!]\eta = \Psi_f^{-1}(g) \text{ where } f \text{ and } g \text{ are the functions}$$

$$g(a) = [\![\Gamma \rhd M{:}\,\sigma]\!]\eta[a/t] \text{ all } a \in U_1^{\mathscr{A}}, \text{ and}$$

$$f(a) = [\![\sigma]\!]\eta[a/t] \text{ all } a \in U_1^{\mathscr{A}}.$$

A *Core*-XML frame is an *environment model* if $[\![\Gamma \rhd M{:}\,\sigma]\!]\eta$ exists, as defined above, for every well-typed term $\Gamma \rhd M{:}\,\sigma$ and every $\eta \models \Gamma$. For further discussion of this style of environment model definition, see Bruce et al. [5].

5.3. Equational Soundness and Completeness

Equations have the form $\Gamma \rhd M = N{:}\,\sigma$, where M and N are terms of type σ (in the context Γ). The equational proof system of *Core*-XML is similar to that of the Girard-Reynolds calculus [16, 53, 5], with the following additional axiom for let:

$$\Gamma \rhd (\mathsf{let}\ x{:}\,\sigma = M \ \mathsf{in}\ N) = [N/x]M{:}\,\tau.$$

A complete presentation of the equational system for *Core*-XML is omitted here since it is an obvious fragment of the equation calculus for XML (which is presented below.)

It is easy to show that in every model, the meaning of each term has the correct semantic type.

LEMMA 5.1 (TYPE SOUNDNESS). *Let $\mathscr{A}$ be a Core-XML model, $\Gamma \rhd M{:}\,\sigma$ a well-typed term, and $\eta \models \Gamma$ and environment. Then*

$$[\![\Gamma \rhd M{:}\,\sigma]\!]\eta \in Dom^{[\![\sigma]\!]\eta}$$

In addition, the methods of Bruce et al. [5] may be used to show that the equational proof system is sound and complete for models that do not have empty types. We also expect that the methods of Meyer et al. [35] may be used to prove equational completeness for models that may have empty types, and that the approach of Mitchell and Moggi [46] will yield a completeness theorem for Kripke-style models, without making any assumption about type inhabitation.

5.4 Examples of Models

Since the only difference between *Core*-XML and the Girard-Reynolds second-order calculus is the distinction between universes, every second-order model may be viewed as a *Core*-XML model with $U_1 = U_2$. Consequently, *Core*-XML may be interpreted in the domain-theoretic and recursion-theoretic models discussed by Amadio et al. [1], Bruce et al. [5], Girard [15], Troelstra [62], McCracken [34] and Mitchell [41]. One difference between the languages, however, is that *Core*-XML has classical set-theoretic models, while the Girard-Reynolds calculus does not [56]. In fact, any model of ordinary (nonpolymorphic) typed lambda calculus may be extended to a model of *Core*-XML by a simple set-theoretic construction.

5.4.1 *Set-Theoretic Models.* If we begin with some model $\mathscr{A} = \langle U_1^{\mathscr{A}}, \rightarrow^{\mathscr{A}}, \{Dom^a | a \in U_1^{\mathscr{A}}\}, \mathscr{I} \rangle$ for the U_1 types and terms, we can extend this to a model for all of *Core*-XML using standard set-theoretic cartesian product. For any ordinary α, we define the set $[U_2]_\alpha$ as follows

$$[U_2]_0 = U_1,$$

$$[U_2]_{\beta+1} = [U_2]_\beta \cup \left\{ \prod_{a \in U_1} f(a) | f: U_1^{\mathscr{A}} \rightarrow [U_2]_\beta \right\},$$

$$[U_2]_\alpha = \bigcup_{\beta < \alpha} [U_2]_\beta \text{ for limit ordinal } \alpha.$$

Note that for any α, the set $[U_2]_\alpha$ contains all cartesian products indexed by functions from

$$[U_1 \rightarrow U_2]_\alpha = \bigcup_{\beta < \alpha} U_1 \rightarrow [U_2]_\beta.$$

The least limit ordinal ω actually gives us a model. The reason for this is that every U_2 type expression σ of *Core*-XML is of the form $\sigma \equiv \Pi t_1 : U_1 \ldots \Pi t_k : U_1 . \tau$ for some $\tau : U_1$. It is easy to show that any type with k occurrences of Π has a meaning in $[U_2]_k$. Consequently, every type expression has a meaning in $[U_2]_\omega$. This is proved precisely in the lemma below. To shorten the statement of the lemma, we let $\mathscr{A}_n$ be the structure obtained from a U_1 model $\mathscr{A}$ by taking

$$[U_1 \rightarrow U_2]_n = \bigcup_{k < n} U_1 \rightarrow [U_2]_k$$

and $U_2 = [U_2]_n$

LEMMA 5.2. *Let $\Gamma \triangleright M: \sigma$ be any well-typed Core-XML expression, with $\sigma \equiv \Pi t_1 \ldots \Pi t_k . \tau$, and such that in the derivation of $\Gamma \triangleright M: \sigma$, every type of every subterm has no more that n occurrences of the quantifier Π. Then for any environment η mapping variables into a U_1 structure $\mathscr{A}$, the meaning $[\![\Gamma \triangleright M: \sigma]\!]\eta$ exists and is well-defined in the structure $\mathscr{A}_n$.*

The lemma is proved by first showing that every $\sigma \equiv \Pi t_1 \ldots \pi t_k . \tau$ with $k < n$ has a meaning in $\mathscr{A}_n$, and then using induction on terms to prove the lemma.

While stage ω yields a model in which $\Pi t: U_1 . \sigma$ is interpreted as ordinary set-theoretic cartesian product over U_1, the set of functions $[U_1 \rightarrow U_2]_\omega$ is not necessarily all set-theoretic functions from U_1 to $U_2 = [U_2]_\omega$. In order to get a truly full set-theoretic model, we may have to consider much larger ordinals. If we assume the existence of an inaccessible cardinal, then induction up to any inaccessible cardinal that is larger than the cardinality of any set in the given U_1 model, including U_1, yields a full set-theoretic model.

5.4.2 *Partial Equivalence Relation Models.* One class of models that is pertinent to the development of the last few sections is obtained by interpreting types as *partial equivalence relations* (PER's) on an applicative structure (see Mitchell [41] for further discussion and references). The ideal model of

MacQueen et al. [30], for example, can be viewed as a PER inference model, as defined in Mitchell [42], by replacing each ideal I with the partial equivalence relation $I \times I$. By the results of Mitchell [41], this gives us a second-order model, and hence a model of *Core*-XML. A similar *Core*-XML model can be constructed from Milner's original description [37], taking U_1 to be the collection of monotypes, and defining the elements of U_2 (the polytypes) by quantification over U_1. In either case, we obtain a *Core*-XML model with a degenerate equational theory (all terms of the same type become equal), but type membership interpreted as expected. Thus a consequence of the type soundness theorem for *Core*-XML models (e.g., see Mitchell [41, 42]) is that *Core*-XML expressions "cannot go wrong." Since the details are essentially straightforward, given the techniques of Mitchell [41, 44], for example, we leave the precise construction to the reader.

5.5 Coherence and the Semantics of *Core*-ML

Since we view the syntax of implicitly-typed *Core*-ML as an abbreviation for explicitly-typed *Core*-XML, a natural way to give semantics to *Core*-ML is by inserting type information into terms, and giving semantics to the resulting *Core*-XML expressions in the models described in this section. While this may seem straightforward, there is one subtle issue that must be brought out. This is the problem of *coherence*: since there are many ways to insert type information into an implicitly typed term, we must ask whether the meaning of an implicitly typed *Core*-ML term is uniquely determined. In other words, do we get different semantic interpretations depending on the way we assign types to subexpressions? A well-taken criticism of our approach, on these grounds, is Ohori [50]. A paper by Breazu-Tannen [3] also discusses the general issue of coherence.

To avoid confusion, we will make a few definitions. We say a function R from *Core*-ML to *Core*-XML is a *type reconstruction function* if, for every $\vdash_{\mathrm{DM}} \Gamma \rhd e\colon \tau$, we have $R(\Gamma \rhd e\colon \tau) = \Gamma \rhd M\colon \tau$ with $\vdash_{\mathrm{X}} \Gamma \rhd M\colon \tau$ and $M^{\circ} = e$, where M° is the type erasure of M defined in Section 4. If $\mathscr{R}$ is a set of reconstruction functions, then an *$\mathscr{R}$-meaning of a Core-ML term $\Gamma \rhd e\colon \tau$ in a Core-XML model $\mathscr{A}$* is the meaning of $R(e)$ in $\mathscr{A}$, for some $R \in \mathscr{R}$. We say that a model $\mathscr{A}$ and set $\mathscr{R}$ of reconstruction functions are *coherent* if all $\mathscr{R}$ meanings of a *Core*-ML term in $\mathscr{A}$ are identical. A general goal in giving a meaning to *Core*-ML terms in a *Core*-XML model is that the meanings should be $\mathscr{R}$ coherent, for some reasonable class $\mathscr{R}$ of reconstruction functions.

For the pure languages of *Core*-ML and *Core*-XML described in this paper, it seems natural to prefer models that are coherent for the class of all type reconstruction functions. Among the examples given in Section 5, the partial equivalence relation models are coherent for all type reconstruction functions, but set-theoretic models and models of the Girard-Reynolds second-order lambda calculus may not be. An example illustrating the failure of coherence for terms that have free polymorphic variables is given in Ohori [50]. However, it is not hard to show that coherence holds for all closed terms in all models, using the strong normalization property of reduction (see Barendregt [2] and Mitchell [44].

A caveat in future work on ML is that when we include features such as recursion, references and exceptions, it is important to consider the order of evaluation. Since the evaluation order used in ML is eager, or call-by-value, some equational principles that are sound for the *Core*-XML models described in this section will fail. Since the coherence or lack of coherence of a model depends on its equational theory, we cannot expect that the semantic interpretation of implicitly typed terms in an explicitly typed way will be coherent for arbitrary type reconstruction functions. A particular example that may be telling for those familiar with ML is a term of the form

$$\mathsf{let}\ x = ref\,nil\ \mathsf{in} \ldots 3::(!x) \ldots 5::(!x) \ldots$$

with all occurrences of x used at the same type. (The expression $y::(!x)$ is ML notation for the list obtained by adding y to the front of the list stored in reference cell x.) If we assume that $nil: \forall t.list(t)$ is a polymorphic list constant and $ref: \forall t.t \rightarrow ref(t)$ creates a reference of any type, then the let-bound variable x in this term may have type $\forall t.ref(list(t))$ or any instance of this polymorphic type. Since the only type that x must have is $ref(list(int))$, we have type reconstructions of the following forms:

$$\mathsf{let}\ x: \forall t.ref(list(t)) = \lambda t.ref\,(list(t))(nil\,t)\mathsf{in} \ldots$$
$$3::(!(x\,int)) \ldots 5::(!(x\,int)) \ldots$$
$$\mathsf{let}\ x: ref(list(int)) = (ref\,list(int))(nil\,int)\mathsf{in} \ldots 3::(!x) \ldots 5::(!x) \ldots$$

If we interpret call-by-value so that an expression beginning with λt is not evaluated until it is applied to an argument, then these two expressions have very different meaning: in the first, two list cells are created, and in the second, only one. Since no sensible equational theory would identify these two programs, it is not possible to have a coherent semantics for any class of reconstruction functions that allows both of these possibilities.

