Higher-Order Modules and the Phase Distinction*

Robert Harper
Carnegie Mellon University
Pittsburgh, PA 15213

John C. Mitchell
Stanford University
Stanford, CA 94305

Eugenio Moggi
University of Edinburgh
Edinburgh EH9-3JZ, U.K.

Abstract

Typed λ-calculus is an important tool in programming language research because it provides an extensible framework for studying language features both in isolation and in their relation to each other. In earlier work we introduced a predicative function calculus, XML, for modeling several aspects of the Standard ML type system. Following MacQueen, our study focused on the use of dependent types to represent the modularity constructs of Standard ML. In addition to shedding some light on the trade-offs between language features, our analysis suggested that the first-order modules system of ML could be naturally extended to higher orders. However, whereas ML maintains a clear distinction between compile-time and run-time in both its implementation and formal semantics, the XML calculus blurs this distinction. Since static type checking is, in our view, essential to the practical utility of ML, we introduce a refinement of the XML calculus for which type checking is decidable at compile time. This calculus is based on a refinement of our earlier treatment of universes, and employs a non-standard equational theory of modules and signatures inspired by a category-theoretic account of the phase distinction.

1 Introduction

The module system of Standard ML [HMM86] is an elegant system for encapsulating and combining program units. The basic entities of the module system are *structures*, which may be thought of as reified environments, *functors*, functions on structures, and *signatures*, which describe the components of structures and their types. Program units are represented as structures that are linked together by the use of functors. The coherence of a combination of program units is ensured by the use of "sharing" specifications.

There are two existing analyses of the module system, each elucidating several important features. In [HMT87b, HMT87a, Tof87] a formal analysis of the Standard ML type system is given, using structured operational semantics to describe the computational behavior of the type checker. While it gives a precise, implementation-independent characterization of the ML type system, this approach is not especially helpful in understanding the logical structure of the ML type system, nor its denotational semantics. In particular, the operational approach does not easily accommodate extensions to the language; each extension must be treated on a case-by-case basis.

The second form of analysis, beginning with [Mac86] and continued in [MH88], gives a type-theoretic account of the language using dependent sum types $\Sigma x{:}A.B$ to explain structures and dependent function types $\Pi x{:}A.B$ to account for functors. In addition to providing some insight into the functional behavior of the module constructs, the XML λ-calculus introduced in [MH88] establishes a framework for studying a number of related *ML-like* languages, and provides the basis for model-theoretic studies of these languages. (The title and approach taken there were inspired

*Reference should be made to the paper, with the same authors and title, published in the proceedings of the 17th POPL ACM Conference. This report is an extended version of the conference paper.

by [Rey81]). For instance unbounded recursion in ML can be handled in XML by considering a simple *theory* introducing a constant *fix* of type $\Pi s{:}U_1.\Pi t{:}U_1.(s\rightarrow t)\rightarrow(s\rightarrow t)$ and the equational axiom

$$s{:}U_1, t{:}U_1, f{:}(s\rightarrow t)\rightarrow(s\rightarrow t) \gg \mathit{fix}(f) = f(\mathit{fix}(f)) : (s\rightarrow t)\rightarrow(s\rightarrow t)$$

expressing the desired fixed-point property. Because variants of the language may be considered as XML theories, the emphasis of the approach is on properties of the language that remain invariant under such extensions.

However, one very important feature of Standard ML has not been accounted for in the type-theoretic analysis is the compile-time nature of type checking. The problem is that the type structure of XML is sufficiently rich as to admit variations in the language that violate the so-called *phase distinction* (introduced by Cardelli [Car88]) between *compile-time* and *run-time* expressions. For example, by introducing a constant A of type $U_1\times \mathit{int}\rightarrow U_1$ (which we may think of as "*Array*"), we obtain types of the form $A(\mathit{int}, e)$ and $A(\mathit{int}, e')$, where e and e' are expressions of type *int*. Since type checking in XML involves testing of type equivalence, the presence of such "mixed-phase" type expressions entail that the type checker must test equivalence of run-time values. The definition of equivalence of run-time values is sensitive to the particular theory in which we are working, and hence there cannot be a single type-checking algorithm for all possible variations in the language. Moreover, in typical cases equality of run-time expressions is undecidable, and so some incomplete method would be needed in any case. Since the "compile-time type checking" property is an essential aspect of ML, we are led to refine our earlier analysis of the language to take account of the phase distinction.

It might be thought at first glance that this is largely a matter of refining the type structure of XML so as to rule out examples such as that given above. After all, no such declaration is syntactically possible in ML, and hence it seems that XML is unwarrantedly liberal in admitting such declarations. While this is indeed the case, it is not the whole story. One benefit of the type-theoretic analysis is that it suggest a natural extension of the ML module system to higher-orders. In ML functors are first-order in the sense that they take structures as arguments and return structures as results; there is no provision for passing functors as arguments, nor returning them as results. For many applications, particularly in their role as the means of combining program units (*i.e.*, "linking"), first-order functors are adequate for practical purposes. However, recent work suggests that higher-order forms may be useful for supporting separate compilation (roughly, a separately-compilable module is abstracted with respect to the functors used within it.) Furthermore, higher-order functors may be used as an alternative to ML's "sharing" specifications to ensure that program units are coherently combined [BL84, Mac86].

Unfortunately, the extension of the module facility to higher orders results in violations of the phase distinction. Specifically, since structures are "mixed-phase" entities, consisting of a compile-time part and a run-time part, the usual functional interpretation of functors introduces, at higher types, phase-violating constructs similar to the example given above. Roughly speaking, A can be thought of as a "functor variable" resulting in a structure consisting of a type and some trivial run-time value. By projecting out the type component, we reproduce the example above in the higher-order modules calculus. Thus the straightforward extension of ML's functors to higher-orders is incompatible with compile-time type checking.

The main objective of this paper is to account for the phase distinction in a type theory supporting higher-order modules. We present a calculus, λ^{ML}, that is at once a refinement and an extension of the XML calculus considered in our earlier work. The universe structure of XML is refined so that the core language (*i.e.*, the language without modules) possesses a natural phase distinction. Then the language is extended in a systematic way to include dependent types for representing structures and functors. In order to preserve the phase distinction a non-standard formulation of the rules for dependent types is needed. The crucial point is that we consider a non-standard equational theory inspired by the categorical viewpoint considered in [Mog89a]. As with the XML calculus, λ^{ML} is parameterized by a theory consisting of a set of type constructors a set of term constructors, and equational axioms governing the term constructors. (Actually, nothing hinges on the equational axioms; our approach is compatible with the computational λ-calculus approach taken in [Mog89b].)

$$
\begin{array}{llll}
k & \in kind & ::= & 1 \mid T \mid k_1 \times k_2 \mid k_1 \rightarrow k_2 \\
u & \in constr & ::= & v \mid 1 \mid \times \mid \rightarrow \mid * \mid \langle u_1, u_2 \rangle \mid \pi_i(u) \mid (\lambda v{:}k.u) \mid u_1\, u_2 \\
\sigma & \in type & ::= & \mathrm{set}(u) \mid \sigma_1 \times \sigma_2 \mid \sigma_1 \rightarrow \sigma_2 \mid (\forall v{:}k.\sigma) \\
e & \in term & ::= & x \mid * \mid \langle e_1, e_2 \rangle \mid \pi_i(e) \mid (\lambda x{:}\sigma.e) \mid e_1\, e_2 \mid (\Lambda v{:}k.e) \mid e[u] \\
\Phi & \in context & ::= & \emptyset \mid \Phi, v{:}k \mid \Phi, x{:}\sigma
\end{array}
$$

Table 1: λ^{ML} raw expressions

The analysis of λ^{ML} focuses on two key properties of the language: decidability, and compile-time type checking. The formal definition of "compile-time type checking" focuses on the equational principles needed for type checking. The general idea is that we prove that only a limited form of equational reasoning need be performed by the type checker, and that this reasoning is independent of the computational properties of run-time expressions. Together with decidability, the compile-time type checking property ensures that the type checker is insensitive to variations in the language, and, for example, will terminate even in the presence of an undecidable theory of run-time values.

This paper is organized as follows. In Section 2 we introduce the core calculus, λ^{ML}, that will serve as the basis for our work. λ^{ML} is essentially the HML calculus given in [Mog89a] and closely related to the *Core*-XML calculus given in [MH88]. In Section 3 we introduce λ^{ML}_{mod}, the full calculus of higher-order modules. This calculus may be seen as a formal presentation of the category of modules determined by λ^{ML} (see [Mog89a] for details). The modularity constructs of ML (with the exception of sharing specifications) are represented in λ^{ML}_{mod} by a non-standard formalization of dependent types based on taking a number of non-standard type equivalences. λ^{ML}_{mod} is proved to be a definitional extension of an auxiliary "structures-only" calculus, and this result is then used to establish decidability and compile-time type checking for the full calculus of modules. Finally, in Section 4 we suggest directions for further research.

