LFCS

Laboratory for Foundations of Computer Science
Department of Computer Science - University of Edinburgh

The Essence of ML

by

John C. Mitchell and Robert Harper

The Essence of ML

ECS-LFCS-87-42
(also published as CSR-250-87)

LFCS Report Series

LFCS
Department of Computer Science
University of Edinburgh
The King's Buildings
Edinburgh EH9 3JZ

November 1987

The Essence of ML*

John C. Mitchell
AT&T Bell Laboratories
Murray Hill, NJ 07974

Robert Harper
Edinburgh University
Edinburgh, Scotland

November 24, 1987

Abstract

Standard ML is a useful programming language with polymorphic expressions and a flexible module facility. One notable feature of the expression language is an algorithm which allows type information to be omitted. We study the implicitly-typed expression language by giving a "syntactically isomorphic" explicitly-typed, polymorphic function calculus. Unlike the Girard–Reynolds polymorphic calculus, for example, the types of our ML calculus may be built-up by induction on type levels (*universes*). For this reason, the pure ML calculus has straightforward set-theoretic, recursion-theoretic and domain-theoretic semantics, and operational properties such as the termination of all recursion-free programs may be proved relatively simply. The *signatures, structures,* and *functors* of the module language are easily incorporated into the typed ML calculus, providing a unified framework for studying the major features of the language (including the novel "sharing constraints" on functor parameters). 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.

1 Introduction

Recent years have seen increasing interest, in computer science, in various forms of typed λ-calculi. One motivation for studying these systems 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 [Gir72,Rey74,MP85]. The richer type systems proposed by Martin-Löf [Mar82], Constable [C*86], and Huet and Coquand [CH84]

*Reprint of a paper presented at the Fifteenth ACM Symposium on Principles of Programming Languages, San Diego, California, January, 1988. All citations should refer to the conference proceedings.

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 meta-language of type theory is somewhat closer to the object languages studied, and there is current interest in re-examining the status of nontermination and $\perp$. 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" [Rey81], 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 [HMM86], 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.

Standard ML is an updated version of the programming "meta–language" of the LCF system [GMW79], comprising a core expression language with polymorphic functions [Mil85] and a module language for defining interdependent program units [Mac85]. 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. We believe that Standard ML may be characterized as follows.

- The type system is a polymorphic lambda calculus with two levels (universes). In addition to polymorphic function spaces, the type constructors include strong sums (described in Section 7), recursive types at the first level, and limited use of equalizers (described in Section 9) at the second level.

- Exceptions provide a means for escaping any control structure to a point where the corresponding exception is trapped.

- The operational semantics are given by a deterministic evaluator, for which the type system is sound with respect to normal termination. In other words, if the evaluator halts with no exception raised, then the type of the result is guaranteed by the typing rules.

- References are provided using the syntax of type constructors, and any value of the first level may be stored in a reference cell.

- Values of each universe are "first-class" in the sense that any operations appropriate to that level may be applied. For example, any element of any type from the first level may be stored in a reference cell, and functors may take any values from the second level as parameters. This is not strictly true of Standard ML as currently implemented, since functors may not take functors as arguments. However, we consider this an implementation restriction, and not an essential feature of the language.

There are many other aspects of ML that contribute to its utility. For example, the combination of named constructors in `datatype` declarations and pattern-matching in function definitions leads to a succinct and powerful programming style. However, this syntax does not seem *essential;* ML without pattern matching would still be ML.

For the reader familiar with Reynolds' study of Algol, the following comparisons may be helpful.

- Idealized Algol leaves expression evaluation undetermined, whereas deterministic evaluation seems central to the way exceptional and normal termination are defined in Standard ML.

- Storage allocation in Idealized Algol adheres to a stack discipline, while Standard ML does not.

- Idealized Algol distinguishes storable values from denotable values (*c.f.*[Gor79]), while Standard ML treats data types and phrase types uniformly.

- Idealized Algol has labels and goto's, which are replaced by the more structured exception mechanism of Standard ML.

While a thorough treatment of evaluation order is beyond the scope of this paper, it may be worthwhile to clarify a few points. Reynolds argues that the procedure mechanism of Algol is call-by-name, but the order of expression evaluation should be regarded as indeterminate. We also believe that in one respect, Standard ML evaluation is order dependent, and in another respect, order independent. However, our point of view is based on an entirely different sort of distinction. Reynolds' view of Algol evaluation relies on the separation between Algol phrase types and Algol data types. Since ML constructs are uniformly applicable to values of all types (at least within the first universe), it is difficult to separate procedures from expressions. Therefore, ML must be viewed as having a single form of run-time evaluation. Since the definition of normal termination, the behavior of exceptions, and the straightforward explanation of side-effects all seem to depend on evaluation order, we believe that full "run-time" evaluation of ML programs is most successfully explained using a deterministic evaluation strategy.

Due to the complexity of Standard ML, there are actually two phases of evaluation in current implementations. The module language provides declarations that bind identifiers to structured values with type components, and these identifiers often occur in type expressions. Consequently, non-trivial evaluation is involved in determining equality of type expressions. During the type-checking phase of compilation, only certain compound forms in type expressions are evaluated, and so compile-time evaluation appears to be finite Church-Rosser, as a consequence of the finiteness of developments [Bar84, Theorem 11.2.25]. Thus, we believe that while "run-time" evaluation is order dependent, order of evaluation may safely be regarded as indeterminate for the purpose of type checking.

The main focus of this paper will be the type system of Standard ML, as this seems a prerequisite for more comprehensive analysis. 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 most 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 [Mil78], and the ideal model of [MPS86] 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 [Mar82], and in keeping with the suggestions of [Mac86]. 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 with higher rank than their arguments.) The universe distinctions are faithful to the separation of *monotypes* from *polytypes* in Milner's earlier work [Mil78,DM82], 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 [Mit87,MPS86,Car] and the explicitly–typed polymorphic calculus of [Gir71,Gir72,Rey74]. In particular, the pure ML calculus without recursion has classical set–theoretic models, while the Girard–Reynolds calculus does not [Rey84].

Some studies of ML typing (*e.g.*, [Car,Mit87,MPS86]) 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 [Gir71,Gir72,Rey74]. 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 [Mac86]. 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, [Coq86,Gir72,How87,MR86], that there exist recursion-free programs that cannot be evaluated to a normal form by any evaluation strategy. As argued in [MR86], this alters the character of the language dramatically. Therefore, we believe that the separation of types into universes is essential to ML.

As an artifact of the way we study universe distinctions, the bulk of this paper will not be concerned with evaluation order. 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. However, 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 or full abstraction would be complicated dramatically by a type of all types.