6. THE ML MODULE LANGUAGE

In this section we briefly review the organization of the Standard ML module system [28, 39, 40]. The basic entities of the Standard ML module system are *structures*, *signatures* and *functors*. Roughly speaking, a structure is a packaged environment, assigning types to type identifiers, values to value identifiers, and structures to structure identifiers. Signatures are a form of "type" or "interface" for a structure, specifying type information for each of the components of the structure. If a structure satisfies the description given in a signature (in a sense to be outlined below), the structure is said to "match" that signature; a given structure will, in general, match a variety of distinct signatures. Functors are functions mapping structures to structures. Since ML does not support higher-order functors (i.e., functors taking functors as arguments or yielding functors as results), there is no need for functor signatures.

Structures are denoted by structure expressions, the basic form of which is a sequence of declarations delimited by keywords struct and end. Structures are not "first-class" in that they may only be bound to structure identifiers or passed as arguments to functors. We will see that this a universe distinction, and not an *ad hoc* restriction of the language. The following declaration binds a structure to the identifier S:

```
structure S =
  struct
    type t = int
    val x: t = 3
  end
```

The structure expression following the equals sign defines an environment mapping t to int and x to 3, and binds this environment to the identifier S. In Standard ML this packaged environment is "timestamped" when the declaration is elaborated, marking it with a unique name that distinguishes it from all other environments, regardless of their internal structure. Such structure expressions are therefore said to be "generative" since each elaboration may be thought of as "generating" a new structure. The reason for making structure expressions generative in this sense is that the modules language provides a form of version control based on specifying that two possibly distinct structures or types must be equal. Since semantic equality of structures is undecidable, timestamps are used as a practical (and efficiently decidable) criterion for structure equality. We will ignore the issue of generativity in what follows, but will return to it in Section 9.3.

The components of a structure are accessed by qualified names, using a syntax reminiscent of record access in many languages. For instance, in the presence of the above binding for the structure identifier S, the identifier S.x refers to the x component of S, and hence evaluates to 3. Similarly, S.t refers to the t component of S and is equivalent to the type int during type checking. This *transparency* of type definitions distinguishes ML structures from abstract data type declarations (see MacQueen [29] and Mitchell and Plotkin [47] for related discussion).

Signatures are a form of "type" or "interface" for structures, and may be bound to signature identifiers using a signature binding, as follows:

```
signature SIG =
  sig
    type t
    val x: t
  end
```

This signature describes the class of structures having a type component, t, and a value component, x, whose type is the type bound to t in the structure. Since the structure S introduced above satisfies these conditions, it is said to *match* the signature SIG. The structure S also matches the following signature SIG′:

```
signature SIG′ =
  sig
    type t
    val x: int
  end
```

This signature is matched by any structure providing a type, t, and a value, x, of type int, which is indeed the case for the structure S. Note, however, that there are structures which match SIG, but not SIG′, namely any structure that provides a type other than int, and a value of that type.

In addition to ambiguities of this form, there is another, more practically-motivated, reason why a given structure may match a variety of distinct signatures. In ML signatures may be used to provide distinct *views* of a structure by a process of *ascription*. The main idea is that the signature may specify fewer components than are actually provided by the structure. The process of ascription introduces a suitable "thinning" coercion that eliminates the additional components of the structure. For example, we may introduce the signature

```
signature SIG″ =
  sig
    val x: Int
  end
```

and subsequently define a view, T, of the structure S, by writing

```
structure T: SIG″ = S
```

It should be clear from our discussion that S matches the signature SIG″ since it provides an x component of type int. The presence of the signature expression SIG″ in the binding for T causes the t component of S to be removed so that subsequently only the identifier T.x is available; the t component of S is not propagated to T, so that the identifier T.t is undefined. To simplify the development we do not detail the signature matching process, and instead regard structures as providing a unique signature describing each component. In this sense we regard signature matching as a convenience similar to that afforded by the type inference algorithm for the core language. For further discussion of signature matching, we refer the reader to Harper et al. [20], Tofte [63] and Milner et al. [40].

Discussion of ML "sharing" specifications is deferred to Section 9.3 below.

Functors (which are functions mapping structures to structures) are introduced using a syntax similar to that found in many programming languages:

```
functor F (S: SIG): SIG =
  struct
    type t = S.t * S.t
    val x: t = (S.x, S.x)
  end
```

This declaration introduces a functor F that takes as argument a structure matching the signature SIG, and yields as result a structure matching the same signature. (In Standard ML the parameter signature is mandatory, but, as a notational convenience, the result signature may be omitted, with the default obtained by an extension of the type inference algorithm for the core language.) When applied to a suitable structure S, the functor F yields as result the structure whose type component, t, is bound to the product of S.t with itself, and whose value component, x, is the pair both of whose components evaluate to the value of S.x.

By making use of free structure variables in signatures, certain forms of dependency of functor results on functor arguments may be expressed. For example, the following declaration specifies the type of y in the result signature of G in terms of the type component t of the argument S:

```
functor G (S: SIG): sig val y: S.t * S.t end =
  struct
    val y = (S.x, S x)
  end
```

This formulation of dependent types is consistent with the account given by MacQueen [29], and is accounted for similarly in our model of ML.

7. FULL XML

7.1 Syntax

In this section we will extend *Core*-XML to a function calculus XML by adding general constructs that allow us to describe the features of the previous section. Following MacQueen [29], we will use general sums and products in the style of Martin-Löf's type theory [33] to model the modules system. While general sums (also called "strong sums;" see Howard [23]) are closely related to structures, and general cartesian products seem necessary to capture dependently-typed functors, the language XML will be somewhat more general than ML. For example, while an ML structure may contain polymorphic functions, there is no direct way to define a polymorphic structure (i.e., a structure that is parametric in a type) in the implicitly-typed programming language. This is simply because there is no provision for explicit binding of type variables. However, polymorphic structures can be "simulated" in ML by using a functor whose parameter is a structure containing only a type binding. In XML, by virtue of the uniformity of the language definition, there will be no restriction on the types of things that can be made polymorphic. For similar reasons, XML will have expressions corresponding to higher-order functors and functor signatures, both of which would be useful additions to the language. In Section 9.3, we will discuss the addition of sharing constraints.

Intuitively, general sums and products correspond to infinitary disjoint union and Cartesian product constructions in set theory. If σ is a type, and σ' is a family of types indexed by σ, then the general product type, $\prod x\colon \sigma.\sigma'(x)$, is a set of functions f such that $f(x)$ is an element of $\sigma'(x)$ for every x in σ. Note that the range type depends on the domain element; for this reason general products are sometimes called "dependent" products. The general sum type, $\sum x\colon \sigma.\sigma'(x)$, consists of pairs p such that $fst(p)$ is an element of σ, and $snd(p)$ is an element of $\sigma'(fst(p))$ (where *fst* and *snd* are the first and second projections). Note that the type of the second component is expressed as a function of the first component: general sums are a form of "dependent type."

Unfortunately, general products and sums complicate the formalization of XML considerably. Since a structure may appear in a type expression, for example, it is no longer possible to describe the well-formed type expressions in isolation from the elements of those types. This also makes the well-formed contexts difficult to define. Therefore, we will define XML by giving a set of inference rules for determining the well-formed contexts, types and terms, in the style of Automath [14], Martin-Löf [32], and LF [18]. The unchecked preterms of XML are given by the following grammar:

$$\begin{aligned} M ::= {} & U_1 | U_2 | \mathit{triv} | M \rightarrow M | \Pi x : M.M | \Sigma x : M.M \\ & | x | * | \lambda x : M.M | MM | \lambda x : M.M | M[N] \\ & | \langle x : \sigma = M, M : \sigma' \rangle | \mathit{fst}(M) | \mathit{snd}(M) \end{aligned}$$

The metavariables M, N, and P range over the preterms. We also use σ and τ to range over preterms, particularly when the term is intended to be a type. Following Cardelli [7], we use an explicitly "dependent" form of ordered pair, $\langle x : \sigma = M, N : \sigma' \rangle$, in which the variable x is bound in N and σ'. We no longer include let as a primitive construct of the language since it is definable using abstraction over the polymorphic type $\sigma : U_2$ as $(\lambda x : \sigma . N)M$.

The type checking rules for XML appear in Tables III through VII. These rules refer to an equational theory of well-typed terms that appears in Tables VIII through XI.

As in *Core*-XML the universes of XML are cumulative in the sense that every U_1 type is a U_2 type as well. This simplifies the system somewhat, and, as we shall see below, is not significantly different from a system with an explicit inclusion of U_1 into U_2. Another feature of our type system is the treatment of ordered pairs of general sum type. The principal advantage of treating the pairing operator as a binding operator is that it makes it simpler to retain explicit typing, for the range type of the dependent sum is explicit in the notation. Without this information, the type of the pair cannot be recovered. The same phenomenon gives rise to the non-uniqueness of signatures in ML. The relation to the ordinary pairing operator is made clear by the equality axioms for pairs: the second projection replaces x in N by M, so that "externally" they behave like ordinary ordered pairs.

7.2 Equations and Reduction

The equational proof system for XML is given in Tables VIII through XI. If we direct the equational axioms from left to right, we obtain a reduction system of the form familiar from other systems of lambda calculus (e.g. Barendregt [2] and Mitchell [44]). Strong normalization is the property that there are no infinite reduction sequences from XML terms. In other words, the simple symbolic interpreter defined by the reduction rules is guaranteed to halt, on any term.

Strong normalization for XML may be proved using a translation into Martin-Löf's 1973 system [31]. It follows that the equational theory of XML is decidable. For other type systems, it is often possible to prove strong normalization by an appropriate method of *logical relations* [60, 44]. We consider it a significant open problem to develop a theory of logical relations for full XML, a task that is complicated by the presence of general Σ and Π types.