2 Core Calculus

We begin by giving the definition of the λ^{ML} core calculus, λ^{ML}, which is essentially the calculus HML of [Mog89a]. This calculus captures many of the essential features of the ML type system, but omits, for the sake of simplicity, ML's concrete and abstract types (which could be modeled using existential types [MP88]), recursive types (which can be described through a λ^{ML} theory), and record types. We also omit consideration of pattern matching, and computational aspects such as side-effects and exceptions (for which see [Mog89b]).

2.1 Syntactic Preliminaries

There are four basic syntactic classes in λ^{ML}: the *kinds*, which classify the *constructors*, and the *types*, which classify the *terms*. The constructors include *monotypes* such as *int*, and type constructors such as *list*. The distinguished kind T classifies the monotypes. The terms of the calculus correspond to the basic expression forms of ML, but with an explicitly-typed syntax. (See [MH88] for a discussion of this choice.) It is important to note that our "types" correspond roughly to ML's "type schemes," the essential differences being that we admit quantification over all kinds, not just the kind of monotypes, and we require that the collection of types satisfy closure conditions not found in ML. (The additional closure properties for type schemas ensure that the category of modules for λ^{ML} is relatively cartesian closed.) This organization is a refinement of the type structure of *Core*-XML. The distinguished kind T is essentially the first universe U_1 of *Core*-XML. However, the second universe, U_2, of *Core*-XML is here refined into the collections of kinds and types. The cumulativity of the *Core*-XML universes is here replaced by the explicit "injection" of T into the collection of types.

Φ *context*	Φ is a context
$\Phi \gg u : k$	u is a constructor of kind k
$\Phi \gg \sigma$ *type*	σ is a type
$\Phi \gg e : \sigma$	e is a term of type σ
$\Phi \gg u_1 = u_2\ k$	u_1 and u_2 are equal constructors of kind k
$\Phi \gg \sigma_1 = \sigma_2$ *type*	σ_1 and σ_2 are equal types
$\Phi \gg e_1 = e_2 : \sigma$	e_1 and e_2 are equal terms of type schema σ

Table 2: λ^{ML} judgement forms

2.2 Syntax

The syntax of λ^{ML} raw expressions is given in Table 1. The collection of term variables, ranged over by x, and the collection of constructor variables, ranged over by v, are assumed to be disjoint. The metavariable τ ranges over the collection of monotypes (constructors of kind T). Contexts consist of a sequence of *declarations* of the form $v{:}k$ and $x{:}\sigma$ declaring the kind or type, respectively, of a constructor or term variable. In addition to the context-free syntax, we require that no variable be declared more than once in a context Φ so that we may unambiguously regard Φ as a partial function with finite domain $\mathrm{Dom}(\Phi)$ assigning kinds to constructor variables and types to term variables.

2.3 Judgement Forms

There are two classes of judgements in λ^{ML}, the *formation judgements* and the *equality judgements.* The formation judgements are used to define the set of well-formed λ^{ML} expressions. With the exception of the kind expressions, there is one formation judgement for each syntactic category. (Every raw kind expression is well-formed.) The equality judgements are used to axiomatize equivalence of expressions. (There is no equality judgement for kinds; kind equivalence is just syntactic identity.) The equality judgements are divided into two classes, the *compile-time* equations and the *run-time* equations, reflecting the intuitive phase distinction: kind and type equivalence are compile-time, term equivalence is run-time. The judgment forms of λ^{ML} are summarized in Table 2. The metavariable $\mathcal{F}$ ranges over formation judgements, $\mathcal{E}$ ranges over equality judgements, and $\mathcal{J}$ ranges over all forms of judgement. We sometimes write $\Phi \gg \alpha$ to stand for an arbitrary judgement when we wish to make the context part explicit.

2.4 Formation Rules

The syntax of λ^{ML} is specified by a set of inference rules for deriving formation judgements given in Appendix A. We write $\lambda^{ML} \vdash \mathcal{F}$ to indicate that the formation judgement $\mathcal{F}$ is derivable using these rules. The formation rules may be summarized as follows. The constructors and kinds form a simply-typed λ-calculus (with product and unit types) with base kind T, and basic constructors 1, $\times$, and $\rightarrow$. The collection of types is built from base types 1 and $\mathrm{set}(\tau)$, where τ is a constructor of kind T, using the type constructors $\times$ and $\rightarrow$, and quantification over an arbitrary kind. The terms amount to an explicitly-typed presentation of the ML core language, similar to that presented in [MH88]. (The **let** construct is omitted since it is definable here.)

2.5 Equality rules

The rules for deriving equational judgements are given in Appendix A. We write $\lambda^{ML} \vdash \mathcal{E}$ to indicate that an equation $\mathcal{E}$ is derivable in accordance with these rules. The λ^{ML} equational rules are formulated so as to ensure that if an equational judgement is derivable, then it is well-formed, meaning that the evident associated formation judgements are derivable. For the sake of convenience we give a brief summary of the equational rules of λ^{ML}.

2.5.1 Compile-Time Equality

Constructors Equivalence of constructor expressions is the standard equivalence of terms in the simply-typed λ-calculus based on the following axioms:

$$(1\ \eta) \qquad \frac{\Phi \gg u : 1}{\Phi \gg u = * : 1}$$

$$(\times\ \beta) \qquad \frac{\Phi \gg u_1 : k_1 \quad \Phi \gg u_2 : k_2}{\Phi \gg \pi_i(\langle u_1, u_2 \rangle) = u_i : k_i}\ (i = 1, 2)$$

$$(\times\ \eta) \qquad \frac{\Phi \gg u : k_1 \times k_2}{\Phi \gg \langle \pi_1(u), \pi_2(u) \rangle = u : k_1 \times k_2}$$

$$(\rightarrow \beta) \qquad \frac{\Phi \gg u_1 : k_1 \quad \Phi, v{:}k_1 \gg u_2 : k_2}{\Phi \gg (\lambda v{:}k_1.u_2)\, u_1 = [u_1/v]u_2 : k_2}$$

$$(\rightarrow \eta) \qquad \frac{\Phi \gg u : k_1 \rightarrow k_2}{\Phi \gg (\lambda v{:}k_1.u\, v) = u : k_1 \rightarrow k_2}\ (v \notin \mathrm{Dom}(\Phi))$$

Types The equivalence relation on types includes the following axioms expressing the interpretation of the basic ML type constructors

$$(1\ T =) \qquad \frac{\Phi\ \mathit{context}}{\Phi \gg \mathrm{set}(1) = 1\ \mathit{type}}$$

$$(\times\ T =) \qquad \frac{\Phi \gg \tau_1 : T \quad \Phi \gg \tau_2 : T}{\Phi \gg \mathrm{set}(\tau_1 \times \tau_2) = \mathrm{set}(\tau_1) \times \mathrm{set}(\tau_2)\ \mathit{type}}$$

$$(\rightarrow T =) \qquad \frac{\Phi \gg \tau_1 : T \quad \Phi \gg \tau_2 : T}{\Phi \gg \mathrm{set}(\tau_1 \rightarrow \tau_2) = \mathrm{set}(\tau_1) \rightarrow \mathrm{set}(\tau_2)\ \mathit{type}}$$

2.5.2 Run-Time Equality

Terms There are seven axioms corresponding to the reduction rules associated with each of the type constructors:

$$(1\ \eta) \qquad \frac{\Phi \gg e : 1}{\Phi \gg e = * : 1}$$

$$(\times\ \beta) \qquad \frac{\Phi \gg e_1 : \sigma_1 \quad \Phi \gg e_2 : \sigma_2}{\Phi \gg \pi_i(\langle e_1, e_2 \rangle) = e_i : \sigma_i}\ (i = 1, 2)$$

$$(\times\ \eta) \qquad \frac{\Phi \gg e : \sigma_1 \times \sigma_2}{\Phi \gg \langle \pi_1(e), \pi_2(e) \rangle = e : \sigma_1 \times \sigma_2}$$

$$(\rightarrow \beta) \qquad \frac{\Phi \gg e_1 : \sigma_1 \quad \Phi, x{:}\sigma_1 \gg e_2 : \sigma_2}{\Phi \gg (\lambda x{:}\sigma_1.e_2)\, e_1 = [e_1/x]e_2 : \sigma_2}$$

$$(\rightarrow \eta) \qquad \frac{\Phi \gg e : \sigma_1 \rightarrow \sigma_2}{\Phi \gg (\lambda x{:}\sigma_1.e\, x) = e : \sigma_1 \rightarrow \sigma_2}\ (x \notin \mathrm{Dom}(\Phi))$$

$$(\forall\ \beta) \qquad \frac{\Phi \gg u : k \quad \Phi, v{:}k \gg e : \sigma}{\Phi \gg (\Lambda v{:}k.e)[u] = [u/v]e : [u/v]\sigma}$$

$$(\forall\ \eta) \qquad \frac{\Phi \gg e : (\forall v{:}k.\sigma)}{\Phi \gg (\Lambda v{:}k.e[v]) = e : (\forall v{:}k.\sigma)}\ (v \notin \mathrm{Dom}(\Phi))$$