The next section contains a short summary of the accepted type inference rules for the core language of Standard ML. In Section 3, an alternative, explicitly-typed core language is given. The two approached are proved equivalent in Section 4, and the semantics of the core language is discussed in Section 5.

Sections 6 through 9 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 Section7. Section 8 considers the importance of universe distinctions and ***type:type***, and Section 9 presents an extension to the type system that includes the "sharing" constraints of MacQueen's module language. Finally, Section 10 discusses the problems of representation independence and full abstraction, with concluding remarks in Section 11. 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 [Mil78,DM82,MPS86,Wan84]. This algorithm allows the ML programmer to write, for example,

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

automatically inferring the fact that the function *id* is a function ¿from type t to t, for any t. Milner's seminal paper [Mil78] 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.

The syntactic part of Milner's analysis is refined in [DM82], 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 [MPS86,Car], where the meanings of polymorphic types are clarified and recursive types are given semantics (see also [Mit87]). 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* [Mil78].

Although there are quite a few constructs in the core expression language of Standard ML, the behavior of the type checker may be understood by considering the fragment presented in [DM82], which we will call *Core*-ML . The syntax of *Core*-ML is given by

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

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 polymorphic expressions have had all type variables implicitly bound. 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 by

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

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, non–polymorphic type. The *polymorphic* type expressions (also called *type schemes*) are defined by

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

Intuitively, the elements of type $\forall t.\sigma$ have type σ for every possible value of the type variable t (which will generally occur in σ). Since $\forall$ binds t in σ, we have $\forall t.\sigma = \forall s.[s/t]\sigma$, where $[s/t]\sigma$ denotes substitution of s for free occurrence of t in σ, as usual. Note that every monomorphic type is also considered a polymorphic type.

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 expressions will generally depend on the types given to free variables, we will use typing statements that incorporate such assumptions. A *type assignment*, or *context*, Γ, is a finite sequence 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 $Dom(\Gamma)$ for the set of expression variables occurring in Γ, and write $\Gamma, x{:}\sigma$ (where $x \notin Dom(\Gamma)$) for the type assignment obtained by appending the indicated binding to Γ. 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 1. 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* in a context Γ iff there is a type scheme σ such that $\vdash_{\mathrm{DM}} \Gamma \triangleright e : \sigma$.

A type σ is a *substitution instance* of a 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 $\sigma = \forall t_1 \ldots t_n.\tau'$, written $\sigma \sqsubseteq \tau$, iff there is a substitution S of monotypes for $t_1 \ldots t_n$ 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 $\forall s_1 \ldots s_k.\tau'$ of σ' such that no s_i $(1 \leq i \leq k)$ occurs free in σ and $\sigma \sqsubseteq \tau'$; see [DM82,Mit87] for further discussion, and [Mit87] 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* 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 every typable expression has a principal typing, ¿from which all other typings may be efficiently derived. A typing $\Gamma \triangleright e : \sigma$ is a *principal typing for expression* e if $\vdash_{\mathrm{DM}} \Gamma \triangleright e : \sigma$ and, whenever $\vdash_{\mathrm{DM}} \Gamma' \triangleright e : \sigma'$, we have $\sigma \sqsubseteq \sigma'$ and $\Gamma'(x) \sqsubseteq \Gamma(x)$ for every x occurring in Γ. In other words, the principle typing for e must be derivable, and it must give the most general type in the "least general" context.[1]

Theorem 2.2 (Damas–Milner) *If an untyped Core-ML expression e is Damas-Milner typable, then there exists a principal typing* $\Gamma \triangleright e : \sigma$. *Furthermore, there is a linear-time algorithm which, given e, computes the principle typing if it exists, and* fails *otherwise.*

[1]This formulation of principal typing differs ¿from [DM82] in that the latter defines the principal typing with respect to a *given* context. Further discussion of this point may be found in [Lei83].

3 Explicitly Typed ML

In contrast to the Milner–style analyses of ML, we will view ML programs as *explicitly* typed, in the sense that variables are explicitly assigned a type at the point where they are bound, and all type manipulations are explicit. Essentially, 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. One reason for taking this position is that the untyped approach does not scale up to include features of the modules system. Another reason is that it leads to significant technical simplifications in the semantics of the language.

Implicitly–typed *Core*-ML is essentially equivalent to an explicitly–typed function calculus we will call *Core*-XML , for *core explicit ML.* The types of *Core*-XML fall into two classes, corresponding to the monomorphic types and polymorphic type 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 are defined by quantifying 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. Note that since every monomorphic type is also considered a polymorphic type (with binding of zero variables), we effectively have $U_1 \subseteq U_2$.

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 un–checked *pre–terms* of explicitly typed *Core*-XML are given by the following grammar:

$$\begin{array}{lcl} M & ::= & x \mid MM \mid \lambda x{:}\tau.M \mid M\tau \mid \lambda t.M \\ & & \mathbf{let}\ x{:}\sigma = M\ \mathbf{in}\ M \end{array}$$

Following Damas–Milner, we retain **let** as a primitive construct in *Core*-XML since its typing rule is not derivable from the other rules. However, in the full XML system it will be definable in terms of abstraction and application, and will therefore be eliminated.

The type checking rules of the language are listed in Table 2. To distinguish implicit *Core*-ML typing from explicit *Core*-XML typing, we will write $\vdash_{\mathrm{X}} \Gamma \triangleright M : \sigma$ if the typing $\Gamma \triangleright M : \sigma$ is derivable from the *Core*-XML typing rules. The essential difference between $\vdash_{\mathrm{DM}}$ and $\vdash_{\mathrm{X}}$ is that the GEN and SPEC rules of the implicit system are replaced by rules for explicit type abstraction and type application. A pre–term M is a *term* of *Core*-XML if $\vdash_X \Gamma \triangleright M : \sigma$ for some Γ and σ.

The difference between *Core*-XML and the Girard–Reynolds polymorphic λ–calculus [BMM98,Rey74,Gir72] lies in the distinction between universes U_1 and U_2. Rule TAPP of *Core*-XML only allows a type application $\Gamma \triangleright M\tau : [\tau/t]\sigma$ 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 [Rey84].

The language *Core*-XML differs from Martin-Löf's type theory and the Nuprl system [C*86] not only in that we do not have $U_1{:}U_2$ (which will change when we come to full XML), but also because types are explicitly assigned to all identifiers at their binding occurrences. For example, rule ABS of *Core*-XML only allows us to type a lambda abstraction $\lambda x{:}\tau.M$ if the type of the bound variable x is explicitly declared. In contrast, Nuprl and Martin-Löf's type theory give types to expressions like $\lambda x.M$. Without explicit typing, *Core*-XML 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.