Table III. Context and Structural Rules for XML

$$\frac{}{\langle\rangle \text{ context}}$$

$$\frac{\Gamma \rhd \tau : U_1}{\Gamma, x{:}\tau \text{ context}} \qquad \frac{\Gamma \rhd \sigma : U_2}{\Gamma, x{:}\sigma \text{ context}}$$
$$(x \notin \text{Dom}(\Gamma))$$

$$\frac{\Gamma \text{ context} \quad \Gamma(x) = \tau}{\Gamma \rhd x : \tau} \qquad \frac{\Gamma \text{ context} \quad \Gamma(x) = \sigma}{\Gamma \rhd x : \sigma}$$

$$\frac{\Gamma \rhd \alpha \quad \Gamma \rhd \tau : U_1}{\Gamma, x{:}\tau \rhd \alpha} \qquad \frac{\Gamma \rhd \alpha \quad \Gamma \rhd \sigma : U_2}{\Gamma, x{:}\sigma \rhd \alpha}$$
$$(x \notin \text{Dom}(\Gamma),\ \alpha \textit{ is } M{:}\tau \textit{ or } M{:}\sigma)$$

$$\frac{\Gamma \rhd M : \tau \quad \Gamma \rhd \tau = \tau' : U_1}{\Gamma \rhd M : \tau'} \qquad \frac{\Gamma \rhd M : \sigma \quad \Gamma \rhd \sigma = \sigma' : U_2}{\Gamma \rhd M : \sigma'}$$

Table IV Universes

$$\frac{\Gamma \text{ context}}{\Gamma \rhd U_1 : U_2}$$

$$\frac{\Gamma \rhd \tau : U_1}{\Gamma \rhd \tau : U_2}$$

7.3 Representing Modules in XML

General sums allow us to write expressions for structures and signatures, provided we regard environments as tuples whose components are accessed by projection functions. For example, the structure

```
struct type t = int val x: t = 3 end
```

may be viewed as the pair $\langle t{:}\, U_1 = int, 3{:}\, t\rangle$. In XML, the components t and x are retrieved by projection functions, so that S.x is regarded as an abbreviation for *snd*(*S*). With general sums we can represent the signature

```
sig type t val x:t end
```

as the type $\Sigma t{:}\, U_1.t$, of which the pair $\langle t{:}\, U_1 = int, 3{:}\, int\rangle$ is a member. The representation of structures by unlabeled tuples is adequate in the sense that it is a simple syntactic translation to replace qualified names by expressions involving projection functions.

Since general products allow us to type functions from any collection to any other collection, we can write functors as elements of product types. For

Table V. Types and Terms in U_1

$$\frac{\Gamma \text{ context}}{\Gamma \rhd \mathit{triv} : U_1} \qquad \frac{\Gamma \text{ context}}{\Gamma \rhd * : \mathit{triv}}$$

$$\frac{\Gamma \rhd \tau : U_1 \quad \Gamma \rhd \tau' : U_1}{\Gamma \rhd \tau \to \tau' : U_1}$$

$$\frac{\Gamma \rhd \tau : U_1 \quad \Gamma \rhd \tau' : U_1 \quad \Gamma, x{:}\tau \rhd M : \tau'}{\Gamma \rhd \lambda x{:}\tau.M : \tau \to \tau'} \qquad (x \notin \mathrm{Dom}(\Gamma))$$

$$\frac{\Gamma \rhd M : \tau \to \tau' \quad \Gamma \rhd N : \tau}{\Gamma \rhd MN : \tau'}$$

Table VI. Types and Terms in U_2

$$\frac{\Gamma \rhd \sigma : U_2 \quad \Gamma, x{:}\sigma \rhd \sigma' : U_2}{\Gamma \rhd \Pi x{:}\sigma.\sigma' : U_2} \qquad (x \notin \mathrm{Dom}(\Gamma))$$

$$\frac{\Gamma \rhd \sigma : U_2 \quad \Gamma, x{:}\sigma \rhd \sigma' : U_2 \quad \Gamma, x{:}\sigma \rhd M : \sigma'}{\Gamma \rhd \lambda x{:}\sigma.M : \Pi x{:}\sigma.\sigma'} \qquad (x \notin \mathrm{Dom}(\Gamma))$$

$$\frac{\Gamma \rhd M : \Pi x{:}\sigma.\sigma' \quad \Gamma \rhd N : \sigma}{\Gamma \rhd M[N] : [N/x]\sigma'}$$

$$\frac{\Gamma \rhd \sigma : U_2 \quad \Gamma, x{:}\sigma \rhd \sigma' : U_2}{\Gamma \rhd \Sigma x{:}\sigma.\sigma' : U_2} \qquad (x \notin \mathrm{Dom}(\Gamma))$$

$$\frac{\Gamma \rhd M : \sigma \quad \Gamma \rhd [M/x]N : [M/x]\sigma' \quad \Gamma, x{:}\sigma \rhd \sigma' : U_2}{\Gamma \rhd \langle x{:}\sigma{=}M, N{:}\sigma' \rangle : \Sigma x{:}\sigma.\sigma'} \qquad (x \notin \mathrm{Dom}(\Gamma))$$

$$\frac{\Gamma \rhd M : \Sigma x{:}\sigma.\sigma'}{\Gamma \rhd \mathit{fst}(M) : \sigma}$$

$$\frac{\Gamma \rhd M : \Sigma x{:}\sigma.\sigma'}{\Gamma \rhd \mathit{snd}(M) : [\mathit{fst}(M)/x]\sigma'}$$

example, the functor bound to F by the declaration (where SIG is the signature above)

```
functor F(S: SIG): SIG =
  struct
    type t = S.t*S.t
    val x:t = (S.x, S.x)
  end
```

Table VII. Equalizer Types for Sharing Constraints

$$\frac{\Gamma \triangleright \sigma : U_2 \quad \Gamma, x{:}\sigma \triangleright \sigma' : U_2 \quad \Gamma, x{:}\sigma \triangleright M : \sigma' \quad \Gamma, x{:}\sigma \triangleright N : \sigma'}{\Gamma \triangleright \{\, x{:}\sigma \mid M = N : \sigma' \,\} : U_2} \quad (x \notin \mathrm{Dom}(\Gamma))$$

$$\frac{\Gamma \triangleright P : \sigma \quad \Gamma \triangleright [P/x]M = [P/x]N : [P/x]\sigma' : \quad \Gamma \triangleright \{\, x{:}\sigma \mid M = N : \sigma' \,\} : U_2}{\Gamma \triangleright P : \{\, x{:}\sigma \mid M = N : \sigma' \,\}}$$

$$\frac{\Gamma \triangleright P : \{\, x{:}\sigma \mid M = N : \sigma' \,\}}{\Gamma \triangleright P : \sigma}$$

$$\frac{\Gamma \triangleright P : \{\, x{:}\sigma \mid M = N : \sigma' \,\}}{\Gamma \triangleright [P/x]M = [P/x]N : [P/x]\sigma'}$$

is defined by the expression

$$\lambda S{:}(\Sigma t{:}U_1.t).\langle s{:}U_1 \\ = (fst(S) \times fst(S)), \langle snd(S), (snd(S)\rangle{:}(fst(S) \times fst(S))\rangle,$$

which has type

$$\Pi S{:}(\Sigma t{:}U_1.t).(\Sigma s{:}U_1.s).$$

The XML calculus is more general than Standard ML in two apparent respects. Since there is no need to require that subexpressions of XML types be closed, we are able to write explicitly-typed functors with nontrivial dependent types in XML. In addition, due to the uniformity of the language, we also have a form of higher-order functors.

Ignoring generativity, structure bindings in Standard ML are transparent in the sense that the components of a bound structure are fully visible within the scope of the binding. To capture this aspect of ML in type-theoretic terms, structure bindings are rendered using transparent let bindings, which are derived from dependent tuples. Specifically, a structure binding of the form

```
structure S =
  struct
    type t = int
    val x t = 7
  end;
```

is represented by the XML, term

$$snd\langle S{:}\Sigma t{:}U_1.t = \langle t{:}U_1 = int, 7{:}t\rangle, \ldots\rangle$$

where "..." is the translation of the remainder of the program, in keeping with the general idea that the top-level is a *let* expression of indefinite extent. Notice that the typing rules governing strong sums ensure that the definition of S is propagated to the remainder of the program, so that, in particular, $fst(S)$ is equivalent to int, as required. Functor bindings are handled similarly.

Table VIII. General Equality Rules for XML

$\Gamma \triangleright M = M : \tau$	$\Gamma \triangleright M = M : \sigma$
$\dfrac{\Gamma \triangleright M = N : \tau}{\Gamma \triangleright N = M : \tau}$	$\dfrac{\Gamma \triangleright M = N : \sigma}{\Gamma \triangleright N = M : \sigma}$
$\dfrac{\Gamma \triangleright M = N : \tau \quad \Gamma \triangleright N = P : \tau}{\Gamma \triangleright M = P : \tau}$	$\dfrac{\Gamma \triangleright M = N : \sigma \quad \Gamma \triangleright N = P : \sigma}{\Gamma \triangleright M = P : \sigma}$
$\dfrac{\Gamma \triangleright M = N : \tau \quad \Gamma, \Gamma' \text{ context}}{\Gamma, \Gamma' \triangleright M = N : \tau}$	$\dfrac{\Gamma \triangleright M = N : \sigma \quad \Gamma, \Gamma' \text{ context}}{\Gamma, \Gamma' \triangleright M = N : \sigma}$
$\dfrac{\Gamma \triangleright M = N : \tau \quad \Gamma \triangleright \tau = \tau' : U_1}{\Gamma \triangleright M = N : \tau'}$	$\dfrac{\Gamma \triangleright M = N : \sigma \quad \Gamma \triangleright \sigma = \sigma' : U_2}{\Gamma \triangleright M = N : \sigma'}$

Table IX. Equality Rules for the Function Type

$$\Gamma \triangleright \lambda x{:}\tau.M = \lambda y{:}\tau.[y/x]M : \tau \rightarrow \tau' \qquad (y \notin \mathrm{FV}(M))$$

$$\Gamma \triangleright (\lambda x{:}\tau.M)N = [N/x]M : \tau'$$

$$\Gamma \triangleright \lambda x{:}\tau.Mx = M : \tau \rightarrow \tau' \qquad (x \notin \mathrm{FV}(M))$$

$$\frac{\Gamma \triangleright \tau_1 = \tau_1' : U_2 \quad \Gamma \triangleright \tau_2 = \tau_2' : U_2}{\Gamma \triangleright \tau_1 \rightarrow \tau_2 = \tau_1' \rightarrow \tau_2' : U_2}$$

$$\frac{\Gamma \triangleright M = M' : \tau \rightarrow \tau' \quad \Gamma \triangleright N = N' : \tau}{\Gamma \triangleright MN = M'N' : \tau'}$$

$$\frac{\Gamma, x{:}\tau \triangleright M = M' : \tau'}{\Gamma \triangleright \lambda x{:}\tau.M = \lambda x{:}\tau.M' : \tau \rightarrow \tau'}$$