2.6 Theories

The λ^{ML} calculus is parametric in a *theory* $\mathcal{T} = (\Phi^{\mathcal{T}}, \mathcal{A}^{\mathcal{T}})$ consisting of a well-formed context $\Phi^{\mathcal{T}}$, and a set $\mathcal{A}^{\mathcal{T}}$ of run-time equational axioms of the form $e_1 = e_2 : \sigma$ such that $\Phi_0 \gg e_i : \sigma$ is derivable for $i = 1, 2$. A theory corresponds to the programming language notion of standard prelude, and might contain declarations such as $int : T$ and $fix : \forall t{:}T.\,\mathrm{set}((t \rightarrow t) \rightarrow t)$, and axioms such as

$$\Lambda t{:}T.\lambda f{:}\,\mathrm{set}(t \rightarrow t).fix(f) = \Lambda t{:}T.\lambda f{:}\,\mathrm{set}(t \rightarrow t).f(fix(f)) : \forall t{:}T.(t \rightarrow t) \rightarrow t.$$

expressing the fixed-point property of *fix*. For $\mathcal{T} = (\Phi^{\mathcal{T}}, \mathcal{A}^{\mathcal{T}})$, we write $\lambda^{ML}[\mathcal{T}] \vdash \mathcal{J}$ to indicate that the judgement $\mathcal{J}$ is derivable in λ^{ML}, taking the variables declared in $\Phi^{\mathcal{T}}$ as basic constructors and terms, and taking the equations in $\mathcal{E}^{\mathcal{T}}$ as non-logical axioms. We write $\lambda^{ML}[\mathcal{T}] \vdash_{ct} \mathcal{J}$ to indicate that the judgement $\mathcal{J}$ is derivable from theory $\mathcal{T}$ using only the compile-time equational rules (and the equational axioms of $\mathcal{T}$.)

2.7 Properties of λ^{ML}

For Φ a λ^{ML} context, let Φ^{c} denote the context obtained by omitting all term variable declarations from Φ, and, conversely, let Φ^{r} denote the context obtained by eliminating all constructor variable declarations from Φ. The following lemma expresses the compile-time type checking property of λ^{ML}:

Lemma 2.1 *Let $\mathcal{T}$ be any theory. The following implications hold:*

If $\lambda^{ML}[\mathcal{T}] \vdash$	*then* $\lambda^{ML}[\Phi^{c}alT, \emptyset] \vdash_{ct}$
Φ context	$\Phi^{\mathsf{c}}, \Phi^{\mathsf{r}}$ context
$\Phi \gg u : k$	$\Phi^{\mathsf{c}} \gg u : k$
$\Phi \gg u_1 = u_2 : k$	$\Phi^{\mathsf{c}} \gg u_1 = u_2 : k$
$\Phi \gg \sigma$ type	$\Phi^{\mathsf{c}} \gg \sigma$ type
$\Phi \gg \sigma_1 = \sigma_2$ type	$\Phi^{\mathsf{c}} \gg \sigma_1 = \sigma_2$ type
$\Phi \gg e : \sigma$	$\Phi^{\mathsf{c}}, \Phi^{\mathsf{r}} \gg e : \sigma$
$\Phi \gg e_1 = e_2 : \sigma$	$\Phi^{\mathsf{c}}, \Phi^{\mathsf{r}} \gg e_i : \sigma$

Since the constructors and kinds form a simply-typed λ-calculus, it is a routine matter to show that equality of well-formed constructors (and, consequently, types) in λ^{ML} is decidable. It is then easy to show that type checking in λ^{ML} is decidable. (This is in fact a well-known property of the polymorphic lambda calculus F_ω (*c.f.* [Gir71, Gir72, Rey74, BMM89]), which may be seen as an impredicative extension of the λ^{ML} calculus.

Lemma 2.2 *There is a straightforward one-pass algorithm which decides, for an arbitrary well-formed theory $\mathcal{T}$ and formation judgement $\mathcal{F}$, whether or not $\lambda^{ML}[\mathcal{T}] \vdash \mathcal{F}$.*

The main technical accomplishment of this paper is to present a full calculus encompassing the module expressions of ML which has a compile-time decidable type checking problem.

k	$\in$ *kind*	$::=$	$\ldots$
u	$\in$ *constr*	$::=$	$\ldots \mid s^{\mathsf{c}}$
σ	$\in$ *type*	$::=$	$\ldots$
e	$\in$ *term*	$::=$	$\ldots \mid s^{\mathsf{r}}$
S	$\in$ *sig*	$::=$	$[v{:}k, \sigma]$
M	$\in$ *mod*	$::=$	$[u, e]$
Φ	$\in$ *context*	$::=$	$\ldots \mid \Phi, s{:}S$

Table 3: λ^{ML}_{str} raw expressions

3 Modules Calculus

3.1 Overview

We begin by recalling the the type-theoretic analysis of the ML module facility initiated in [Mac86] and further developed in [MH88] (see also [NPS88, C+86, Mar84] for related ideas). According to those studies, a structure is an element of a *strong sum* type such as $S = \Sigma t{:}T.\sigma(t)$, and hence is, in this case, a tuple $[\tau, e]$ consisting of a type and a term. Selection of components of a structure is accomplished using the projections *Fst* and *Snd*. Functors are modeled as elements of *dependent function* types such as $\Pi s{:}(\Sigma t{:}T.\sigma(t)).(\Sigma t{:}T.\sigma'(t))$. A functor is therefore represented as a λ-term mapping structures to structures. As discussed in the introduction, the straightforward formulation of dependent types runs afoul of the intuitive phase distinction, and compile-time type checking is lost. For example, if F is a functor variable of signature $S \rightarrow S$ (where S is as above), then $\mathit{Fst}(F\,[\mathit{int}, 3])$ is an irreducible type expression involving a run-time sub-expression.

In this section we develop a calculus λ^{ML}_{mod} of higher-order modules with a phase distinction based on the categorical analysis given in [Mog89a]. We begin with an auxiliary "structures-only" calculus that is primarily a technical device needed for the proofs. The full calculus of higher-order modules has a standard syntax for dependent strong sums and functions, but a non-standard equational theory inspired by the categorical interpretation of program modules [Mog89a]. The calculus also employs a single non-standard typing rule for structures needed for the proof of decidability. The non-standard aspects of the calculus are justified by a proof that it is a definitional extension of the "structures-only" calculus, which itself bears a straightforward relationship to the core calculus. This definitional extension result is then used to prove that λ^{ML}_{mod} is decidable and admits compile-time type checking.

3.2 The Calculus of Structures

The calculus of structures, λ^{ML}_{str}, is an extension of λ^{ML} obtained by the introduction of structures and signatures. We assume given a set of structure variables, disjoint from constructor and term variables, and ranged over by the metavariable s and variants. The additional syntax of λ^{ML}_{str} is given in Table 3. Note that contexts are extended to include declarations of structure identifiers; the same conventions regarding non-repetition of declarations and the associated use of contexts as functions apply here as well.

The judgement forms of λ^{ML} are extended with two additional formation judgements, and two additional equality judgements, summarized in Table 4. The rules for deriving judgements in λ^{ML}_{str} are obtained by extending the rules of λ^{ML} (taking contexts now in the extended sense) by the rules for structures given in Appendix B, together with the following two rules governing the use of structure variables:

$$([\,]\,E_1) \qquad \frac{\Phi\ \mathit{context}}{\Phi \gg s^{\mathsf{c}} : k}\ (\Phi(s) = [v{:}k, \sigma])$$

$$([\,]\,E_2) \qquad \frac{\Phi\ \mathit{context}}{\Phi \gg s^{\mathsf{r}} : [s^{\mathsf{c}}/v]\sigma}\ (\Phi(s) = [v{:}k, \sigma])$$

$\Phi \gg S\ sig$	S is a signature
$\Phi \gg M : S$	M is a structure of signature S
$\Phi \gg S_1 = S_2\ sig$	S_1 and S_2 are equal signatures
$\Phi \gg M_1 = M_2 : S$	M_1 and M_2 are equal structures of signature S

Table 4: λ^{ML}_{str} judgement forms

The notion of theory and derivability with respect to a theory remains the same as in λ^{ML}.