Equations have the form $\Gamma \triangleright M = N : \tau$, where M and N are terms of type τ (in the context Γ). The equational proof system of *Core*-XML is similar to that of the Girard–Reynolds calculus [Gir71,Rey74,BMM98], with the following additional axiom for **let**:

$$\Gamma \triangleright (\textbf{let } x{:}\sigma = M \textbf{ in } N) = [N/x]M : \tau$$

A complete presentation of the equational inference system is omitted due to lack of space.

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 [Lei83, Section 7]. However, the technical 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 a typed term M is defined as follows:

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

Theorem 4.1 *If* $\vdash_{\mathrm{X}} \Gamma \triangleright M : \sigma$, *then* $\vdash_{\mathrm{DM}} \Gamma \triangleright M^\circ : \sigma$.

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

Theorem 2.2 states that there is an algorithm which finds, for any typable expression e, a principal typing for e. It is a simple matter to modify this algorithm so that it produces as well a derivation of the principal typing in the Damas–Milner system. Applying Theorem 4.2, we obtain an algorithm X that, given a typable expression e, yields an explicitly typed term M such that $\vdash_{\mathrm{X}} \Gamma \triangleright M : \sigma$ (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 $\forall t_1 \ldots t_n.\tau$. For example, the explicitly-typed term produced from

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

is

$$\lambda t.\textbf{let } I{:}\forall t.t \to t = \lambda t.\lambda x{:}t.x \textbf{ in } I[t \to t](I[t]).$$

Note that the principal type of **let** $I = \lambda x.x$ **in** II is $\forall t.t \to t$, which is the type of the explicitly-typed term. The syntactic completeness of Algorithm X follows from that of the type inference algorithm.

It is worth pointing out that since *Core*-XML has **let** as a primitive, the translation used in Theorem 4.2 does not alter the structure of terms. In particular, we do not treat **let** $x = N$ **in** M as an abbreviation for $[N/x]M$. This distinguishes our translation from the explicit typing translation of [Wan84].

5 Semantics of Core–XML

The *Core*-XML language has a straightforward model theory that is similar to the semantics of second-order lambda calculus described in [BM84,BMM98,Mit87], except that we have two universes instead of one collection of types. 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 $U_1 \subseteq U_2$. Due to space considerations, precise definitions are omitted.

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 in [ABL86,BMM98,Gir72,Tro73,McC79,M One difference between the languages, however, is that *Core*-XML has classical set-theoretic models, while the Girard-Reynolds calculus does not [Rey84]. In fact, any model of ordinary (non-polymorphic) typed lambda calculus may be extended to a model of *Core*-XML by a simple set-theoretic construction.

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; see [Mit86b] for further discussion and references). The ideal model of [MPS86], for example, can be viewed as a PER inference model, as defined in [Mit87], by replacing each ideal I with the partial equivalence relation $I \times I$. By the results of [Mit86b], 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 [Mil78], 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, but type membership interpreted as expected. Thus a consequence of the type soundness theorem for *Core*-XML models (see, *e.g.*, [Mit86b,Mit87]) is that *Core*-ML expressions "cannot go wrong."

A final, related topic is the theory of logical relations for *Core*-XML , along the lines of [MM85,Sta85]. Since universes allow us to construct relations by induction on types, strong normalization for *Core*-XML may be proved by a relatively straightforward extension of the argument given in [Bar84, Appendix A]. However, it is worth pointing out that strong normalization for *Core*-XML follows from strong normalization of the Girard-Reynolds calculus, and so is also a corollary of the theorem originally proved by Girard [Gir71,Gir72,Mit86b].

6 Review of Standard ML Modules

An important part of Standard ML is the module system, proposed in [Mac85] and further documented in [HMM86]. We will review the main features of the module design by example and then, in the next section, describe the full XML function calculus encompassing modules.

The basic units of the module system are called ***structures***, ***signatures*** and ***functors***. A structure is essentially a (heterogeneous) environment. For our purposes, an environment assigns meaning to value identifiers (by mapping them to values), to type identifiers (by mapping them to types), and to structure identifiers (by mapping them to structures). The type of a structure is a signature, which lists the identifiers bound by the structure and their types. For example, the signature of a structure binding x to 3 will specify that x has type int. Since structures may bind type and structure identifiers as well as value identifiers, the notion of "type" must be extended to encompass these cases as well, as we will see below. Functors are functions mapping structures to structures, but currently there are no explicit functor signatures in Standard ML.

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 entirely "first-class" in that they may only be bound to structure identifiers or passed as arguments to functors. However, 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 = 7
  end
```

The structure expression following the equals sign defines an environment mapping t to int and x to 7. The way Standard ML is currently implemented, structure expressions are "generative," which means that for the purpose of type checking, each structure elaboration yields a distinct structure. The reason for this is that the modules language allows a user to specify that two structures must be equal, using sharing specifications, and so structure equality must be made efficiently decidable. More will be said about this in Section 9.

The components of a structure are accessed by qualified names, using a syntax reminiscent of record access in many languages. For instance, the identifier S.x refers to the x component of S, and therefore has value 7. Similarly, S.t refers to the t component of S, and denotes the type int. The typing rule for qualified names, which is made precise using signatures, gives S.x type S.t. During the type checking phase of compilation, S.t will be evaluated to int, when necessary, so that S.x may be used as an integer expression. This "transparency" of type definitions distinguishes Standard ML structures from abstract data type declarations (see [MP85] for related discussion).

Signatures are denoted by signature expressions, the basic form of which is the specification. Signatures may be bound to signature identifiers using a signature binding, as follows.