8. PREDICATIVITY AND THE RELATIONSHIP BETWEEN UNIVERSES

8.1 Universes

Each of the constructs of XML is designed to capture a specific part of the programming language. In an effort to provide a vocabulary for discussing extensions to ML, and to simplify the presentation of the type theory, we have allowed arbitrary combinations of constructs and straightforward extensions like higher-order functor expressions. While generalizing in certain ways that seem syntactically and semantically natural, we have retained the

Table X. Equality Rules for the Product Type

$$\Gamma \rhd \lambda x{:}\sigma.M = \lambda y{:}\sigma.[y/x]M : \Pi x{:}\sigma.\sigma' \qquad (y \notin \mathrm{FV}(M))$$

$$\Gamma \rhd (\lambda x{:}\sigma.M)N = [N/x]M : [N/x]\sigma'$$

$$\Gamma \rhd \lambda x{:}\sigma.Mx = M : \Pi x{:}\sigma.\sigma' \qquad (x \notin \mathrm{FV}(M))$$

$$\frac{\Gamma \rhd \sigma_1 = \sigma_1' : U_2 \quad \Gamma, x{:}\sigma_1 \rhd \sigma_2 = \sigma_2' : U_2}{\Gamma \rhd \Pi x{:}\sigma_1.\sigma_2 = \Pi x{:}\sigma_1'.\sigma_2' : U_2}$$

$$\frac{\Gamma, x{:}\sigma \rhd M = M' : \sigma'}{\Gamma \rhd \lambda x{:}\sigma.M = \lambda x{:}\sigma.M' : \Pi x{:}\sigma.\sigma'}$$

$$\frac{\Gamma \rhd M = M' : \Pi x{:}\sigma.\sigma' \quad \Gamma \rhd N = N' : \sigma}{\Gamma \rhd M[N] = M'[N'] : [N/x]\sigma'}$$

Table XI. Equality Rules for the Sum Type

$$\Gamma \rhd \mathit{fst}\langle x{:}\sigma{=}M, N{:}\sigma'\rangle = M : \sigma$$

$$\Gamma \rhd \mathit{snd}\langle x{:}\sigma{=}M, N{:}\sigma'\rangle = [M/x]N : [M/x]\sigma'$$

$$\frac{\Gamma \rhd \sigma_1 = \sigma_1' : U_2 \quad \Gamma, x{:}\sigma_1 \rhd \sigma_2 = \sigma_2' : U_2}{\Gamma \rhd \Sigma x{:}\sigma_1.\sigma_2 = \Sigma x{:}\sigma_1'.\sigma_2' : U_2} \qquad (x \notin \mathrm{Dom}(\Gamma))$$

$$\frac{\Gamma \rhd M = M' : \sigma \quad \Gamma \rhd [M/x]N = [M'/x]N' : [M/x]\sigma'}{\Gamma \rhd \langle x{:}\sigma{=}M, N{:}\sigma'\rangle = \langle x{:}\sigma{=}M', N'{:}\sigma'\rangle : \Sigma x{:}\sigma.\sigma'} \qquad (x \notin \mathrm{Dom}(\Gamma))$$

$$\frac{\Gamma \rhd M = M' : \Sigma x{:}\sigma.\sigma'}{\Gamma \rhd \mathit{fst}(M) = \mathit{fst}(M') : \sigma}$$

$$\frac{\Gamma \rhd M = M' : \Sigma x{:}\sigma.\sigma'}{\Gamma \rhd \mathit{snd}(M) = \mathit{snd}(M') : [\mathit{fst}(M)/x]\sigma'}$$

distinction between monomorphic and polymorphic types by keeping U_1 and U_2 distinct. The restrictions imposed by universes are essential to the proof of Theorem 4.1 and have the technical advantage of leading to far simpler semantic model constructions. However, it may seem reasonable to generalize ML polymorphism by lifting the universe restrictions (as in the Girard-Reynolds second-order lambda calculus), or alter the design decisions $U_1 \subseteq U_2$ and $U_1 : U_2$.

In this section we will show that the decision to take $U_1 \subseteq U_2$ and $U_1 : U_2$ is essentially forced by the other constructs of XML, and that in the presence of structures and functors, the universe restrictions are essential if we are to avoid introducing a type of all types. We refer the reader to Coquand [11], Howe [24], Meyer and Reinhold [36] and Cardelli [6] for background information and further discussion of the merits of *type*: *type*. As discussed in the introduction, it seems fair to say that *type*: *type* would certainly change the character of ML dramatically. However, further research is needed to understand the ramifications of *type*:*type* more precisely.

8.2 U_1 as a Subset of U_2

In XML, we have $U_1 \subseteq U_2$, since every U_1 type is also treated as a U_2 type. The main reason for this is that it simplifies both the use of the language, and a number of technical details in its presentation. For example, by putting every $\tau : U_1$ into U_2 as well, we can write

$$\frac{\sigma : U_2}{\prod t : U_1 . \sigma : U_2}$$

for the $\prod$-formation rule, instead of giving two separate cases for $\tau : U_1$ and $\sigma : U_2$. An important part of this design decision is that $U_1 \subseteq U_2$ places no additional semantic constraints on XML. More specifically, if we remove the relevant typing rule from the language definition, we are left with a system in which every U_1 type is represented as a retract of some U_2 type. This allows us to faithfully translate XML into the language without $U_1 \subseteq U_2$, so that every semantic model of XML without $U_1 \subseteq U_2$ may serve as a semantic model for XML with $U_1 \subseteq U_2$. The justification for assuming $U_1 \subseteq U_2$ is made more precise by the following lemma.

LEMMA 8.1. *Let $\tau : U_1$ be any type from the first universe, and let t be a variable that is not free in τ. Then there are XML contexts*

$$i[\ ] \equiv \lambda t : U_1 . [\ ] \quad \textit{and} \quad j[\ ] \equiv [\ ]\mathit{triv},$$

where triv may be any type with the following properties:

—$\Gamma \triangleright i[M] : \prod t : U_1 . \tau$ *whenever* $\Gamma \triangleright M : \tau$

—$\Gamma \triangleright j[m] : \tau$ *whenever* $\Gamma \triangleright M : \prod t : U_1 . \tau$

—$\Gamma \triangleright j[i[M]] = M : \tau$ *for all* $\Gamma \triangleright M : \tau$.

In other words, given the hypothesis above, we may assume $\tau : U_2$ without loss of generality.

Using the contexts $i[\ ]$ and $j[\ ]$, it is quite easy to translate every term in XML with $U_1 \subseteq U_2$ into an equivalent expression that is typed without using $U_1 \subseteq U_2$. Essentially, the translation replaces every use of $\tau : U_1$ as a U_2 type with $(\prod t : U_1 . \tau) : U_2$, and encloses terms in contexts $i[\ ]$ and $j[\ ]$ to make the typing work out right. Since this translation preserves equality and the structure of terms, there is no loss of generality in having $U_1 \subseteq U_2$.

8.3 Strong Sums and $U_1 : U_2$

In the explicitly-typed core language *Core*-XML, we have $U_1 \subseteq U_2$ but not $U_1 : U_2$. However, when we added general product and sum types, we also made the assumption that $U_1 : U_2$. The reasons for this are similar to the reasons for taking $U_1 \subseteq U_2$: it makes the syntax more flexible, simplifies the technical presenttion, and does not involve any unnecessary semantic assumptions. A precise statement is spelled out in the following lemma.

LEMMA 8.2 *In any fragment of XML which is closed under the term formation rules for types of the form $\Sigma t : U_1 . \tau$, with $\tau : U_1$, there are contexts*

$$i[\;] \equiv \langle [\;], * \rangle \quad and \quad j[\;] = fst[\;],$$

where triv may be any U_1 type with closed term $* : triv$, satisfying the following conditions:

(1) *If* $\Gamma \rhd \tau : U_1$, *then* $\Gamma \rhd i[\tau] : (\Sigma t : U_1 . triv)$.
(2) *If* $\Gamma \rhd M : (\Sigma t : U_1 . triv)$, *then* $\Gamma \rhd j[M] : U_1$.
(3) $\Gamma \rhd [i[\tau]] = \tau : U_1$ *for all* $\Gamma \rhd \tau : U_1$.

In other words, given the hypotheses above, since $(\Sigma t : U_1 . triv) : U_2$, we may assume $U_1 : U_2$ without loss of generality.

In words, the lemma says that in any fragment of XML, with sums over U_1 (and some U_1 type *triv* containing a closed term $*$), we can represent U_1 by the type $\Sigma t : U_1 . triv$. Therefore, even if we drop $U_1 : U_2$ from the language definition, we are left with a representation of U_1 inside U_2. For this reason, we might as well simplify matters and take $U_1 : U_2$.

8.4 Impredicativity and "*type*:*type*"

In XML, as in ML, polymorphic functions are not actually applicable to arguments of all types. For example, the identity function defined by $id(x) = x$ has polymorphic type, but it can only be applied to elements of types from the first universe. We cannot apply the same identity function *id* to both integers and structures. One way to eliminate this restriction is to eliminate the distinction between U_1 and U_2. If we replace U_1 and U_2 by a single universe in the definition of *Core*-XML, then we obtain the second-order lambda calculus of Girard and Reynolds [16, 15, 53]. (A similar technique is used to introduce impredicativity into Nuprl in Howe [24].) The Girard-Reynolds calculus has a number of reasonable theoretical properties (e.g. see Bruce et al. [5], Mitchell [41] and Mitchell and Plotkin [47]) and seems to be a useful tool for studying polymorphism in programming.

However, if we make the *full* XML calculus impredicative by eliminating the distinction between U_1 and U_2, the language becomes very different from the Girard-Reynolds calculus. Specifically, since we have general products and $U_1 : U_2$, it is quite easy to see that if we let $U_1 = U_2$ then Meyer and Reinhold's language $\lambda^{\tau : \tau}$ with a type of all types [36] becomes a sublanguage of XML. Note that although the term formation rules of XML only provide general products over U_2 types, letting $U_1 = U_2$ will give us products over all types.

LEMMA 8.3 *Any fragment of XML with $U_1 : U_2$, $U_1 = U_2$, and closed under the type and term formation rules associated with general products is capable of expressing all terms of $\lambda^{\tau:\tau}$ of Meyer and Reinhold* [36].