The calculus of structures may be understood in terms of a translation into the core calculus, which amounts to a proof that λ^{ML}_{str} may be interpreted into the category of modules of [Mog89a]. For Φ a λ^{ML}_{str} context, define Φ to be the λ^{ML} context obtained by replacing all structure variable declarations $s : [v{:}k, \sigma]$ by the pair of declarations $s^{\mathsf{c}} : k$ and $s^{\mathsf{r}} : [s^{\mathsf{c}}/v]\sigma$.

Lemma 3.1 *Let $\mathcal{T}$ be a well-formed λ^{ML} theory.*

1. *If* $\lambda^{ML}_{str}[\mathcal{T}] \vdash \Phi$ context, *then* $\lambda^{ML}[\mathcal{T}] \vdash \Phi$ context.
2. *For any λ^{ML} formation or equality judgement $\Phi \gg \alpha$, $\lambda^{ML}_{str}[\mathcal{T}] \vdash \Phi \gg \alpha$ iff $\lambda^{ML}[\mathcal{T}] \vdash \Phi \gg \alpha$.*
3. *$\lambda^{ML}_{str}[\mathcal{T}] \vdash \Phi \gg [u, e] : [v{:}k, \sigma]$ iff $\lambda^{ML}[\mathcal{T}] \vdash \Phi \gg u : k$ and $\lambda^{ML}[\mathcal{T}] \vdash \Phi \gg e : [u/v]\sigma$, and similarly for structure equality.*

Proof. By a straightforward induction on λ^{ML}_{str} derivations. ∎

It is an immediate consequence of this lemma and the decidability of type equivalence in λ^{ML} that type equivalence is λ^{ML}_{str} is decidable. This will be important for the decidability of type checking in the full modules calculus.

3.3 The Calculus of Modules

The fact that the category of modules of [Mog89a] is relatively Cartesian closed amounts to the fact that Σ and Π types are *definable* in λ^{ML}_{str}. This may seem surprising, since λ^{ML}_{str} is a rather minimal calculus of structures. The key idea is to regard *all* module expressions as "mixed-phase" entities, consisting of a compile-time part and a run-time part. For basic structures of the form $[u, e]$, the partitioning is clear: u, a constructor, is the compile-time part, and e, a term is the run-time part. For more complex module expressions such as functors, the separation is not quite so explicit. Consider the signature $S = [v{:}T, \mathrm{set}(v)]$, and let $F{:}S \to S$ be a functor. Since the type of this functor lies within the first-order fragment of λ^{ML}, we may rely on Standard ML for intuition. A functor such as F takes a structure of signature S as argument, and returns a structure, also of signature S. On the face of it, F might compute the type component of the result as a function of *both* the type and term component of the argument. However, no such computation is possible in ML since there are no primitives for building types from terms. Thus we may regard F as consisting of two parts, the compile-time part, which computes the type component of the result as a function of the type component of the argument, and the run-time part, which computes the term component of the result as a function of both the type and term component of the argument (recall that we are working in an explicitly-typed framework with explicit polymorphism.) In this way a functor may be regarded as a special form of structure.

For example, suppose that I is the identity functor $\lambda s{:}S.s$. Under this interpretation, I stands for the structure

$$[\lambda s^{\mathsf{c}}{:}T.s^{\mathsf{c}}, \Lambda s^{\mathsf{c}}{:}T.\lambda s^{\mathsf{r}}{:}\mathrm{set}(s^{\mathsf{c}}).s^{\mathsf{r}}]$$

of signature (but it is not the only signature for such a structure)

$$[f{:}T{\to}T, \forall s^{\mathsf{c}}{:}T.\,\mathrm{set}(s^{\mathsf{c}}{\to}f s^{\mathsf{c}})].$$

k	$\in$ *kind*	$::=$	$\ldots$
u	$\in$ *constr*	$::=$	$\ldots \mid Fst(M)$
σ	$\in$ *type*	$::=$	$\ldots$
e	$\in$ *term*	$::=$	$\ldots \mid Snd(M)$
S	$\in$ *sig*	$::=$	$[v{:}k, \sigma] \mid 1 \mid (\Sigma s{:}S_1.S_2) \mid (\Pi s{:}S_1.S_2)$
M	$\in$ *mod*	$::=$	$s \mid [u, e] \mid * \mid \langle M_1, M_2\rangle \mid \pi_i(M) \mid (\lambda s{:}S.M) \mid M_1\, M_2$
Φ	$\in$ *context*	$::=$	$\ldots \mid \Phi, s{:}S$

Table 5: λ^{ML}_{mod} raw expressions

In other words, I stands for the structure consisting of the identity constructor on types, and the polymorphic identity on terms.

If functors are to be regarded as structures, then application must correspondingly be extended to structures (of appropriate signature). In keeping with the above discussion, structure application may be defined as the structure resulting from the application of the first component of the functor to the first component of the argument, and the second component of the functor to both components of the argument. More precisely, if $[u, e]$ is a structure of signature $[f{:}k' \to k, \forall v'{:}k'.\sigma' \to [fv'/v]\sigma]$, and $[u', e']$ is a structure of signature $[v'{:}k', \sigma']$, then the application $[u, e]\,[u', e']$ is defined to be the structure $[uu', eue']$ of signature $[v{:}k, \sigma]$. As we shall see below, these typing conditions are satisfied whenever the first structure is the image of a functor under the translation sketched in the next paragraph. Moreover, both type correctness and equality are preserved under the translation.

Although in a sense λ^{ML}_{str} "already" has higher-order modules, the syntax for denoting them is quite unnatural since it forces the user to be explicit about the decomposition of module constructs into a compile-time and a run-time part, even in the case of first-order functors such as are found in ML. Our goal, then, is to consider a more natural notation similar to that used in [Mac86, MH88] that nonetheless respects the phase distinction. This is achieved by employing a non-standard equational theory that, when used during type checking, makes explicit the underlying "split" interpretation of module expressions, and hence eliminates apparent phase violations. For example, if A is a functor of signature $[t{:}T, \mathrm{set}(int)] \to [t{:}T, 1]$, then the type expression $\sigma = Fst(A\,[int, 3])$ is equal, using the non-standard rules, to $Fst(A)\;int$, which is free of run-time subexpressions. As a result, if e is a term of type σ, then the application

$$(\lambda x{:}\,\mathrm{set}(Fst(A\,[int, 5])).x)\;e$$

is type-correct, whereas in the absence of the non-standard equations this would not be so (assuming $3 \neq 5 : int$).

The raw syntax of λ^{ML}_{mod} is an extension of that of λ^{ML}; the extensions are given in Table 5. The judgement forms are the same as for λ^{ML}_{str}, and are axiomatized by the structure rules of Appendix B together with the rules of Appendix C. The λ^{ML}_{mod} calculus is parametric in a theory, defined as in λ^{ML} (*i.e.*, we do not admit module constants, or axioms governing module expressions.)

The formation rules of λ^{ML}_{mod} are essentially the standard rules for dependent strong sums and dependent function types. The equational rules include the expected rules for dependent types, together with the non-standard rules summarized in Table 6.

Beside the non-standard equational rules (and *orthogonal* to them), there is also a non-standard typing rules for structures:

$$\frac{\begin{array}{l}\Phi \gg M : [v{:}k, \sigma] \\ \Phi, v{:}k \gg \sigma' \ \mathit{type} \\ \Phi \gg Snd\ M : [Fst\ M/v]\sigma'\end{array}}{\Phi \gg M : [v{:}k, \sigma']}$$

The non-standard typing rule is *consistent* with the interpretation in the category of modules [Mog89a], but (we conjecture that) without it the main properties of λ^{ML}_{mod}, namely the compile-

Non-standard equational rules for signatures

$$(1 >) \qquad \frac{\Phi \ context}{\Phi \gg 1 = [v{:}1, 1] \ sig}$$

$$(\Sigma >) \qquad \frac{\Phi, v_1{:}k_1 \gg \sigma_1 \ type \quad \Phi, v_1{:}k_1, v_2{:}k_2 \gg \sigma_2 \ type}{\Phi \gg (\Sigma s{:}[v_1{:}k_1, \sigma_1].[v_2{:}k_2, [Fst(s)/v_1]\sigma_2]) = [v{:}k_1 \times k_2, [\pi_1 v/v_1]\sigma_1 \times [\pi_1 v, \pi_2 v/v_1, v_2]\sigma_2] \ sig}$$

$$(\Pi >) \qquad \frac{\Phi, v_1{:}k_1 \gg \sigma_1 \ type \quad \Phi, v_1{:}k_1, v_2{:}k_2 \gg \sigma_2 \ type}{\Phi \gg (\Pi s{:}[v_1{:}k_1, \sigma_1].[v_2{:}k_2, [Fst(s)/v_1]\sigma_2]) = [v{:}k_1 \rightarrow k_2, (\forall v_1{:}k_1.\sigma_1 \rightarrow [v\, v_1/v_2]\sigma_2)] \ sig}$$