```
signature SIG =
  sig
```

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

This signature asserts that t is a type identifier, and that x is a value identifier of type t. It should be clear that SIG is a legitimate signature for the structure S defined above.

It is worth remarking that there may be many distinct signature expressions that accurately describe a given structure. For example, the signature

```
signature SIG' =
  sig
    type t
    val x : int
  end
```

is also an acceptable specification of the structure S, for the value of x is in fact an integer.

In addition to ambiguities of this form, there is another, more practically–motivated, reason why structures do not possess unique (or most general) signatures. It seems important in practice that signatures be interpreted as *constraints* or *views* of a structure, rather than precise specifications. For example, in ML the structure S defined above also matches the signature

```
signature SIG'' =
  sig
    val x : int
  end
```

since it clearly provides a component x of type int. This is related to the form of subtyping discussed in [Car84,CW86]. Although structures do not have unique signatures, it is shown in [HMT87b] that every structure has a "most general" signature, much as each expression of the core language has a most general type. For further discussion of signature matching, we refer the reader to [HMT87b].

To simplify our treatment of Standard ML, we will simplify signature matching so that each structure has a unique signature listing all of its components and their types. This allows us to view the more general form of signature matching as a convenience of concrete syntax, much as we view the implicitly-typed *Core*-ML as a shorthand for *Core*-XML . In particular, we regard the signature matching algorithm of Standard ML as providing the automatic insertion of coercion functions that restrict structures to the required signatures.

Functors (which are functions mapping structures to structures) are generally declared using a syntax reminiscent of the typed function declarations used in many languages:

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

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. The functor F takes a structure matching the signature SIG (defined above), and returns a structure matching the same signature, but with the type t of the result being the Cartesian product of S.t with itself, and, correspondingly, with x being the pair (S.x,S.x).

A peculiarity of the module system is that signatures are not allowed to have free type variables. This implies that a result signature cannot refer to a parameter or its signature, and hence certain forms of dependency cannot be expressed. For example, the following declaration is not legal in ML, since the signature describing the result of functor application contains a free occurrence of S.

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

We would like to have S bound by the functor expression, but this is prohibited by current implementations. However, with the result signature removed, the declaration above becomes legal Standard ML. This anomaly, which is at variance with the account given in [Mac86], appears to be an oversight arising from the lack of functor signatures. To avoid this problem, and to provide a more uniform language, we will incorporate functor signatures into our model of ML.

7 Full XML

In this section we will extend *Core*-XML to a function calculus XML by adding general constructs that allow us to describe signatures, structures and functors. Following [Mac86], we will use general sums and products in the style of Martin-Löf's type theory [Mar82]. While general sums (also called "strong sums;" see [How80]) are closely related to structures, and general cartesian products seem necessary to capture functors, the language XML will be somewhat more general than Standard ML. For example, an ML structure may contain polymorphic functions, but there is no way to define a polymorphic structure in the implicitly-typed programming language. This is simply because there is no provision for explicit binding of type variables. 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. Although higher-order functors are omitted from ML, they seem to be useful for supporting separate compilation. And, as mentioned in the last section, functor signatures smooth over some trouble spots. Consequently, we consider the introduction of these features a benefit of XML. In Section 9, we will discuss the addition of sharing constraints.

Intuitively, general sums and products may be viewed as straightforward set-theoretic constructions. If A is an expression defining some collection (either a type or universe, for example), and B is an expression with free variable x which defines a collection for each $x \in A$, then $\Sigma x{:}A.B$ and $\Pi x{:}A.B$ are called the *sum* and *product of the family B over the index set A*, respectively. The product $\Pi x{:}A.B$ is the Cartesian product of the family $\{ B(x) \mid x \in A \}$; its elements may be regarded as functions f such that $f(a) \in [a/x]B$ for each $a \in A$. The sum $\Sigma x{:}A.B$ is the disjoint union of the family $\{ B(x) \mid x \in A \}$; its members are defined to be ordered pairs $\langle a, b \rangle$ with $a \in A$ and $b \in [a/x]B$. Since the elements of sum types are pairs, general sums come with projection functions *fst* and *snd* for first and second components.

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 declaration of structure `S` in the last section, with `t` = `int` and `x` = `7`, may be viewed as binding `S` to the pair $\langle int, 7 \rangle$. In XML, the components `t` and `x` are retrieved by projection functions, so that `S.t` is regarded as an abbreviation for $snd(S)$. With general sums we can represent the signature `SIG` as the type $\Sigma t{:}U_1.t$, of which the tuple $\langle int, 7 \rangle$ 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. However, this way of viewing structures does not provide any natural interpretation of Standard ML `open`, which imports declarations ¿from the given structure into the current scope. Therefore, it would be preferable to define a type system with environments as part of the syntax.

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 example, the functor `F` of the last section is defined by the expression

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

which has type

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

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 higher-order functors.

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. 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 [DB80] and Martin-Löf [Mar82]. The un-checked pre-terms of XML are given by the following grammar:

$$\begin{array}{rcl} M & ::= & x \mid U_1 \mid U_2 \mid \mathrm{triv} \mid M \to M \mid \Pi x{:}M.M \\ & & \Sigma x{:}M.M \mid * \mid \lambda x{:}M.M \mid MM \mid \langle M, M\rangle \\ & & \mathit{fst}(M) \mid \mathit{snd}(M) \end{array}$$

The metavariables M, N, and P range over the pre-terms. We also use σ and τ to range over pre-terms, particularly when the term is intended to be a type. The observant reader will note that we no longer have **let**. This is because **let** $x{:}\sigma = M$ **in** N may be written as $(\lambda x{:}\sigma.N)M$ in XML, using lambda abstraction over the polymorphic type $\sigma{:}U_2$. The type checking rules for XML appear in Tables 3 through 7.

The following lemma summarizes some technical properties of the type system.

Lemma 7.1 *1. If* $\vdash \Gamma \triangleright M : \tau$, *then* $\vdash \Gamma$ context.

2. If $\vdash \Gamma \triangleright M : \tau$, *then either* $\tau \equiv U_2$ *or* $\vdash \Gamma \triangleright \tau : U_2$.

3. If $\vdash \Gamma, x{:}\tau \triangleright M : \sigma$ *and* $\vdash \Gamma \triangleright N : \tau$, *then* $\vdash \Gamma \triangleright [N/x]M : [N/x]\sigma$.

Strong normalization for XML may be proved using a translation into Martin-Löf's 1973 system [Mar73]. We consider it an important open problem to develop a theory of logical relations for full XML, a task that is significantly complicated by the presence of general Σ and Π types.

While we have outlined the translation of Standard ML signature, structure and functor expressions into XML, the status of the corresponding declarations deserves further explanation. The most natural treatment of these declarations might seem to be via lambda abstraction, regarding

```
structure S: SIG = struct ⟨body⟩ end;
⟨program⟩
```

as meaning

$(\lambda \mathtt{S{:}SIG}.\ \langle \mathit{program}\rangle)\langle \mathit{body}\rangle.$

However, this does not model the behavior of the Standard ML type checker accurately, since the program must be typed without knowing the values of typed declared in S. In fact, this form of parameterization seems to provide data abstraction, as noted in [Mac85].

An important aspect of Standard ML is that signature and functor declarations may only occur at "top level," which means they cannot be embedded in other constructs, and structures may only be declared inside other structures. Furthermore, recursive declarations are not allowed. Consequently, it is possible to treat signature, structure and functor declarations by simple macro expansion. By this process, the program