PROOF. The proof is a straightforward induction on the typing rules of $\lambda^{\tau:\tau}$. Since we assume that $U_1 = U_2$, we may unambiguously write U for the collection of types. This makes it easy to see that the typing rules and equational rules of $\lambda^{\tau:\tau}$ are derived rules of XML with $U_1 : U_2$, $U_1 = U_2$ and general products. (This is not surprising, since the language $\lambda^{\tau:\tau}$ is designed to be a "minimal" calculus with a type of all types and general products.) In particular, if U is the collection of all types, then we clearly have $U : U$, by hypothesis. □

By Lemma 8.2, we know that sums over U_1 give us $U_1 : U_2$. This proves the following theorem.

THEOREM 8.4 *The function calculus $\lambda^{\tau:\tau}$ with a type of all types may be interpreted in any fragment of XML without universe distinctions which is closed under general products, and sums over U_1 of the form $\Sigma t : U_1 . \tau$.*

Intuitively, this says that any language without universe distinctions that has general products (ML functors) and general sums restricted to U_1 (ML structures with type and value but not necessarily structure components) also contains the language $\lambda^{\tau:\tau}$ with a type of all types. Since there are a number of questionable properties of $\lambda^{\tau:\tau}$ such as nontermination without explicit recursion and undecidable type checking, relaxing the universe restrictions of XML would alter the language dramatically.

8.5 Tradeoff Between Weak and Strong Sums

When we first discovered Theorem 8.4, we announced it as a tradeoff theorem in programming language design.[2] The "tradeoff" implied by Theorem 8.4 is between impredicative polymorphism and the kind of Σ types used to represent ML structures in XML. Generally speaking, impredicative polymorphism is more flexible than predicative polymorphism, and Σ types allow us to type more terms than the existential types associated with data abstraction (see Mitchell and Plotkin [47]).

Either impredicative polymorphism with the "weaker" existential types, or restricted predicative polymorphism with "stronger" sum types seems reasonable. By the normalization theorem for the impredicative Girard-Reynolds calculus [15, 41],[3] we know that impredicative polymorphism with existential types is strongly normalizing. As noted in Section 7, a translation into

[2] We described our "tradeoff theorem" in the TYPES electronic mail forum in the Spring of 1986. Hook and Howe then replied that they had discovered a similar phenomenon independently [22]. We also learned that Coquand had proved the same theorem by a different means (see Coquand [11]), which was in preparation at the time of our announcement.

[3] Girard's original proof included existential types. While the somewhat simpler proof in Mitchell [41] does not, normalization with existential types can easily be derived by encoding $\exists t.\sigma$ as $\forall r[\forall t(\sigma \to r) \to r]$.

Martin-Löf's 1973 system [31] shows that XML with predicative polymorphism and "strong" sums is also strongly normalizing. However, by Theorem 8.4, we know that if we combine strong sums with impredicative polymorphism by taking $U_1 = U_2$, the most natural way of achieving this end, then we must admit a type of all types. By Girard's paradox [11, 36, 24], *type*: *type* (in the presence of other constructs) implies that strong normalization fails. In short, assuming we wish to avoid *type*: *type* and nonnormalizing recursion-free terms, we have a tradeoff between impredicative polymorphism and strong sums.

In formulating the XML type theory, it became apparent that there were actually several ways to combine impredicative polymorphism with strong sums. The most reasonable is this: instead of adding impredicative polymorphism by equating the two universes, we may add a form of impredicative polymorphism by adding a new type binding operator with the formation rule

$$\frac{\Gamma, t : U_1 \rhd \tau : U_1}{\Gamma \rhd \forall t : U_1 . \tau : U_1}.$$

Intuitively, this rule says that if τ is a U_1 type, then we will also have the polymorphic type $\forall t : U_1 . \tau$ in U_1. The term formation-rules for this sort of polymorphic type would allow us to apply any polymorphic function of type $\forall t : U_1 . \tau$ to any type in U_1, including a polymorphic type of the form $\forall s : U_1 . \sigma$. However, we would still have strong sums like $\Sigma t : U_1 . \tau$ in U_2 instead of U_1. The normalization theorem for this calculus follows from that of the theory of constructions with strong sums at the level of types [11] by considering U_1 to be *prop*, and U_2 to be $type_0$.

9. EXTENSIONS

9.1 Introduction

ML contains a variety of language features beyond those we have considered so far. For the benefit of the reader familiar with ML, we briefly sketch an approach to type declarations and sharing constraints in XML.

9.2 Type Declarations

There are three mechanisms for introducing types and type constructors in ML: type abbreviations, concrete data type declarations, and abstract data type declarations. A type abbreviation is a form of "compile-time" let-binding which allows for the definition of a type constructor in terms of types and type constructors that have been previously introduced. A concrete type declaration simultaneously introduces a recursively-defined type constructor, a finite collection of *value constructors*, and a *pattern matching* construct. An abstract type declaration introduces a "private" concrete data type, together with a set of "public" operations on that type. We give a brief description of each form below. For more information, see Harper et al. [19] and Milner et al. [40].

In the remainder of this section, we show how the three forms of type declarations just mentioned may be treated in XML. In each case, we do this by first extending *Core*-XML with a form of declaration that resembles the surface syntax of ML and then showing how this may be desugared into simpler XML constructs. While type abbreviations may be interpreted directly in pure XML, concrete and abstract type declarations require the extension of XML with disjoint unions, type recursion and existential types.

We begin with type abbreviations. We extend the grammar of *Core*-XML with a transparent type binding construct of the form

$$\text{type } (t_1, \ldots, t_n)t = \tau \text{ in } e.$$

The scope of the type constructor t is the expression e; the scope of the type variables $t_1, \ldots, t_n$ is the type expression τ. The effect of such a transparent type binding is to introduce an n-argument type construct t with the property that $(\tau_1, \ldots, \tau_n)t$ is equivalent to $[\tau_1, \ldots, \tau_n / t_1, \ldots, t_n]\tau$ during type checking of e. For example, the expression

$$\text{type } (s_1, s_2)t = s_1 \to s_2 \text{ in } e$$

has the effect of introducing a two-place type constructor t within e so that during type checking of e the types $(int, int)t$ and $int \to int$ are equivalent.

We may represent type declarations in pure XML using a combination of product and function types at the U_2 level. Specifically, we regard the expression

$$\text{type } (t_1, \ldots, t_n)t = \tau \text{ in } e$$

as short-hand for the XML expression

$$snd(\langle t : U_1^n \to U_1 = \lambda\langle t_1 : U_1, \ldots, t_n : U_1\rangle.\tau, e\rangle)$$

where U_1^n stands for the n-fold Cartesian product $U_1 \times \cdots \times U_1$, and where the pattern-directed, λ-abstraction abbreviates the less perspicuous

$$\lambda t : U_1^n.[\pi_1^n(t), \ldots, \pi_n^n(t)/t_1, \ldots, t_n]e.$$

Here, and below, π_i^n stands for the appropriate combination of first and second projections to select the ith component from a value of n-fold product type.

The reason we use pairing and projections associated with Σ types for type abbreviations, instead of the apparently simpler alternative,

$$(\lambda t : U_1^n \to U_1.e)(\lambda\langle t_1 : U_1, \ldots, t_n : U_1\rangle.\tau),$$

is that in the latter term, the expression $\lambda t : U_1^n \to U_1.e$ would have to be well-typed. This requires e to be well-typed for *any* function $t : U_1^n \to U_1$. In contrast, a pair $\langle t : U_1^n \to U_1 = M, N\rangle$ is typed by showing that the term $[M/t]N$ obtained by substitution is well-typed. This is easily seen in the appropriate Σ typing rule in Table VI.

Concrete type definitions are somewhat more complex since they simultaneously introduce a recursively-defined type constructor, a finite set of value constructors for building values of that type, and a pattern matching construct for destructuring values of that type. To account for concrete data types in *Core*-XML, we extend the grammar of expressions as follows:

$$\begin{aligned} e ::= &\ \mathsf{datatype}(t_1, \dots, t_n)t = c_1 \ \mathsf{of}\ \tau_1 | \cdots | c_m \ \mathsf{of}\ \tau_m \ \mathsf{in}\ e \\ &| \mathsf{case}\ e\ \mathsf{of}\ c_1(x_1 : \tau_1) \Rightarrow e_1 | \cdots | c_m(x_m : \tau_m) \Rightarrow e_m \end{aligned}$$

Note that the vertical bars, "|", in datatype and case expressions are part of the object syntax of *Core*-XML, not metanotation. Both the datatype and case forms are binding operators. The scope of the type constructor t in a datatype expression of the above form includes both the body of the expression, e, and the type expressions $\tau_1, \dots, \tau_m$, reflecting the fact that t may be defined recursively. The scope of the value constructors $c_1, \dots, c_m$ associated with t is the body e of the declaration. The scope of the type parameters $t_1, \dots, t_n$ of t is limited to the type expressions $\tau_1, \dots, \tau_m$. In a case expression of the above form, the scope of each variable x_i is limited to the corresponding expression $e_i (1 \leq i \leq m)$.

The effect of a datatype expression is to introduce within the body of the expression an n-place type constructor and m value constructors. The type constructor is recursively defined in terms of the given value constructors in a manner outlined below. The case construct supports simultaneous case analysis and decomposition of values of the type introduced by a datatype expression in a manner similar to that of Standard ML. The full Standard ML language provides a somewhat richer form of pattern-matching that admits both layered and nested patterns, but we do not consider this generalization here.

ML concrete data type declarations may be accounted for in an extension of XML with disjoint union types at the U_1 level, existential types [47] at the U_2 level, and the ability to form recursively-defined type constructors at the U_2 level. We briefly summarize these extensions before discussing the interpretation of concrete data type declarations in XML.