Non-standard equational rules for modules

$$(1\ I >) \qquad \frac{\Phi \ context}{\Phi \gg * = [*, *] \ [v{:}1, 1]}$$

$$(\Sigma\ I >) \qquad \frac{\begin{array}{c}\Phi, v_1{:}k_1 \gg \sigma_1 \ type \quad \Phi, v_1{:}k_1, v_2{:}k_2 \gg \sigma_2 \ type \\ \Phi \gg u_1 : k_1 \quad \Phi \gg e_1 : [u_1/v_1]\sigma_1 \\ \Phi \gg u_2 : k_2 \quad \Phi \gg e_2 : [u_1, u_2/v_1, v_2]\sigma_2\end{array}}{\Phi \gg \langle [u_1, e_1], [u_2, e_2] \rangle = [\langle u_1, u_2 \rangle, \langle e_1, e_2 \rangle] : [v{:}k_1 \times k_2, [\pi_1 v/v_1]\sigma_1 \times [\pi_1 v, \pi_2 v/v_1, v_2]\sigma_2]}$$

$$(\Sigma\ E_1 >) \qquad \frac{\begin{array}{c}\Phi, v_1{:}k_1 \gg \sigma_1 \ type \quad \Phi, v_1{:}k_1, v_2{:}k_2 \gg \sigma_2 \ type \\ \Phi \gg u : k_1 \times k_2 \quad \Phi \gg e : [\pi_1 u/v_1]\sigma_1 \times [\pi_1 u, \pi_2 u/v_1, v_2]\sigma_2\end{array}}{\Phi \gg \pi_1[u, e] = [\pi_1 u, \pi_1 e] : [v_1{:}k_1, \sigma_1]}$$

$$(\Sigma\ E_2 >) \qquad \frac{\begin{array}{c}\Phi, v_1{:}k_1 \gg \sigma_1 \ type \quad \Phi, v_1{:}k_1, v_2{:}k_2 \gg \sigma_2 \ type \\ \Phi \gg u : k_1 \times k_2 \quad \Phi \gg e : [\pi_1 u/v_1]\sigma_1 \times [\pi_1 u, \pi_2 u/v_1, v_2]\sigma_2\end{array}}{\Phi \gg \pi_2[u, e] = [\pi_2 u, \pi_2 e] : [v_2{:}k_2, [\pi_1 u/v_1]\sigma_2]}$$

$$(\Pi\ I >) \qquad \frac{\begin{array}{c}\Phi, v_1{:}k_1 \gg \sigma_1 \ type \quad \Phi, v_1{:}k_1, v_2{:}k_2 \gg \sigma_2 \ type \\ \Phi, v_1{:}k_1 \gg u : k_2 \quad \Phi, v_1{:}k_1, x{:}\sigma_1 \gg e : [u/v_2]\sigma_2\end{array}}{\begin{array}{r}\Phi \gg (\lambda s{:}[v_1{:}k_1, \sigma_1].[Fst\, s, Snd\, s/v_1, x][u, e]) = [(\lambda v_1{:}k_1.u), (\Lambda v_1{:}k_1.\lambda x{:}\sigma_1.e)] : \\ [v{:}k_1 \rightarrow k_2, (\forall v_1{:}k_1.\sigma_1 \rightarrow [v\, v_1/v_2]\sigma_2)]\end{array}}$$

$$(\Pi\ E >) \qquad \frac{\begin{array}{l}\Phi, v_1{:}k_1 \gg \sigma_1 \ type \quad \Phi, v_1{:}k_1, v_2{:}k_2 \gg \sigma_2 \ type \\ \Phi \gg u_1 : k_1 \quad \Phi \gg e_1 : [u_1/v_1]\sigma_1 \\ \Phi \gg u : k_1 \rightarrow k_2 \quad \Phi \gg e : (\forall v_1{:}k_1.\sigma_1 \rightarrow [v\, v_1/v_2]\sigma_2)\end{array}}{\Phi \gg [u, e]\, [u_1, e_1] = [u\, u_1, e[u_1]\, e_1] : [v_2{:}k_2, [u_1/v_1]\sigma_2]}$$

Table 6: Non-standard equations

expression	translation	induction hypotheses
$Fst(M)$	u	where $M^\flat = [u, e]$
$Snd(M)$	e	where $M^\flat = [u, e]$
s	$[s^{\mathsf{c}}, s^{\mathsf{r}}]$	
$[v{:}k, \sigma]$	$[\mathbf{v}{:}k, [\mathbf{v}/v]\sigma^\flat]$	
1	$[\mathbf{v}{:}1, 1]$	
$(\Sigma s{:}S_1.S_2)$	$[\mathbf{v}{:}(k_1 \times k_2), ([\pi_1\mathbf{v}/\mathbf{v}]\sigma_1 \times [\pi_1\mathbf{v}, \pi_2\mathbf{v}/s^{\mathsf{c}}, \mathbf{v}]\sigma_2)]$	where $S_i^\flat = [\mathbf{v}{:}k_i, \sigma_i]$
$(\Pi s{:}S_1.S_2)$	$[\mathbf{v}{:}(k_1 \rightarrow k_2), \forall s^{\mathsf{c}}{:}k_1.[s^{\mathsf{c}}/\mathbf{v}]\sigma_1 \rightarrow [\mathbf{v}\, s^{\mathsf{c}}/\mathbf{v}]\sigma_2]$	where $S_i^\flat = [\mathbf{v}{:}k_i, \sigma_i]$
$*$	$[*, *]$	
$\langle M_1, M_2\rangle$	$[\langle u_1, u_2\rangle, \langle e_1, e_2\rangle]$	where $M_i^\flat = [u_i, e_i]$
$\pi_i M$	$[\pi_i u, \pi_i e]$	where $M^\flat = [u, e]$
$(\lambda s{:}S.M)$	$[(\lambda s^{\mathsf{c}}{:}k.u), (\Lambda s^{\mathsf{c}}{:}k.\lambda s^{\mathsf{r}}{:}[s^{\mathsf{c}}/\mathbf{v}]\sigma.e)]$	where $S^\flat = [\mathbf{v}{:}k, \sigma]$ and $M^\flat = [u, e]$
$M_1\, M_2$	$[u_1\, u_2, e_1[u_2]\, e_2]$	where $M_i^\flat = [u_i, e_i]$

Table 7: Translation of λ^{ML}_{mod} into λ^{ML}_{str}

time type checking theorem and the decidability of typing judgements, would still hold. The reason for having such rule is mainly pragmatic: to have a simple type checking algorithm (see Definition 3.9). Moreover, this additional typing rule captures a particularly natural property of Σ-types (once uniqueness of type has been abandoned), namely that a structure M should be *identified* with its expansion $[Fst\, M, Snd\, M]$. A typical example of typing judgement derivable by the non-standard typing rule is $s{:}[v{:}k, \sigma] \gg s : [v{:}k, [Fst\, s/v]\sigma]$.

3.4 Translation of λ^{ML}_{mod} into λ^{ML}_{str}

The non-standard equational theory used in the definition of λ^{ML}_{mod} is justified by proving that λ^{ML}_{mod} is a *definitional extension* of λ^{ML}_{str}, in a sense to be made precise below. This definitional extension result will then play an important role in establishing the decidability and compile-time type checking property of λ^{ML}_{mod}.

We begin by giving a translation $\_^\flat$ from raw λ^{ML}_{mod} expressions into raw λ^{ML}_{str} expressions. This translation is defined by induction on the structure of λ^{ML}_{mod} expressions. Apart from the cases given in Table 7, the translation is defined to commute with the expression constructors. For the basis we associate with every module variable s a constructor variable s^{c} and a term variable s^{r} in λ^{ML}_{str}. For convenience in defining the translation we fix a constructor variable $\mathbf{v}$ that may occur in expressions of λ^{ML}_{str}, but not in expressions of λ^{ML}_{mod}. Signatures of λ^{ML}_{mod} will be translated to λ^{ML}_{str} signatures of the form $[\mathbf{v}{:}k, \sigma]$. The translation is extended "declaration-wise" to contexts: $\Phi^\flat$ is obtained from Φ by replacing declarations of the form $x{:}\sigma$ by $x{:}\sigma^\flat$, and declarations of the form $s{:}S$ by $s{:}S^\flat$. Note that the translation leaves λ^{ML} expressions fixed; consequently, the translation need not be extended to theories.