```
structure S =
  struct
    type t = int
    val x : t = 7
  end;
S.x + 3
```

may be typed by observing that the type of `S.x` is `S.t` $= fst(\langle int, 7\rangle)$, which may be simplified to *int*. Thus the first step in translating a Standard ML program into XML is to replace all occurrences of signature, structure and functor identifiers with the expressions to which they are bound. Then, type expressions may be simplified using the type equality rule of Table 4, as required. With the exception of "generativity," which may be regarded as a means of characterizing a particularly simple approximation to XML equality (*c.f.*[HMT87b,HMT87a]), this process models elaboration during the Standard ML type checking phase fairly accurately.

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 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 assumptions $U_1 \subseteq U_2$ and $U_1{:}U_2$ are essentially benign, and that in the presence of structures and functors, the universe restrictions are essential if we wish to avoid a type of all types. Since there is insufficient space to debate the merits of *type*:*type*, we refer the reader to [Coq86,How87,MR86] for background information and further discussion. Based on previous investigation, 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 is 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

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

for the Π-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 appropriate 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*

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

where triv *may be any type, with the following properties:*

- $\Gamma \rhd i[M] : \Pi t{:}U_1.\tau$ *whenever* $\Gamma \rhd M : \tau$
- $\Gamma \rhd j[M] : \tau$ *whenever* $\Gamma \rhd M : \Pi t{:}U_1.\tau$
- $\Gamma \rhd j[i[M]] = M : \tau$ *for all* $\Gamma \rhd M : \tau$.

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 $(\Pi t{:}U_1.\tau) : U_2$, 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 presentation, 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*

$$i[\,] \equiv \langle [\,], * \rangle \quad \textit{and} \quad j[\,] \equiv \mathit{fst}[\,],$$

where triv *may be any U_1 type with closed term* $*{:}\mathrm{triv}$, *satisfying the following conditions.*

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

Intuitively, the lemma say 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 $\Sigma t{:}U_1.\mathrm{triv}$. 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 Standard 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 [Gir71,Gir72,Rey74]. (A similar technique is used to introduce impredicativity into Nuprl in [How87].) The Girard–Reynolds calculus has a number of reasonable theoretical properties (see, e.g., [BMM98,Mit86b,MP85]) and seems to be a useful tool for studying polymorphism in programming languages.

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 [MR86] becomes a sublanguage of XML.

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 [MR86].*

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.

By Lemma 8.2, we know that sums over U_1 give us $U_1{:}U_2$. Therefore, we have 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 $\lambda^{\tau:\tau}$, relaxing the universe restrictions of XML would be seem distinct departure from ML.

8.5 Trade–off between weak and strong sums

When we first discovered Theorem 8.4, we announced it as a trade–off theorem in programming language design[2]. The "trade–off" 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 (sometimes called "strong sums" [How80]) allow us to type more terms than the existential types associated with data abstraction (see [MP85]).

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 [Gir72,Mit86b][3], we know that impredicative polymorphism with existential types is strongly normalizing. As noted in Section 7, a translation into Martin-Löf's 1973 system [Mar73] 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 [Coq86,MR86,How87], *type*:*type* (in the presence of other constructs) implies that strong normalization fails. In short, assuming we wish to avoid *type*:*type* and non-normalizing recursion-free terms, we have a trade–off 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

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

[2]We described our "trade–off 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 [HH86]. We also learned that Coquand had proved the same theorem by a different means in [Coq86], which in preparation at the time of our announcement.

[3]Girard's original proof included existential types. While the somewhat simpler proof in [Mit86b] does not, normalization with existential types can easily be derived by encoding $\exists t.\sigma$ as $\forall r[\forall t(\sigma \rightarrow r) \rightarrow r]$.

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. We do not know if this extension of XML is strongly normalizing.

9 Sharing

A distinctive feature of ML's modules facility is the *sharing constraint* used to ensure that incrementally constructed systems are built from compatible components. For example, if the functor F builds a structure T as a function of two structures R and S, it may be necessary for a type R.t defined in R to be the same as the type S.t defined in S. This is specified as follows.

```
functor F ( R : SIGR, S : SIGS sharing R.t=S.t) = ...
```

where SIGR and SIGS are suitable signatures for R and S. An application of F to two structures R and S is legal iff the t component of each is the same type. Sharing specifications are not limited to types; one can also require that two component structures of R and S be equal. However, equations are syntactically restricted to equations between "paths", which are either simple identifiers, or identifiers qualified by one of more component designations (*e.g.*, S.p.q.r is a path, but $\mathtt{S.x}+3$ is not).

Sharing specifications may be added to XML using equalizer types, which are an adaptation of a well-known idea from category theory [LS86,ML71]. Informally, the equalizer type

$$\{\, x{:}\sigma \mid M = N{:}\tau \,\}$$

is the collection of all elements x of type σ satisfying the equation $M \doteq N : \tau$, which may involve the variable x. It is worth mentioning that equalizers are also part of the Nuprl type theory [C*86], arising from the combination of *set* and *equality* type constructors. More specifically, the Nuprl the set type $\{\, x{:}\sigma \mid \tau \,\}$ consists of those elements a of type σ such that the type $[a/x]\tau$ is inhabited, and the equality type $a = b \,\mathrm{in}\, \sigma$ is inhabited exactly when a and b are equal elements of type σ. So it is easy to see that our equalizer types are the composition of two Nuprl types. The typing rules for equalizers appear in Table 7.

Functors with sharing constraints are written in XML by restricting the domain to an appropriate equalizer type. For example, the Standard ML functor