Disjoint unions, which we write using the symbol $+$, are likely to be familiar from a variety of programming languages. If $\tau_1, \tau_2 : U_1$, then the disjoint union type $\tau_1 + \tau_2$ is also a U_1 type expression. Expressions of union type are formed using injection functions, *inl* and *inr* according the rules that if $\Gamma \rhd M : \tau_1$, then $\Gamma \rhd inl M : \tau_1 + \tau_2$ and if $\Gamma \rhd M : \tau_2$ then $\Gamma \rhd in \tau M : \tau_1 + \tau_2$. The *case* statement is used to test which summand a value belongs to, according to the following rule.

$$\frac{\Gamma \rhd M : \tau_1 + \tau_2 \quad \Gamma, x_1 : \tau_1 \rhd N : \rho \quad \Gamma, x_2 : \tau_2 \rhd P : \rho}{\Gamma \rhd case\ M\ of\ inl(x_1 : \tau_1) \Rightarrow N | inr(x_2 : \tau_2) \Rightarrow P : \rho}.$$

Existential types, which may be regarded as a "weak" form of Σ type, are formed according to the rule below. Although we could existentially quantify over any collection from U_2, we will only need existential quantification over collections of the form $U_1^n \to U_1$. For simplicity, we only present the formation and typing rules for the form of existential types we need.

$$\frac{\Gamma, t: U_1^n \to U_1 \rhd \sigma : U_2}{\Gamma \rhd \exists t: U_1^n \to U_1.\sigma : U_2} \qquad (t \notin \mathrm{Dom}(\Gamma)).$$

There are two differences between existential types in XML and the language considered by Mitchell and Plotkin [47]. The form given here is more general in that we quantify over n-ary type constructors, rather than just types. It is more restrictive in that we only provide *predicative* quantification in the sense that the existential type $\exists t: U_1^n \to U_1.\sigma$ belongs to the second, rather than the first, universe. Expressions of existential type are formed and used according to the following two rules.

$$\frac{\Gamma \rhd \tau : U_1^n \to U_1 \quad \Gamma \rhd M:[\tau/t]\sigma \quad \Gamma, t: U_1^n \to U_1 \rhd \sigma : U_2}{\Gamma \rhd \langle t: U_1^n \to U_1 = \tau, M: \sigma\rangle : \exists t: U_1^n \to U_1.\sigma} \qquad (t \notin \mathrm{Dom}(\Gamma))$$

$$\frac{\Gamma \rhd M: \exists t: U_1^n \to U_1.\sigma \quad \Gamma, t: U_1^n \to U_1, x: \sigma \rhd N: \rho}{\Gamma \rhd \mathit{abstype}\, t: U_1^n \to U_1 \mathit{\ with\ } x: \sigma \mathit{\ is\ } M \mathit{\ in\ } N: \rho} \qquad (t \notin FV(\rho))$$

For further discussion of existential types, the reader is referred to Mitchell and Plotkin [47], Cardelli and Wegner [8], and MacQueen [28].

To account for recursively-defined type constructors, we introduce a fixed-point operator

$$\mathit{fix}_n: ((U_1^n \to U_1) \to (U_1^n \to U_1)) \to (U_1^n \to U_1)$$

for each $n \geq 0$. (For the special case $n = 0$, the fixed-point operator has type $(U_1 \to U_1) \to U_1$.) Intuitively, for any type functional $\theta: (U_1^n \to U_1) \to (U_1^n \to U_1)$, the type function $\mathit{fix}_n \theta$, when applied to an n-tuple of types $\langle \tau_1, \ldots, \tau_n \rangle$, yields a type isomorphic to $\theta(\mathit{fix}_n \theta)\langle \tau_1 \ldots \tau_n \rangle$. Formally, the extension of XML with recursive type functions over U_1 is completed by adding the constants

$$In_n: \Pi\theta: (U_1^n \to U_1) \to (U_1^n \to U_1).\Pi t: U_1^n.(\theta(\mathit{fix}_n \theta)t) \to (\mathit{fix}_n \theta)t$$
$$Out_n: \Pi\theta: (U_1^n \to U_1) \to (U_1^n \to U_1).\Pi t: U_1^n.(\mathit{fix}_n \theta)t \to \theta(\mathit{fix}_n \theta)t$$

for each $n \geq 0$, together with equational axioms making them mutually inverse to one another. Rather than define the solution of type constructor equations only up to isomorphism, it would also be possible to take $\mathit{fix}_n \theta\langle \tau_1, \ldots, \tau_n \rangle$ to be equal to $\theta(\mathit{fix}_n \theta)\langle \tau_1, \ldots, \tau_n \rangle$, but this would allow more XML terms to be typable than would be accepted by the usual ML type checking algorithm.

With this additional machinery in hand we may represent concrete data types in XML as follows. The concrete data type declaration

$$\mathsf{datatype}\ (t_1, \ldots, t_n)t = c_1: \tau_1 | \cdots | c_m: \tau_m \ \mathsf{in}\ e$$

is rendered in XML as the elimination form

$$\mathit{abstype}\, t: U_1^n \to U_1 \mathit{\ with\ } c: \sigma \mathit{\ is\ } \langle t: U_1^n \to U_1 = \mathit{fix}_n(\theta), d: \sigma \rangle \mathit{\ in\ } e'$$

associated with the existential type $\exists t : U_1^n \to U_1 . \sigma$, where the expressions θ, σ, d, and e' are given below. The main idea behind this representation is that a concrete data type declaration introduces a "new" recursively-defined type constructor, together with operations corresponding to each of the value constructors, and an operation corresponding to the case analysis form associated with that type. The typing rules governing abstype ensure that the type constructor t is distinct from both the given definition and from all other type constructors in that scope. In view of the fact that abstype binds the variable t in e' it is always possible to arrange (by an application of α-conversion) that t is distinct from all other type constructors. Thus the ML notion of "type generativity" is reduced to the more familiar idea of renaming of bound variables.

The expression θ of type $(U_1^n \to U_1) \to (U_1^n \to U_1)$ is defined to be

$$\lambda t : U_1^n \to U_1 . \lambda \langle t_1 : U_1, \ldots, t_n : U_1 \rangle . (\tau_1 + \cdots + \tau_m).$$

Note that by the definition of θ and the rules governing recursively-defined type constructors, the type $\mathit{fix}_n \theta \langle t_1, \ldots, t_n \rangle$ is isomorphic (via *In* and *Out*) to the type $[\mathit{fix}_n \theta / t]\tau_1 + \cdots + [\mathit{fix}_n \theta / t]\tau_m$. The type σ is defined to be the product $\sigma_1 \times \cdots \times \sigma_m \times \sigma_{m+1}$, where for each $1 \le i \le m$, the type σ_i is

$$\Pi t_1 : U_1 \ldots \Pi t_n : U_1 . \tau_1 \to t\langle t_1, \ldots, t_n \rangle,$$

and the type σ_{m+1} is

$$\Pi t_1 : U_1 \ldots \Pi t_m : U_1 . \Pi u : U_1 . t\langle t_1, \ldots, t_n \rangle \to (\theta t \langle t_1, \ldots, t_n \rangle \to u) \to u.$$

Intuitively, σ_i is the type of the ith constructor, for $1 \le i \le m$, and σ_{m+1} is the type of the *case* construct for the data type. The expression d of type $[\mathit{fix}_n \theta / t]\sigma$ is the tuple $\langle d_1, \ldots, d_m, d_{m+1} \rangle$, where for each $1 \le i \le m$, the expression d_i is

$$\lambda t_1 : U_1 \ldots \lambda t_n : U_1 . \lambda x_i : [\mathit{fix}_n \theta / t]\tau_i . \mathit{In}_n \theta \langle t_1, \ldots, t_n \rangle (\mathit{inj}_i^m x)$$

and the expression d_{m+1} is

$$\lambda t_1 : U_1 \ldots \lambda t_n : U_1 . \lambda u : U_1 . \lambda x : \mathit{fix}_n \theta \langle t_1, \ldots, t_n \rangle . \lambda f : \theta(\mathit{fix}_n \theta)\langle t_1, \ldots, t_n \rangle \\ \to u . f(\mathit{Out}_n \theta \langle t_1, \ldots, t_n \rangle x).$$

Here inj_i^m stands for the appropriate combination of *inl*'s and *inr*'s to inject a value of the ith summand into an n-ary disjoint union type. Intuitively, d_i is the implementation of the ith constructor, for $1 \le i \le m$, and d_{m+1} provides the *case* construct for the data type. It is not hard to see for each $1 \le i \le m+1$, the expression d_i is of type $[\mathit{fix}_n \theta / t]\sigma_i$ so that the tuple d has the required type $[\mathit{fix}_n \theta / t]\sigma$.

The expression e' is obtained from e by replacing occurrences of a value constructor c_i with the corresponding projection $\pi_i^{m+1} c$, and by interpreting the case analysis form

$$\mathsf{case}\ r\ \mathsf{of}\ c_1(x_1 : \tau_1) \Rightarrow e_1 | \cdots | c_m(x_m : \tau_m) \Rightarrow e_m$$

as the application $\pi^{m+1}_{m+1} c \rho_1 \ldots \rho_n \rho r f$ where r has type $t\langle \rho_1, \ldots, \rho_n \rangle$, the entire expression has type ρ, and the function f is given by

$$\begin{aligned}
&\lambda x\colon \theta(\mathit{fix}_n \theta)\langle \rho_1, \ldots, \rho_n \rangle.\\
&\quad \mathit{case}\ x\ \mathit{of}\ \mathit{inl}(x_1\colon \tau'_1) \Rightarrow e_1 | \mathit{inr}(y_1\colon \tau''_1) \Rightarrow \ldots\\
&\qquad \mathit{case}\ y_{m-2}\colon \tau''_{m-2}\ \mathit{of}\ \mathit{inl}(x_{m-1}\colon \tau'_{m-1}) \Rightarrow e_{m-1} | \mathit{inr}(x_m\colon \tau'_m) \Rightarrow e_m,
\end{aligned}$$

where for each $1 \leq i \leq m$, the type τ'_i is $[\mathit{fix}_n \theta t/t]\tau_i$, and the type τ''_i is the "partial sum" type $\tau'_{i+1} + \cdots + \tau'_m$.

For example, the *Core*-XML expression

$$\text{datatype}\ t\ \mathit{list} = \mathit{nil} | \mathit{cons}\ \text{of}\ t \times t\ \mathit{list}\ \text{in}\ e$$

is represented by the XML expression

$$\mathit{abstype}\ \mathit{list}\colon U_1 \to U_1\ \mathit{with}\ \mathit{ncc}\colon\sigma\ \mathit{is}\ \langle \mathit{list}\colon U_1 \to U_1 = \mathit{fix}_1 \theta, d\colon \sigma \rangle\ \mathit{in}\ e'$$

where

$$\begin{aligned}
\theta &= \lambda L\colon U_1 \to U_1.\lambda t\colon U_1.\mathit{triv} + (t \times L(t))\\
\sigma &= \sigma_{nil} \times \sigma_{cons} \times_{case}\\
\sigma_{nil} &= \textstyle\prod t\colon U_1.\mathit{triv} \to \mathit{list}(t)\\
\sigma_{cons} &= \textstyle\prod t\colon U_1.(t \times \mathit{list}(t)) \to \mathit{list}(t)\\
\sigma_{case} &= \textstyle\prod t\colon U_1.\prod u\colon U_1.\mathit{list}(t) \to (\mathit{triv} + (t \times \mathit{list}(t)) \to u) \to u\\
e &= \langle e_{nil}, e_{cons}, e_{case} \rangle\\
e_{nil} &= \lambda t\colon U_1.\lambda x\colon \mathit{triv}.\mathit{In}_1 \theta(\mathit{inl}(x))\\
e_{cons} &= \lambda t\colon U_1.\lambda x\colon t \times (\mathit{fix}_1 \theta t).\mathit{In}_1 \theta(\mathit{inr}(x))\\
e_{case} &= \lambda t\colon U_1.\lambda u\colon U_1.\lambda x\colon \mathit{fix}_1 \theta t.\lambda f\colon (1 + (t \times (\mathit{fix}_1 \theta t))) \to u.f(\mathit{Out}_1 \theta t x)
\end{aligned}$$

The expression e' is obtained from e as described above, replacing occurrences of *nil* and *cons* by suitable projections of *ncc*, and replacing case analyses on terms of type ρ *list* by suitable applications of the case analysis function, $\pi_1(\pi_2 \mathit{ncc})$.