Lemma 3.2 (Substitutivity) *The translation $\_^\flat$ commutes with substitution. In particular if $M^\flat = [u, e]$, then $([M/s]\_)^\flat = [u, e/s^{\mathsf{c}}, s^{\mathsf{r}}](\_^\flat)$.*

Proof. By induction on the structure of the raw expression $\_$. ∎

Theorem 3.3 ($\_^\flat$ interpretation) *Let $\mathcal{T}$ be a well-formed theory, and let $\mathcal{J}$ be a λ^{ML}_{mod} judgement. If $\lambda^{ML}_{mod}[\mathcal{T}] \vdash \mathcal{J}$, then $\lambda^{ML}_{str}[\mathcal{T}] \vdash \mathcal{J}^\flat$.*

Proof. By induction on the derivation of $\mathcal{J}$ in λ^{ML}_{mod}. The crucial points are

1. The non-standard equations become identities under translation.

2. Occurrences of $Fst(\_)$ and $Snd(\_)$ are eliminated; in particular, redices are contracted.

3. The module-level β and η redices become pairs of constructor- and term-level β and η redices.

$\blacksquare$

Conversely, λ^{ML}_{str} is essentially a sub-calculus of λ^{ML}_{mod}, differing only in the treatment of structure variables. To make this precise, define the embedding $\_^{\mathsf{e}}$ of λ^{ML}_{str} raw expressions into λ^{ML}_{mod} raw expressions by replacing all occurrences of s^{c} by $Fst(s)$, and all occurrences of s^{r} by $Snd(s)$.

Theorem 3.4 ($\_^{\mathsf{e}}$ interpretation) *Let $\mathcal{T}$ be a well-formed theory, and let $\mathcal{J}$ be a λ^{ML}_{str} judgement. If $\lambda^{ML}_{str}[\mathcal{T}] \vdash \mathcal{J}$, then $\lambda^{ML}_{mod}[\mathcal{T}] \vdash \mathcal{J}^{\mathsf{e}}$.*

Proof. By induction on the derivation of $\mathcal{J}$ in λ^{ML}_{str}. $\blacksquare$

Theorem 3.5 (Definitional Extension) *Let $\mathcal{T}$ be a well-formed theory.*

- *For any formation judgement $\mathcal{F}$ of λ^{ML}_{str}, if $\lambda^{ML}_{str}[\mathcal{T}] \vdash \mathcal{F}$, then $(\mathcal{F}^{\mathsf{e}})^{\flat}$ is syntactically equal to $\mathcal{F}$, modulo the names of bound variables.*
- *If $\lambda^{ML}_{mod}[\mathcal{T}] \vdash \Phi \gg M : S$, then the following equality judgements are derivable in $\lambda^{ML}_{mod}[\mathcal{T}]$:*
  - $\Phi_s \gg \Phi(s) = (\Phi(s)^{\flat})^{\mathsf{e}}\ \mathrm{sig}$, *for all $s \in \mathrm{Dom}(\Phi)$, where $\Phi \equiv \Phi_s, s{:}\Phi(s), \Phi^s$ (and similarly for x and v in $\mathrm{Dom}(\Phi)$)*
  - $\Phi \gg S = (S^{\flat})^{\mathsf{e}}\ \mathrm{sig}$
  - $\Phi \gg M = (M^{\flat})^{\mathsf{e}} : S$

  (and similarly for the other formation judgements.)

Proof. The first statment it is obvious, since $(\_^{\mathsf{e}})^{\flat}$ is Syntactically equal to $\_$ (up to α-conversion), for any λ^{ML}_{str} raw expression $\_$.

The second statment is proved by induction on the derivation of a formation judgement. Note that in the case of rules (*type eq*) and (*sig eq*) one uses the induction hypothesis plus rules (*type eq* $=$) and (*sig eq* $=$) respectively. $\blacksquare$

Corollary 3.6 (Conservative Extension) *Let $\mathcal{T}$ be an arbitrary well-formed theory. For any λ^{ML}_{str} judgement $\mathcal{J}$, $\lambda^{ML}_{mod}[\mathcal{T}] \vdash \mathcal{J}^{\mathsf{e}}$ iff $\lambda^{ML}_{str}[\mathcal{T}] \vdash \mathcal{J}$.*

3.5 Compile-Time Type Checking for λ^{ML}_{mod}

The compile-time equational theory of λ^{ML}_{mod} and λ^{ML}_{str} is defined to consist of all equational principles except those that would not be sound if the calculus of terms were extended to admit computational behaviors such as side-effects and non-termination.

Definition 3.7 (compile-time calculus) *Compile-time provability in λ^{ML}_{mod} and λ^{ML}_{str} is defined by disallowing the use of all β and η rules for term equivalence, and all β and η rules for module equivalence, apart from $([\,]\ \beta_1)/([\,]\ \beta_2)/([\,]\ \eta)$.*

Let us designate the β and η axioms for terms of λ^{ML} by $\beta\eta$, then the full λ^{ML}_{mod} calculus may be recovered by working in the theory $(\emptyset, \beta\eta)$, since the β and η axioms for modules are derivable in such a theory.

It may be easily verified that the variants of Theorems 3.3, 3.4 and 3.5 obtained by considering compile-time derivability hold.

Theorem 3.8 (Compile-time type checking) *Given any well-formed theory $\mathcal{T} = (\Phi^{\mathcal{T}}, \mathcal{A}^{\mathcal{T}})$, the following implications hold:*

If $\lambda^{ML}_{mod}[\mathcal{T}] \vdash$	*then* $\lambda^{ML}_{mod}[\Phi^{\mathcal{T}}, \emptyset] \vdash_{ct}$
Φ context	Φ context
$\Phi \gg \sigma$ type	$\Phi \gg \sigma$ type
$\Phi \gg S$ sig	$\Phi \gg S$ sig
$\Phi \gg u : k$	$\Phi \gg u : k$
$\Phi \gg e : \sigma$	$\Phi \gg e : \sigma$
$\Phi \gg M : S$	$\Phi \gg M : S$

If $\lambda^{ML}_{mod}[\mathcal{T}] \vdash$	*then* $\lambda^{ML}_{mod}[\Phi^{\mathcal{T}}, \emptyset] \vdash_{ct}$
$\Phi \gg \sigma_1 = \sigma_2$ type	$\Phi \gg \sigma_1 = \sigma_2$ type
$\Phi \gg S_1 = S_2$ sig	$\Phi \gg S_1 = S_2$ sig
$\Phi \gg u_1 = u_2 : k$	$\Phi \gg u_1 = u_2 : k$
$\Phi \gg e_1 = e_2 : \sigma$	$\Phi \gg e_i : \sigma$
$\Phi \gg M_1 = M_2 : S$	$\Phi \gg M_i : S$ $\Phi \gg [Fst\ M_1, Snd\ M_1] = [Fst\ M_2, Snd\ M_1] : S$

Proof. By induction on the derivation of a judgement in λ^{ML}_{mod}. ■

3.6 Decidability of λ^{ML}_{mod}

The decidability of λ^{ML}_{mod} is proved by giving an algorithm that "flattens" structures and signatures during type checking. As a result, checking signature equivalence is reduced to checking type equivalence in λ^{ML}_{str}, and this is, as we have already argued, decidable. The main complication in the algorithm stems from the failure of unicity of types. For example, the structure $[int, 3]$ has both of the inequivalent signatures $[t{:}T, \mathrm{set}(t)]$ and $[t{:}T, int]$. Our approach is to compute the "most specific" signature for a structure (in the foregoing example this would be the second) which will always have the form $[v{:}k, \sigma]$ where v does not occur free in σ. (As a notational convenience, we will usually omit explicit designation of the non-occurring variable, and write such signatures in the form $[{:}k, \sigma]$.) The algorithm defined below takes as input a raw context Φ and, for instance, a raw module expression M of λ^{ML}_{mod} and produces one of the following results:

- The context $\Phi^\flat$ and $M^\flat \equiv [u, e]{:}[{:}k, \sigma]$, meaning that $\Phi \gg M : [{:}k, \sigma]$ is derivable in λ^{ML}_{mod}.
- An error, meaning that Φ *context* is not derivable in λ^{ML}_{mod} or that $\Phi \gg M : S$ is not derivable in λ^{ML}_{mod} for any S.

Definition 3.9 (Type-checking algorithm) *The type-checking algorithm TC is given by a deterministic set of inference rules to derive judgements of the following form:*

input		*output*
Φ	$\twoheadrightarrow$	$\Phi^\flat$ context
$\Phi \gg \sigma$	$\twoheadrightarrow$	$\Phi^\flat \gg \sigma^\flat$ type
$\Phi \gg S$	$\twoheadrightarrow$	$\Phi^\flat \gg S^\flat$ sig
$\Phi \gg u$	$\twoheadrightarrow$	$\Phi^\flat \gg u^\flat : k$
$\Phi \gg e$	$\twoheadrightarrow$	$\Phi^\flat \gg e^\flat : \sigma$
$\Phi \gg M$	$\twoheadrightarrow$	$\Phi^\flat \gg M^\flat : [{:}k, \sigma]$

In the last three cases the algorithm TC not only computes the translation, but also a kind, type or signature. A sample of the inference rules that constitute the algorithm is given in Table 8.