```
functor F ( P : SIG1 sharing P.s = P.t ) : SIG2 = ...
```

requires a structure P whose s and t components are equal. If SIG1 declares that both s and t are types, s is the first component of P, and t is the second, then this functor is given by an XML term of the form

$$\lambda P{:}\{\, x{:}\sigma_1 \mid \mathit{fst}(x) = \mathit{snd}(x) : U_1 \,\}.\ \ldots$$

where σ_1 is the XML notation for SIG1.

In practice, as Standard ML is currently implemented, equality for the purposes of type checking is based on the idea of generativity mentioned briefly in Sections 6 and 7. The reasons for this are discussed in [Mac85,HMT87b], and are related to the algorithmic difficulty pointed out in [Sol78]: structure and type equality are checked at compile time, and must be made efficiently decidable. While the full system of equational inference rules is omitted from this paper due to space considerations, the intended rules parallel those in [BMM98,Gir72,MR86,Mit87,Rey74] and capture a semantic notion of equality. The equations used in Standard ML type checking constitute one of many decidable approximations to XML equality, and the effect of sharing constraints varies according to the kind of approximation used. The consequences of introducing equalizers, in the context of the particular equational reasoning used in the current type checker, remain a topic for further investigation.

As mentioned above, only a restricted form of equalizer may appear in Standard ML programs. One justification for the restriction to paths is that strong normalization fails otherwise, as follows. It is well known that recursion is definable in the untyped lambda calculus, via the fixed-point operator Y, and that untyped lambda calculus may be interpreted in typed lambda calculus satisfying an equation $t = t \rightarrow t$ between types. (Further discussion of Y may be found in [Bar84], for example, and the relationship between untyped lambda calculus and type (or domain) equations in [Bar84,Sco80].) Given this, and the fact that equalizers allow us to type terms with respect to equational hypotheses, it is easy to show that equalizers give us terms without normal form. For example, if Γ is a context containing the typing assumption $x{:}\{\, y{:}\mathrm{triv} \mid \tau = \tau \rightarrow \tau : U_1 \,\}$, for any U_1 type τ, then by the typing rules in Table 7, we may conclude that $\Gamma \rhd \tau = \tau \rightarrow \tau : U_1$. Therefore, using the type equality rule from Table 4, we may give any term with type τ type $\tau \rightarrow \tau$, and vice versa. This allows is 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 $x{:}\{\, y{:}\mathrm{triv} \mid \tau = \tau \rightarrow \tau : U_1 \,\}$ and nonnormalizing body.

10 Towards Representation Independence and Full Abstraction

Two important issues in the study of programming languages are representation independence [Rey83,MM85,Mit86a] and full abstraction [Mil77,Plo77], neither of which seems to have been successfully applied to the study of full ML. Roughly speaking, representation independence is the property that the behavior of well-typed programs is independent of low-level decisions about the implementation of basic data types. For example, in a typed language with booleans, it should not matter whether *true* is represented by 1, and *false* by 0, or vice versa, as long as all the operations on booleans are implemented appropriately. Full abstraction is related to the observable equivalence of program expressions. We say expressions M and N are observationally equivalent if, for any program context $\mathcal{C}[\,]$ (*i.e.*, expression with a "hole" such that $\mathcal{C}[M]$ and $\mathcal{C}[N]$ are closed terms of printable type), the result of evaluating $\mathcal{C}[M]$ is the same as evaluating $\mathcal{C}[N]$. A fully abstract interpretation for a programming language is a semantic model with the property that observationally equivalent expressions are equal.

Both representation independence and full abstraction seem difficult to study within the implicitly–typed framework of Milner's original paper and subsequent similar studies. Proving representation independence for ML involves showing that if we change the representation for some type like the booleans, the output of any full program remains unchanged. But because implicitly–typed interpretations involve meanings for all untyped terms, it is technically difficult to study "arbitrary" representations for booleans. Using the explicitly–typed framework of this paper, it seems straightforward to use the techniques of [MM85,Mit86a] to prove representation independence for ML. In fact, since the types of our function calculus are stratified into levels, representation independence for ML seems much simpler than for the Girard–Reynolds calculus, and it seems likely that the results of [Mit86a] on substitution equivalence for abstract data type implementations could be strengthened considerably. We consider this an important direction for future research.

Although the development of fully–abstract models is not an easy task (*c.f.*[Mul84]), our explicitly–typed, stratified calculus seems the simplest adequate framework. To begin with, it is not entirely clear how to define observational equivalence in an implicitly–typed setting. One definition is that two terms M and N of the same type are observationally equivalent if, for every program context $\mathcal{C}[\,]$ such that both $\mathcal{C}[M]$ and $\mathcal{C}[N]$ are well–typed, we have $\mathcal{C}[M] = \mathcal{C}[N]$. An alternative would be to say that a context with $\mathcal{C}[M]$ well–typed and $\mathcal{C}[N]$ not would distinguish M from N. Neither definition seems very appealing, since typed terms are compared according to their untyped behavior, but perhaps there is some merit to one of these.

A more promising approach to full abstraction is to associate a different equivalence relation with each type (see, e.g., [CZ86,FS87,Mit86b]), so that two terms could be considered equivalent without having equal untyped meanings. This allows us to incorporate the fact that, for example, M and N may be equal when used as boolean functions, but unequal as integer functions. However, as shown in [Mit86b] for the Girard–Reynolds calculus, models based on partial equivalence relations are a special case of explicitly–typed models. Therefore, there is no loss of generality in considering an explicitly-typed syntax and semantics. For this reason, we expect the explicitly–typed analysis of ML given here will be useful in any future study of full abstraction.

11 Conclusion and Directions for Further Investigation