Abstract type declarations are accounted for in *Core*-XML by adding expressions of the form:

$$\text{abstype}(t_1, \ldots, t_n)t = c_1\ \text{of}\ \tau_1 | \cdots | c_m\ \text{of}\ \tau_m\ \text{with}\ x_1\colon \rho_1 = e_1, \ldots, x_k\colon \rho_k = e_k\ \text{in}\ e.$$

Informally, the effect of an abstype declaration is to introduce a "private" concrete data type for use in the definition of the "public" operations in the with clause, but hiding this declaration from the "client" expression e, which has access only to these public operations. More precisely, the scope of the constructors $c_1, \ldots, c_m$ is limited to the definitions $e_1, \ldots, e_k$ of $x_1, \ldots, x_k$, even though the scope of the type constructor t includes not only the e_i's but also the body e. On the other hand, the scope of the variables $x_1, \ldots, x_k$ naming the public operations is limited to e. (We omit, for simplicity, the possibility of mutually recursive definitions of the public operations.)

The representation of abstype expressions in XML is similar to the representation of datatype expressions, except that the recursive type is kept abstract by making the value constructors and case analysis forms available only in the definitions of the operations of the abstract type. Thus an abstype expression of the above form is represented by the expression

$$abstype\, t: U_1^n \to U_1 \;with\; x: \rho \;is\; \langle fix_n(\theta): U_1^n \to U_1, p: \rho \rangle \;in\; e$$

where p is the expression

$$let\; c_1: \sigma_1 = d_1, \ldots, c_m: \sigma_m = d_m, c_{m+1}: \sigma_{m+1} = d_{m+1} \;in\; \langle e_1, \ldots, e_k \rangle$$

of type $\rho = \rho_1 \times \cdots \rho_k$, and where the expressions θ, $\sigma_1, \ldots, \sigma_{m+1}$, and $d_1, \ldots, d_{m+1}$ are as above.

9.3 Generativity and Sharing

A distinctive feature of the ML module facility is the use of *sharing constraints* to ensure that incrementally constructed systems are built from compatible components. The typical situation in which sharing specifications are required arises when defining a functor that builds a structure out of two argument structures, each of which are to have a third component in common. (MacQueen [28] gives an example in which a parser module is built from a lexer module and a symbol table module, each of which make use of a symbol module. In order for the parser to be well-defined, the lexer and the symbol table must *share* the same symbol implementation. See MacQueen [28] for more details.) Such a situation may be described schematically as follows. We are to define a functor F taking as argument two structures, R of signature SIG_R and S of signature SIG_S, which have a common component T of signature SIG_T. The arguments to F may be packaged into a single structure of signature SIG defined by

```
signature SIG =
  sig
    structure R. SIG_R
    structure S: SIG_S
  end
```

so that F may be introduced by a declaration of the form

```
functor F(X: SIG). SIG_F = ...
```

where SIG_F is the signature of the result of F. But this declaration is inadequate since it fails to ensure that R and S are built from a common substructure T. This is achieved by the use of a sharing constraint as follows:

```
signature SIG_share =
  sig
    structure R: SIG_R
    structure S: SIG_S
    sharing R.T = S.T
functor F(X. SIG_share)· SIG_F = ...
```

The signature SIG_share specifies that the component structures R and S share the *same* substructure T so that their use in the body of F is guaranteed to be sensible. An application of F to a structure is well-formed only if the type checker can determine that the required equational specification holds.

A simple way to account for sharing specifications in XML would be to employ the notion of an *equality type* introduced by Martin-Löf [31]. Informally, the equality type $M =_{\sigma} N$, for σ a U_2 type, is inhabited iff M and N are equal elements of type σ, according to the rules of equality for XML. The typing and equality rules for the equality type appear in Table XII. Signatures with sharing constraints are represented using equality types as follows. The signature SIG_share above is represented by the type

$$\Sigma R\colon \sigma_R . \Sigma S\colon \sigma_S . p(R) =_{\sigma_T} q(S)$$

where σ_R, σ_S, and σ_T represent the corresponding ML signatures, and where p and q are suitable compositions of projections to select the component of R and S, respectively, corresponding to their common component T.

Although this approach seems appealing at first glance, equality types fail to account for ML sharing specifications in two important respects. First, they are far more expressive than ML sharing specifications since they allow the imposition of arbitrary equational constraints, in contrast to ML which admits sharing constraints only between "paths," which are represented in XML as compositions of projection functions. This restriction to equations between paths seems essential, as illustrated by the following example. It is well known that recursion is definable in the untyped lambda calculus, via the fixed-point operator Y, and that the untyped lambda calculus may be interpreted in a typed lambda calculus satisfying an equation $t = t \rightarrow t$ between types. (Further discussion of Y may be found in Barendregt [2], for example, and the relationship between untyped lambda calculus and type (or domain) equations in Bruce and Meyer [4] and Scott [57].) Given this, and the fact that equality types allow us to type terms with respect to equational hypotheses, it is easy to show that equality types give us terms without normal form. For example, if Γ is a context containing the typing assumptions $x\colon \tau =_{U_1} \tau \rightarrow \tau$, for any U_1 type τ, then by the typing rules in Table XII, we may conclude that $\Gamma \triangleright \tau = \tau \rightarrow \tau\colon U_1$. Therefore, using the type equality rule from Table III, we may give any term with type τ type $\tau \rightarrow \tau$, and vice versa. This allows us to give any untyped lambda term type τ, including untyped terms with no normal form. Discharging the typing assumption via lambda abstraction, we can write a closed, well-typed functor with parameter $x{:}\{y\colon \mathit{triv} | \tau = \tau \rightarrow \tau\colon U_1\}$ and nonnormalizing body.

A second sense in which equality types are inappropriate is that they express semantic equivalence of structures, rather than the much more restricted notion of structure equivalence based on unique names described in Section 6. The type-theoretic account of modules given above does not attempt to account for ML notion of generativity, and hence cannot be readily extended to give a faithful account of ML sharing specifications. We consider

Table XII. Equality Type

$$\frac{\Gamma \triangleright \sigma : U_2 \quad \Gamma \triangleright M : \sigma \quad \Gamma \triangleright N : \sigma}{\Gamma \triangleright M =_\sigma N : U_2}$$

$$\frac{\Gamma \triangleright M = N : \sigma}{\Gamma \triangleright \mathit{refl}(M,N) : M =_\sigma N}$$

$$\frac{\Gamma \triangleright P : M =_\sigma N}{\Gamma \triangleright M = N : \sigma}$$

$$\frac{\Gamma \triangleright \sigma = \sigma' : U_2 \quad \Gamma \triangleright M = M' : \sigma \quad \Gamma \triangleright N = N' : \sigma}{\Gamma \triangleright M =_\sigma N = M' =_{\sigma'} N' : U_2}$$

$$\frac{\Gamma \triangleright M = M' : \sigma \quad \Gamma \triangleright N = N' : \sigma}{\Gamma \triangleright \mathit{refl}(M,N) = \mathit{refl}(M',N') : M =_\sigma N}$$

a proper account of ML notion of generative structure equality to be an important direction for future work.

10. CONCLUSION AND DIRECTIONS FOR FURTHER INVESTIGATION

We have given a precise description of the type system for much of ML, using a function calculus called XML. Our analysis is based on the belief that ML is profitably viewed as an explicitly-typed, predicative language with dependent product and sum types. Explicit typing is central to giving a single account of both the core expression language and the module system, and seems useful for further study. In particular, in papers of Moggi [48] and Harper et al. [21], which were written after the work described here was completed, XML is used to study the separation between compile-time and run-time in Standard ML. The distinction between U_1 and U_2 in XML reflects the typing rules of ML and leads to a number of significant technical simplifications in the study of the language. Moreover, universe distinctions seem essential to the character of ML, as discussed in Section 8.

Some important aspects of ML have been omitted. With regard to the core language, we have omitted treatment of recursion, references and exceptions. These language features raise important theoretical questions. We hope that an explicitly-typed study of polymorphic references would clarify the relationship between polymorphism and type inference, a continuing trouble spot in the ML type checker. With regard to the modules system, we have omitted treatment of the coercive aspects of signature matching, and of sharing specifications in signatures. It seems likely that the coercions associated with signature ascription may be accounted for in this framework by giving a precise account of compile-time elaboration as a process of translation from the ML concrete syntax into the abstract syntax of the XML calculus. Such a formalization would provide an interesting alternative to the methods used in the definition of ML [40]. Sharing specifications, and the associated notion of "structure generativity," remain important topics for further research.

Another important direction is to develop an accurate, straightforward presentation of ML operational semantics. As with other versions of lambda calculus, equality in XML is given by an equational axiom system. This equational system may also be formulated as a set of reduction rules, as usual. However, for the extension of XML obtained by adding exceptions, references and recursion, capturing the operational semantics of ML relies on careful consideration of the order in which rewrite rules are applied. (For example, if Ω is a divergent expression, then $(\lambda x.0)\Omega$ diverges in the current call-by-value implementation, but $(\lambda x.0)\Omega = 0$ is provable using the usual λ-calculus style reasoning.) It would be interesting to explore a typed calculus that is faithful to the operational semantics, following the pattern established by Plotkin's λ_v-calculus [51] and Martin-Löf's type theory [32]. Some useful related ideas are developed in Moggi [49].

ACKNOWLEDGMENTS

Thanks to Dave MacQueen for many insightful discussions of ML, and comments on this paper in particular. Thanks also to John Greiner, Peter Lee, Eugenio Moggi, and Andrzej Tarlecki for comments on an earlier draft.