TC is parametric in a theory $\mathcal{T}$, and we write $TC[\mathcal{T}]$ for the instance of the algorithm in which the constants declared in $\Phi^{\mathcal{T}}$ are regarded as variables. More precisely, $\Phi \twoheadrightarrow \Phi^\flat$ *context* in $TC[\mathcal{T}]$ iff $\Phi^{\mathcal{T}}, \Phi \twoheadrightarrow \Phi^{\mathcal{T}}, \Phi^\flat$ *context* in *TC*.

That the inference rules determine an algorithm is stated by the following theorem:

$$(\Phi, s{:}S) \qquad \frac{\Phi \gg S \;\twoheadrightarrow \Phi^\flat \gg S^\flat \;\; sig}{\Phi, s{:}S \;\twoheadrightarrow \Phi^\flat, s{:}S^\flat \;\; context} \; (s \notin \mathrm{Dom}(\Phi))$$

$$([\,]\; sig) \qquad \frac{\Phi, v{:}k \gg \sigma \;\twoheadrightarrow \Phi^\flat, v{:}k \gg \sigma^\flat \;\; type}{\Phi \gg [v{:}k, \sigma] \;\twoheadrightarrow \Phi^\flat \gg [v{:}k, \sigma^\flat] : sig}$$

$$([\,]\; I) \qquad \frac{\Phi \gg u \;\twoheadrightarrow \Phi^\flat \gg u^\flat : k \quad \Phi \gg e \;\twoheadrightarrow \Phi^\flat \gg e^\flat : \sigma}{\Phi \gg [u, e] \;\twoheadrightarrow \Phi^\flat \gg [u, e] : [{:}k, \sigma]}$$

$$([\,]\; E_1) \qquad \frac{\Phi \gg M \;\twoheadrightarrow \Phi^\flat \gg [u, e] : [{:}k, \sigma]}{\Phi \gg Fst(M) \;\twoheadrightarrow \Phi^\flat \gg u : k}$$

$$([\,]\; E_2) \qquad \frac{\Phi \gg M \;\twoheadrightarrow \Phi^\flat \gg [u, e] : [{:}k, \sigma]}{\Phi \gg Snd(M) \;\twoheadrightarrow \Phi^\flat \gg e : \sigma}$$

$$(var) \qquad \frac{\Phi \;\twoheadrightarrow \Phi^\flat \;\; context}{\Phi \gg s \;\twoheadrightarrow \Phi^\flat \gg [s^{\mathsf{c}}, s^{\mathsf{r}}] : [{:}k, [s^{\mathsf{c}}/\mathbf{v}]\sigma]} \; (\Phi^\flat(s) = [\mathbf{v}{:}k, \sigma])$$

$$(1\; I) \qquad \frac{\Phi \;\; context \twoheadrightarrow \Phi^\flat \;\; context}{\Phi \gg * \;\twoheadrightarrow \Phi^\flat \gg [*, *] : [{:}1, 1]}$$

$$(\Sigma\; I) \qquad \frac{\Phi \gg M_1 \;\twoheadrightarrow \Phi^\flat \gg [u_1, e_1] : [{:}k_1, \sigma_1] \quad \Phi \gg M_2 \;\twoheadrightarrow \Phi^\flat \gg [u_2, e_2] : [{:}k_2, \sigma_2]}{\Phi \gg \langle M_1, M_2 \rangle \;\twoheadrightarrow \Phi^\flat \gg [\langle u_1, u_2 \rangle, \langle e_1, e_2 \rangle] : [{:}k_1 \times k_2, \sigma_1 \times \sigma_2]}$$

$$(\Sigma\; E_i) \qquad \frac{\Phi \gg M \;\twoheadrightarrow \Phi^\flat \gg [u, e] : [{:}k_1 \times k_2, \sigma_1 \times \sigma_2]}{\Phi \gg \pi_i M \;\twoheadrightarrow \Phi^\flat \gg [\pi_i u, \pi_i e] : [{:}k_i, \sigma_i]}$$

$$(\Pi\; I) \qquad \frac{\Phi, s{:}S_1 \gg M \;\twoheadrightarrow \Phi^\flat, s{:}[\mathbf{v}{:}k_1, \sigma_1] \gg [u, e] : [{:}k_2, \sigma_2]}{\begin{array}{r}\Phi \gg (\lambda s{:}S_1.M) \;\twoheadrightarrow \Phi^\flat \gg [(\lambda s^{\mathsf{c}}{:}k_1.u), (\Lambda s^{\mathsf{c}}{:}k_1.\lambda s^{\mathsf{r}}{:}[s^{\mathsf{c}}/\mathbf{v}]\sigma_1.e)] : \\ [{:}k_1 \to k_2, \forall s^{\mathsf{c}}{:}k_1.[s^{\mathsf{c}}/\mathbf{v}]\sigma_1 \to \sigma_2]\end{array}}$$

$$(\Pi\; E) \qquad \frac{\begin{array}{l}\Phi \gg M \;\twoheadrightarrow \Phi^\flat \gg [u, e] : [{:}k_1 \to k_2, \forall v{:}k_1.\sigma_1 \to \sigma_2] \\ \Phi \gg M_1 \;\twoheadrightarrow \Phi^\flat \gg [u_1, e_1] : [{:}k_1, \sigma]\end{array}}{\Phi \gg M\, M_1 \;\twoheadrightarrow \Phi^\flat \gg [u\, u_1, e[u_1]\, e_1] : [{:}k_2, [u_1/v]\sigma_2]} \; \lambda^{ML}_{str} \vdash \Phi^\flat \gg \sigma = [u_1/v]\sigma_1 \;\; type$$

Table 8: Type checking algorithm (selected rules)

Theorem 3.10 (Decidability) *It is decidable whether a raw type-checking judgement* $lhs \twoheadrightarrow rhs$ *is derivable using the inference rules in Definition 3.9.*

Proof. By lexicographic induction on $\langle size(lhs), length(\Phi)\rangle$. ■

Theorem 3.11 (Soundness) *Let* $\mathcal{T}$ *be a well-formed theory. The following implications hold:*

If $TC[\mathcal{T}] \vdash$	*then* $\lambda^{ML}_{mod}[\mathcal{T}] \vdash_{ct}$
$\Phi \twoheadrightarrow \Phi^\flat$ context	Φ context
$\Phi \gg \sigma \twoheadrightarrow \Phi^\flat \gg \sigma^\flat$ type	$\Phi \gg \sigma$ type
$\Phi \gg S \twoheadrightarrow \Phi^\flat \gg S^\flat$ sig	$\Phi \gg S$ sig
$\Phi \gg u \twoheadrightarrow \Phi^\flat \gg u^\flat : k$	$\Phi \gg u : k$
$\Phi \gg e \twoheadrightarrow \Phi^\flat \gg e^\flat : \sigma$	$\Phi \gg e : \sigma^{\mathsf{e}}$
$\Phi \gg M \twoheadrightarrow \Phi^\flat \gg [u, e] : [:k, \sigma]$	$\Phi \gg M : [:k, \sigma^{\mathsf{e}}]$

Theorem 3.12 (Completeness) *Let* $\mathcal{T}$ *be any well-formed theory. The following implications hold:*

If $\lambda^{ML}_{mod}[\mathcal{T}] \vdash_{ct}$	*then* $TC[\mathcal{T}] \vdash$ & $\lambda^{ML}_{str}[\mathcal{T}] \vdash_{ct}$
Φ context	$\Phi \twoheadrightarrow \Phi^\flat$ context
$\Phi \gg \sigma$ type	$\Phi \gg \sigma \twoheadrightarrow \Phi^\flat \gg \sigma^\flat$ type
$\Phi \gg S$ sig	$\Phi \gg S \twoheadrightarrow \Phi^\flat \gg S^\flat$ sig
$\Phi \gg u : k$	$\Phi \gg u \twoheadrightarrow \Phi^\flat \gg u^\flat : k$
$\Phi \gg e : \sigma$	$\Phi \gg \sigma \twoheadrightarrow \Phi^\flat \gg \sigma^\flat$ type $\Phi \gg e \twoheadrightarrow \Phi^\flat \gg e^\flat : \sigma'$ $\Phi^\flat \gg \sigma^\flat = \sigma'$ type
$\Phi \gg M : S$	$\Phi \gg S \twoheadrightarrow \Phi^\flat \gg [\mathbf{v}{:}k, \sigma]$ sig $\Phi \gg M \twoheadrightarrow \Phi^\flat \gg [u, e] : [:k, \sigma']$ $\Phi^\flat \gg \sigma' = [u/\mathbf{v}]\sigma$ type