We have given an precise description of the type system for much of Standard ML, using a function calculus called XML. Our analysis is based on the belief that ML is most 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 studying representation independence or full abstraction, as outlined in Section 10. The distinction between U_1 and U_2 in XML reflects the typing rules of Standard ML, and leads to a number of significant technical simplifications in the study of XML. Moreover, universe distinctions seem essential to the character of ML, as discussed in Section 8.

Some important aspects of Standard ML have been omitted. For the language we have considered, actual implementations compare type expressions using the concept of "generativity," which is discussed in [HMT87b,HMT87a]. In addition, signature matching allows structures to match signatures that specify fewer components than the structures actually provide. We believe more accurate descriptions of both of these characteristics of Standard ML could be given within the framework of this paper, and that it would be worthwhile to do so. It would also be useful to extend our account of ML to a fuller language with exceptions, recursive types and/or references. In particular, we hope that an explicitly-typed study of polymorphic references would clarify the connection between polymorphism and side effects, a continuing trouble spot in the ML type inference algorithm.

Along with representation independence and fully abstract models, an important topic for further study is the status of data abstraction in Standard ML. It is implicit in the discussion of [Mac85] that **abstype** declarations, as described in [GMW79,MP85] may be replaced by "in-line" applications of functor expressions. This seems quite plausible, but it is important to clarify the precise sense in which functor application is equivalent to **abstype**. Based on past experience, *e.g.*[MM85,Mit86a], a theory of logical relations seems to be the appropriate method of investigation.

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 Standard ML relies on careful consideration of the order in which re-write 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 useful to develop a typed calculus that is faithful to the operational semantics, following the pattern established by Plotkin's λ_v–calculus [Plo75] and Martin-Löf's type theory [Mar82].

Acknowledgements:

Thanks to Dave MacQueen for many insightful discussions of ML over the last few years, and comments on this paper in particular. Thanks also to Andrzej Tarlecki for comments on an earlier draft.

References

[ABL86] R. Amadio, K. Bruce, and G. Longo. The finitary projection model for second order lambda calculus and solutions to higher order domain equations. In *IEEE Symp. Logic in Computer Science*, pages 122–130, 1986.

[Bar84] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North Holland, 1984. (revised edition).

[BM84] K. Bruce and A. Meyer. A completeness theorem for second-order polymorphic lambda calculus. In *Proc. Int. Symp. on Semantics of Data Types, Sophia-Antipolis (France), Springer LNCS 173*, pages 131–144., 1984.

[BMM98] K. B. Bruce, A. R. Meyer, and J. C. Mitchell. The semantics of second-order lambda calculus. *Information and Computation*, 198+. (to appear).

[C*86] Constable et al. *Implementing Mathematics with the NuprlProof Development System.* Volume 37 of *Graduate Texts in Mathematics*, Prentice-Hall, 1986.

[Car] R. Cartwright. ?? In *??*, page ??, ?? ??

[Car84] Luca Cardelli. The semantics of multiple inheritance. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, pages 51–67, Springer-Verlag, 1984.

[CH84] T. Coquand and G. Huet. A theory of constructions. In *Proc. Int. Symp. on Semantics of Data Types, Sophia-Antipolis (France), Springer LNCS 173*, June 1984. Paper does not appear in proceedings.

[Coq86] T. Coquand. An analysis of girard's paradox. In *Proc. IEEE Symp. on Logic in Computer Science*, pages 227–236, June 1986.

[CW86] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 18(4), December 1986.

[CZ86] M. Coppo and M. Zacchi. Type inference and logical relations. In *Proc. IEEE Symp. on Logic in Computer Science*, pages 218–226, June 1986.

[DB80] N.G. De Bruijn. A survey of the project automath. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 579–607, Academic Press, 1980.

[DM82] L. Damas and R. Milner. Principal type schemes for functional programs. In *9-th ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[FS87] P. Freyd and A. Scedrov. Some semantic aspects of polymorphic lambda calculus. In *IEEE Symp. Logic in Computer Science*, pages 315–319, June 1987.

[Gir71] J.-Y. Girard. Une extension de l'interpretation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In J.E. Fenstad, editor, *2nd Scandinavian Logic Symposium*, pages 63–92, North-Holland, 1971.

[Gir72] J.Y. Girard. Interpretation fonctionelle et elimination des coupures de l'arithmetique d'ordre superieur. These D'Etat, Universite Paris VII, 1972.

[GMW79] M.J. Gordon, R. Milner, and C.P. Wadsworth. *Edinburgh LCF.* Volume 78 of *Lecture Notes in Computer Science*, Springer-Verlag, 1979.

[Gor79] M.J.C. Gordon. *The Denotational Description of Programming Languages.* Springer-Verlag, 1979.

[HH86] J. Hook and D. Howe. *Impredicative strong existential equivalent to type:type.* Technical Report TR 86-760, Cornell University, 1986.

[HMM86] Robert Harper, David MacQueen, and Robin Milner. *Standard ML.* Technical Report ECS–LFCS–86–2, Laboratory for the Foundations of Computer Science, Edinburgh University, March 1986.

[HMT87a] Robert Harper, Robin Milner, and Mads Tofte. *The Semantics of Standard ML.* Technical Report ECS–LFCS–87–36, Laboratory for the Foundations of Computer Science, Edinburgh University, August 1987.

[HMT87b] Robert Harper, Robin Milner, and Mads Tofte. A type discipline for program modules. In *TAPSOFT '87*, Springer-Verlag, March 1987.

[How80] W. Howard. The formulas-as-types notion of construction. In *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 479–490, Academic Press, 1980.

[How87] D.J. Howe. The computational behavior of girard's paradox. In *IEEE Symp. Logic in Computer Science*, pages 205–214, June 1987.

[Lei83] D. Leivant. Polymorphic type inference. In *Proc. 10-th ACM Symp. on Principles of Programming Languages*, pages 88–98, 1983.

[LS86] J. Lambek and P.J. Scott. *Introduction to Higher-Order Categorical Logic.* Cambridge studies in advanced mathematics 7, 1986.

[Mac85] D.B. MacQueen. Modules for standard ml. *Polymorphism*, 2(2), 1985. 35 pages. An earlier version appeared in Proc. 1984 ACM Symp. on Lisp and Functional Programming.

[Mac86] D.B. MacQueen. Using dependent types to express modular structure. In *Proc. 13-th ACM Symp. on Principles of Programming Languages*, 1986. To appear.

[Mar73] Per Martin-Löf. An intuitionistic theory of types: predicative part. In H. E. Rose and J. C. Shepherdson, editors, *Logic Colloquium, '73*, pages 73–118, North-Holland, Amsterdam, 1973.

[Mar82] Per Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, North-Holland, Amsterdam, 1982.