REFERENCES

1. Amadio, R., Bruce, K., and Longo, G. The finitary projection model for second order lambda calculus and solutions to higher order domain equations. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, (1986), 122–130.
2. Barendreg, H. P. *The Lambda Calculus: Its Syntax and Semantics*. 2nd ed. North-Holland, Amsterdam, 1984.
3. Breazu-Tannen, V., Coquand, T., Gunter, C. A., and Scedrov, A. Inheritance as explicit coercion. *Inf. Comput. 93*, 1 (1991), 172–221.
4. Bruce, K., and Meyer, A. A completeness theorem for second-order polymorphic lambda calculus. In *Proceedings of the International Symposium on Semantics of Data Types* (Sophia-Antipolis, France) Springer Berlin, *LNCS 173*, 1984, 131–144.
5. Bruce, K. B., Meyer, A. R., and Mitchell, J. C. The semantics of second-order lambda calculus. *Inf. Comput. 85* 1 (1990), 76–134. Reprinted in *Logical Foundations of Functional Programming*, G. Huet, Ed., Addison-Wesley, Reading, Mass., 1990, 213–273.
6. Cardelli, L. A semantics of multiple inheritance. *Inf. Comput. 76* (1988), 138–164. Special issue devoted to *Symposium on Semantics of Data Types* (Sophia-Antipolis, France, 1984).
7. Cardelli, L. Structural subtyping and the notion of powertype. In *Proceedings of the 15th ACM Symposium Principles of Programming Languages* (1988), 70–79.
8. Cardelli, L., and Wegner, P. On understanding types, data abstraction, and polymorphism, *ACM Comput. Surv. 17*, 4 (1985), 471–522.
9. Cartwright, R. Types as intervals. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages* (Jan. 1985), 22–36.
10. Constable, R. L., et al. Implementing mathematics with the Nuprl proof development system. In *Graduate Texts in Mathematics* Vol. 37. Prentice-Hall, Englewood Cliffs, N.J., 1986.
11. Coquand, T. An analysis of Girard's paladox. In *Proceedings of the IEEE Symposium on Logic in Computer Science* (June 1986), 227–236.
12. Coquand, T., and Huet, G. The calculus of constructions. *Inf. Comput. 76*, 2/3 (1988), 95–120.

13. Damas, L., and Milner, R. Principal type schemes for functional programs. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages* (1982), 207–212.
14. De Bruijn, N. G. A survey of the project Automath. In *To H B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism.* Academic Press, New York, 1980, 579–607.
15. Girard, J.-Y. Interpretation fonctionelle et elimination des coupures de l'arithmetique d'ordre superieur. These D'Etat, Universite Paris VII, 1972.
16. Girard, J -Y. Une extension de l'interpretation de Godel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In *2nd Scandinavian Logic Symposium*, J. E. Fenstad, Ed., (North-Holland, Amsterdam, 1971), 63–92.
17. Gordon, M. J., Milner, R., and Wadsworth C. P. *Edinburgh LCF. LNCS 78*, Springer, Berlin, 1979
18. Harper, R., Honsell, F., and Plotkin, G. A framework for defining logics. In *Proceedings of the IEEE Symposium on Logic in Computer Science* (June 1987), 194–204. To appear in *J. ACM.*
19. Harper, R., MacQueen, D. B., and Milner, R Standard ML Tech. Rep. ECS-LFCS-86-2, Lab. for Foundations of Computer Science, Univ. of Edinburgh, Mar. 1986.
20. Harper, R , Milner, R., and Tofte, M. A type discipline for program modules. In *TAPSOFT 87, LNCS 250*, Springer, Berlin, 1987
21. Harper, R., Mitchell, J C , and Moggi, E. Higher-order modules and the phase distinction. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages* (Jan. 1990), 341–354.
22. Hook, J., and Howe, D. Impredicative strong existential equivalent to type·type. Tech. Rep. TR 86-760, Cornell Univ. 1986
23. Howard, W. The formulas-as-types notion of construction. In *To H. B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism* Academic Press, 1980, 479–490.
24. Howe, D. J. The computational behavior of Girard's paradox. In *Proceedings of the IEEE Symposium on Logic in Computer Science* (June 1987), 205–214.
25 Kanellakis, P. C., Mairson, H G., and Mitchell, J. C. Unification and ML type reconstruction In *Computational Logic, Essays in Honor of Alan Robinson.* MIT Press, 1991, 444–478.
26. Kfoury, A. J., Tiuryn, J., and Urzyczyn, P. ML typability is Dexptime-complete. In *Proceedings of the 15th Colloqium on Trees in Algebra and Programming. LNCS 431*, Springer, 1990, 206–220. To appear in *J ACM.* under the title, "An Analysis of ML Typability."
27 Leivant, D. Polymorphic type inference. In *Proceedings of the 10th ACM Symposium on Principles of Programming Languages* (1983), 88–98.
28. MacQueen, D. B. Modules for standard ML. *Polymorphism 2*, 2 (1985), 1–35. An earlier version appeared in *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming.*
29. MacQueen, D. B. Using dependent types to express modular structure. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages* (1986), 277–286
30. MacQueen, D., Plotkin. G., and Sethi, R. An ideal model for recursive polymorphic types. *Inf Control 71*, 1/2 (1986), 95–130.
31. Martin-Löf, P. An intuitionistic theory of types: Predictive part. In H. E. Rose and J. C. Shepherdson, Eds. *Logic Colloquium, '73*, Amsterdam, 1973, North-Holland, 73–118.
32. Martin-Löf, P. Constructive mathematics and computer programming In *Sixth International Congress for Logic, Methodology, and Philosophy of Science* North-Holland, Amsterdam, 1982, 153–175
33. Martin-Löf, P. *Intuitionistic Type Theory.* Bibliopolis, Napoli, 1984.
34. McCracken, N. An investigation of a programming language with a polymorphic type structure. Ph.D. Thesis, Syracuse Univ., 1979.
35. Meyer, A. R., Mitchell, J. C., Moggi, E., and Statman, R. Empty types in polymorphic lambda calculus. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages* (Jan. 1987), 253–262. Reprinted with minor revisions in *Logical Foundations of Functional Programming.* G. Huet, Ed , Addison-Wesley, 1990, 273–284.

36. MEYER, A. R., AND REINHOLD, M. D. Type is not a type. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages* (Jan. 1986), 287–295.
37. MILNER, R. A theory of type polymorphism in programming. *JCSS, 17* (1978), 348–375.
38. MILNER, R. The Standard ML core language. *Polymorphism 2*, 2 (1985), 1–28. An earlier version appeared in *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*.
39. MILNER, R., AND TOFTE, M. *Commentary on Standard ML.* MIT Press, 1991.
40. MILNER, R., TOFTE, M., AND HARPER, R. *The Definition of Standard ML.* MIT Press, 1990.
41. MITCHELL, J. C. A type-inference approach to reduction properties and semantics of polymorphic expressions. In *ACM Conference on LISP and Functional Programming* (Aug. 1986), 308–319. Reprinted with minor revisions in *Logical Foundations of Functional Programming*, G. Huet, Ed., Addison-Wesley, 1990, 195–212.
42. MITCHELL, J. C. Polymorphic type inference and containment. *Inf. Comput. 76*, 2/3 (1988), 211–249. Reprinted in *Logical Foundations of Functional Programming*, G. Huet, Ed., Addison-Wesley, 1990, 153–194.
43. MITCHELL, J. C. Representation independence and data abstraction. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages* (Jan. 1986), 263–276.
44. MITCHELL, J. C. Type systems for programming languages. In *Handbook of Theoretical Computer Science, Volume B*, J. van Leeuwen, Ed., North-Holland, Amsterdam, 1990, 365–458.
45. MITCHELL, J. C., AND MEYER, A. R. Second-order logical relations. In *Logics of Programs, LNCS 193*, Springer, Berlin, 1985, 225–236.
46. MITCHELL, J. C., AND MOGGI, E. Kripke-style models for typed lambda calculus. *Ann. Pure Appl. Logic 51* (1991), 99–124. Preliminary version in *Proceedings of the IEEE Symposium on Logic in Computer Science* (1987), 303–314.
47. MITCHELL, J. C., AND PLOTKIN, G. D. Abstract types have existential types. *ACM Trans. Program. Lang. Syst. 10*, 3 (1988), 470–502. Preliminary version appeared in *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*, 1985.
48. MOGGI, E. A category-theoretic account of program modules. *Math. Structures Comput. Sci. 1*, 1 (1991), 103–139.
49. MOGGI, E. Computational lambda calculus and monads. In *Proceedings of the IEEE Symposium on Logic in Computer Science* (1989), 14–23.
50. OHORI, A. A simple semantics for ML polymorphism. In *Functional Programming and Computer Architecture*, 1989, 281–292.
51. PLOTKIN, G. D. Call-by-name, call-by-value and the lambda calculus. *Theor. Comput. Sci. 1* (1975), 125–159.
52. PLOTKIN, G. D. LCF considered as a programming language. *Theor. Comput. Sci. 5* (1977), 223–255.
53. REYNOLDS, J. C. Towards a theory of type structure. In *Paris Colloqium on Programming, LNCS 19*. Springer, Berlin, 1974, 408–425.
54. REYNOLDS, J. C. The essence of Algol. In *Algorithmic Languages*, de Bakker and van Vliet, Eds. IFIP, North-Holland, Amsterdam, 1981, 345–372.
55. REYNOLD, J. C. Types, abstraction, and parametric polymorphism. In *Information Processing '83*, North-Holland, Amsterdam, 1983, 513–523.
56. REYNOLDS, J. C. Polymorphism is not set-theoretic. In *Proceedings of the International Symposium on Semantics of Data Types* (Sophia-Antirolis, France), *LNCS 173*, Springer, Berlin, 1984, 145–156.
57. SCOTT, D. S. Relating theories of the lambda calculus. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism.* Academic Press, 1980, 403–450.
58. SEELY, R. A. G. Locally cartesian closed categories and type theory. *Math. Proc. Camb. Phil. Soc. 95* (1984), 33–48.
59. SEELY, R. A. G. Categorical semantics for higher-order polymorphic lambda calculus. *J. Symbolic Logic 52* (1987), 969–989.
60. STATMAN, R. Logical relations and the typed lambda calculus. *Inf. Control 65* (1985), 85–97.
61. STOUGHTON, A. *Fully Abstract Models of Programming Languages.* Pitman, London, and Wiley, New York, 1988.

62. TROELSTRA, M. Mathematical investigation of intuitionistic arithmetic and analysis. *LNM 344*, Springer, Berlin, 1973.

63 TOFTE, M. Operational semantics and polymorphic type inference. Ph.D. dissertation, Edinburgh Univ., 1988. Available as Edinburgh Univ. Laboratory for Foundations of Computer Science Tech Rep. ECS-LFCS-88-54

64. WAND, M. A types-as-sets semantics for Milner-style polymorphism. In *Proceedings of the 11th ACM Symposium on Principles of Programming Languages* (Jan 1984), 158–164.

Received May 1990; revised February 1992; accepted March 1992