[McC79] N. McCracken. *An Investigation of a Programming Language with a Polymorphic Type Structure.* PhD thesis, Syracuse Univ., 1979.

[Mil77] R. Milner. Fully abstract models of typed lambda calculi. *Theoretical Computer Science*, 4(1), 1977.

[Mil78] R. Milner. A theory of type polymorphism in programming. *JCSS*, 17, 1978. pages 348-375.

[Mil85] R. Milner. The standard ml core language. *Polymorphism*, 2(2), 1985. 28 pages. An earlier version appeared in Proc. 1984 ACM Symp. on Lisp and Functional Programming.

[Mit86a] J.C. Mitchell. Representation independence and data abstraction. In *Proc. 13-th ACM Symp. on Principles of Programming Languages*, pages 263–276, January 1986.

[Mit86b] J.C. Mitchell. A type-inference approach to reduction properties and semantics of polymorphic expressions. In *ACM Conference on LISP and Functional Programming*, pages 308–319, August 1986.

[Mit87] J.C. Mitchell. Polymorphic type inference and containment. *Information and Computation*, this issue, 1987.

[ML71] S. Mac Lane. *Categories for the Working Mathematician.* Volume 5 of *Graduate Texts in Mathematics*, Springer-Verlag, 1971.

[MM85] J.C. Mitchell and A.R. Meyer. Second-order logical relations. In *Logics of Programs*, pages 225–236, Springer-Verlag LNCS 193, June 1985.

[MP85] J.C. Mitchell and G.D. Plotkin. Abstract types have existential types. In *Proc. 12-th ACM Symp. on Principles of Programming Languages*, pages 37–51, January 1985. Revised and expanded version to appear in ACM TOPLAS.

[MPS86] D. MacQueen, G Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71(1/2):95–130, 1986.

[MR86] A.R. Meyer and M.B. Reinhold. Type is not a type. In *Proc. 13-th ACM Symp. on Principles of Programming Languages*, pages 287–295, January 1986.

[Mul84] K. Mulmuley. A semantic characterization of full abstraction for typed lambda calculus. In *Proc. 25-th IEEE Symp. on Foundations of Computer Science*, pages 279–288, 1984.

[Plo75] Gordon Plotkin. Call-by-name, call-by-value, and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[Plo77] G.D. Plotkin. Lcf considered as a programming language. *Theoretical Computer Science*, 13, 1977.

[Rey74] J.C. Reynolds. Towards a theory of type structure. In *Paris Colloq. on Programming*, pages 408–425, Springer-Verlag LNCS 19, 1974.

[Rey81] J.C. Reynolds. The essence of algol. In de Bakker and van Vliet, editors, *Algorithmic Languages*, pages 345–372, IFIP, North Holland, 1981.

[Rey83] J.C. Reynolds. Types, abstraction, and parametric polymorphism. In *Information Processing '83*, pages 513–523, North-Holland, Amsterdam, 1983.

[Rey84] J.C. Reynolds. Polymorphism is not set-theoretic. In *Proc. Int. Symp. on Semantics of Data Types, Sophia-Antipolis (France), Springer LNCS 173*, pages 145–156, Springer-Verlag, 1984.

[Sco80] D.S. Scott. Relating theories of the lambda calculus. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 403–450, Academic Press, 1980.

[Sol78] M. Solomon. Tyoe definitions with parameters. In *Proc. 5-th ACM Symp. on Principles of Programming Languages*, pages 31–38, 1978.

[Sta85] R. Statman. Logical relations and the typed lambda calculus. *Information and Control*, 65:85–97, 1985.

[Tro73] A.S. Troelstra. *Mathematical Investigation of Intuitionistic Arithmetic and Analysis.* Volume 344 of *Lecture Notes in Mathematics*, Springer-Verlag, 1973.

[Wan84] M. Wand. A types-as-sets semantics for milner-style polymorphism. In *Proc. 11-th ACM Symp. on Principles of Programming Languages*, pages 158–164, January 1984.

$$\text{VAR} \qquad \Gamma \rhd x : \sigma \quad (\Gamma(x) = \sigma)$$

$$\text{GEN} \qquad \frac{\Gamma \rhd e : \sigma}{\Gamma \rhd e : \forall t.\sigma} \quad (\textit{t not free in } \Gamma)$$

$$\text{SPEC} \qquad \frac{\Gamma \rhd e : \sigma}{\Gamma \rhd e : \sigma'} \quad (\sigma \sqsubseteq \sigma')$$

$$\text{ABS} \qquad \frac{\Gamma, x{:}\tau \rhd e' : \tau'}{\Gamma \rhd \lambda x.e' : \tau \to \tau'} \quad (x \notin \mathrm{FV}(\Gamma))$$

$$\text{APP} \qquad \frac{\Gamma \rhd e : \tau' \to \tau \quad \Gamma \rhd e' : \tau'}{\Gamma \rhd ee' : \tau}$$

$$\text{LET} \qquad \frac{\Gamma \rhd e : \sigma \quad \Gamma, x{:}\sigma \rhd e' : \tau'}{\Gamma \rhd \mathbf{let}\ x = e\ \mathbf{in}\ e' : \tau'} \quad (x \notin \mathrm{FV}(\Gamma))$$

Table 1: Damas–Milner Type Assignment

$$\text{VAR} \qquad \Gamma \rhd x : \sigma \quad (\Gamma(x) = \sigma)$$

$$\text{TABS} \qquad \frac{\Gamma \rhd M : \sigma}{\Gamma \rhd \lambda t.M : \Pi t{:}U_1.\sigma} \quad (\textit{t not free in } \Gamma)$$

$$\text{TAPP} \qquad \frac{\Gamma \rhd M : \Pi t{:}U_1.\sigma}{\Gamma \rhd M\tau : [\tau/t]\sigma}$$

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

$$\text{APP} \qquad \frac{\Gamma \rhd M : \tau' \to \tau \quad \Gamma \rhd M' : \tau'}{\Gamma \rhd MM' : \tau}$$

$$\text{LET} \qquad \frac{\Gamma \rhd M : \sigma \quad \Gamma, x{:}\sigma \rhd M' : \tau'}{\Gamma \rhd \mathbf{let}\ x{:}\sigma = M\ \mathbf{in}\ M' : \tau'} \quad (x \notin \mathrm{FV}(\Gamma))$$

Table 2: *Core*-XML Type System

$$\frac{}{()\ \text{context}} \qquad \frac{\Gamma \rhd \tau : U_i}{\Gamma, x{:}\tau\ \text{context}} \quad (i = 1, 2;\ x \notin \mathit{Dom}(\Gamma))$$

$$\frac{\Gamma, x{:}\tau\ \text{context}}{\Gamma, x{:}\tau \rhd x : \tau} \qquad \frac{\Gamma \rhd M : \tau \quad \Gamma'\ \text{context}}{\Gamma' \rhd M : \tau} \quad (\forall x \in \mathit{Dom}(\Gamma).\Gamma(x) = \Gamma'(x))$$

Table 3: Context and structural rules for XML

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

$$\frac{\Gamma \rhd M : \sigma \quad \Gamma \rhd \sigma = \tau : U_i}{\Gamma \rhd M : \tau} \qquad (i = 1, 2)$$

Table 4: Universes and type equality

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

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

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

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

Table 5: Types and terms in U_1

$$\frac{\Gamma \triangleright \sigma : U_2 \quad \Gamma, x{:}\sigma \triangleright \tau : U_2}{\Gamma \triangleright \Pi x{:}\sigma.\tau : U_2} \qquad (x \notin Dom(\Gamma))$$

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

$$\frac{\Gamma \triangleright M : \Pi x{:}\sigma.\tau \quad \Gamma \triangleright N : \sigma}{\Gamma \triangleright MN : [N/x]\tau}$$

$$\frac{\Gamma \triangleright \sigma : U_2 \quad \Gamma, x{:}\sigma \triangleright \tau : U_2}{\Gamma \triangleright \Sigma x{:}\sigma.\tau : U_2} \qquad (x \notin Dom(\Gamma))$$

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

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

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

Table 6: Types and terms in U_2

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

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

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

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

Table 7: Equalizer types for sharing constraints