If $\lambda^{ML}_{mod}[\mathcal{T}] \vdash_{ct}$	*then* $TC[\mathcal{T}] \vdash$ & $\lambda^{ML}_{str}[\mathcal{T}] \vdash_{ct}$
$\Phi \gg \sigma_1 = \sigma_2$ type	$\Phi \gg \sigma_i \twoheadrightarrow \Phi^\flat \gg \sigma_i^\flat$ type $\Phi^\flat \gg \sigma_1^\flat = \sigma_2^\flat$ type
$\Phi \gg S_1 = S_2$ sig	$\Phi \gg S_i \twoheadrightarrow \Phi^\flat \gg S_i^\flat$ sig $\Phi^\flat \gg S_1^\flat = S_2^\flat$ sig
$\Phi \gg u_1 = u_2 : k$	$\Phi \gg u_i \twoheadrightarrow \Phi^\flat \gg u_i^\flat : k$ $\Phi^\flat \gg u_1^\flat = u_2^\flat : k$
$\Phi \gg e_1 = e_2 : \sigma$	$\Phi \gg \sigma \twoheadrightarrow \Phi^\flat \gg \sigma^\flat$ type $\Phi \gg e_i \twoheadrightarrow \Phi^\flat \gg e_i^\flat : \sigma_i$ $\Phi^\flat \gg \sigma^\flat = \sigma_i$ type $\Phi^\flat \gg e_1^\flat = e_2^\flat : \sigma^\flat$
$\Phi \gg M_1 = M_2 : S$	$\Phi \gg S \twoheadrightarrow \Phi^\flat \gg [\mathbf{v}{:}k, \sigma]$ sig $\Phi \gg M_i \twoheadrightarrow \Phi^\flat \gg [u_i, e_i] : [:k, \sigma_i]$ $\Phi^\flat \gg u_1 = u_2 : k$ $\Phi^\flat \gg \sigma = [u_i/\mathbf{v}]\sigma_i$ type $\Phi^\flat \gg e_1 = e_2 : \sigma$

Proof. By induction on the derivation of a judgement in λ^{ML}_{mod}. ■

Corollary 3.13 *Given any well-formed theory* $\mathcal{T}$*, the derivability of formation judgements in* $\lambda^{ML}_{mod}[\mathcal{T}]$ *is decidable and does not depend on run-time axioms nor the axioms in* $\mathcal{T}$.

4 Conclusion

Although the relatively straightforward ML-like function calculus XML of [MH88] illustrates some important properties of ML-like languages, it does not provide an adequate basis for the design of a compile-time type checker. Similar problems arise in other programming language models based on dependent types. To address this pragmatic issue, we have developed an alternate form of the XML calculus in which there is a clear compile-time/run-time distinction. Essentially, our technique is to add equational axioms that allow us to decompose structures and functors into separate compile-time and run-time components. While the phase distinction in λ^{ML} reduces to the syntactic difference between types and their elements, the general technique seems applicable to other forms of phase distinction.

The basis for our development is the "category of modules" over an indexed category, which is an instance of the Grothedieck construction. General properties of the category of modules are explained in the companion paper [Mog89a]. In the specific case of λ^{ML}, our non-standard equational axioms lead to a calculus which bears a natural relationship to the category of modules. In future work, it would be interesting to explore the exact connection between our calculus and the categorical construction, and to develop phase distinctions for languages whose type expressions may contain "run-time" subexpressions in more complicated ways.

References

[BL84] R. Burstall and B. Lampson. A kernel language for abstract data types and modules. In *Proc. Int. Symp. on Semantics of Data Types, Sophia-Antipolis (France), Springer LNCS 173*, pages 1–50, 1984.

[BMM89] K. B. Bruce, A. R. Meyer, and J. C. Mitchell. The semantics of second-order lambda calculus. *Information and Computation*, 1989. (to appear).

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

[Car88] L. Cardelli. Phase distinctions in type theory. Manuscript, 1988.

[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.

[HMM86] R. Harper, D.B. MacQueen, and R. Milner. Standard ml. Technical Report ECS–LFCS–86–2, Lab. for Foundations of Computer Science, University of Edinburgh, March 1986.

[HMT87a] R. Harper, R. Milner, and M. Tofte. The semantics of standard ML. Technical Report ECS–LFCS–87–36, Lab. for Foundations of Computer Science, University of Edinburgh, August 1987.

[HMT87b] R. Harper, R. Milner, and M. Tofte. A type discipline for program modules. In *TAPSOFT '87*, volume 250 of *LNCS*. Springer-Verlag, March 1987.

[Mac86] D.B. MacQueen. Using dependent types to express modular structure. In *Proc. 13-th ACM Symp. on Principles of Programming Languages*, pages 277–286, 1986.

[Mar84] P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, Napoli, 1984.

[MH88] J.C. Mitchell and R. Harper. The essence of ML. In *Proc. 15-th ACM Symp. on Principles of Programming Languages*, pages 28–46, January 1988.

[Mog89a] E. Moggi. A category-theoretic account of program modules. In *Summer Conf. on Category Theory and Computer Science*, pages 101–117, 1989.

[Mog89b] E. Moggi. Computational lambda calculus and monads. In *Fourth IEEE Symp. Logic in Computer Science*, pages 14–23, 1989.

[MP88] J.C. Mitchell and G.D. Plotkin. Abstract types have existential types. *ACM Trans. on Programming Languages and Systems*, 10(3):470–502, 1988. Preliminary version appeared in *Proc. 12-th ACM Symp. on Principles of Programming Languages,* 1985.

[NPS88] B. Nordstrom, K. Peterson, and J. Smith. Programming in martin-löf's type theory. University of Gothenburg / Chalmers Institue of Technology, Book draft of Midsummer 1988.

[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.

[Tof87] M. Tofte. *Operational Semantics and Polymorphic Type Inference.* PhD thesis, University of Edinburgh, 1987.

A Core calculus

Contexts: Φ *context*

$(\Phi = \emptyset)$ $$\frac{}{\emptyset\ context}$$

$(\Phi, v{:}k)$ $$\frac{\Phi\ context}{\Phi, v{:}k\ context}\ (v \notin \mathrm{Dom}(\Phi))$$

$(\Phi, x{:}\sigma)$ $$\frac{\Phi \gg \sigma\ type}{\Phi, x{:}\sigma\ context}\ (x \notin \mathrm{Dom}(\Phi))$$

Constructors: $\Phi \gg u : k$

(var) $$\frac{\Phi\ context}{\Phi \gg v : k}\ (\Phi(v) = k)$$

$(1\ T)$ $$\frac{\Phi\ context}{\Phi \gg 1 : T}$$

$(\times\ T)$ $$\frac{\Phi\ context}{\Phi \gg \times : T \to T \to T}$$

$(\to T)$ $$\frac{\Phi\ context}{\Phi \gg \to : T \to T \to T}$$

$(1\ I)$ $$\frac{\Phi\ context}{\Phi \gg * : 1}$$

$(\times\ I)$ $$\frac{\Phi \gg u_1 : k_1 \quad \Phi \gg u_2 : k_2}{\Phi \gg \langle u_1, u_2 \rangle : k_1 \times k_2}$$

$(\times\ E)$ $$\frac{\Phi \gg u : k_1 \times k_2}{\Phi \gg \pi_i(u) : k_i}\ (i = 1, 2)$$

$(\rightarrow\ I)$ $$\frac{\Phi, v{:}k_1 \gg u : k_2}{\Phi \gg (\lambda v{:}k_1.u) : k_1 \rightarrow k_2}$$

$(\rightarrow\ E)$ $$\frac{\Phi \gg u_1 : k_1 \rightarrow k_2 \quad \Phi \gg u_2 : k_1}{\Phi \gg u_1\, u_2 : k_2}$$

Types: $\Phi \gg \sigma\ \mathit{type}$

$(T \subseteq \mathit{type})$ $$\frac{\Phi \gg \tau : T}{\Phi \gg \mathrm{set}(\tau)\ \mathit{type}}$$

$(1\ \mathit{type})$ $$\frac{\Phi\ \mathit{context}}{\Phi \gg 1\ \mathit{type}}$$

$(\times\ \mathit{type})$ $$\frac{\Phi \gg \sigma_1\ \mathit{type} \quad \Phi \gg \sigma_2\ \mathit{type}}{\Phi \gg \sigma_1 \times \sigma_2\ \mathit{type}}$$

$(\rightarrow\ \mathit{type})$ $$\frac{\Phi \gg \sigma_1\ \mathit{type} \quad \Phi \gg \sigma_2\ \mathit{type}}{\Phi \gg \sigma_1 \rightarrow \sigma_2\ \mathit{type}}$$

$(\forall\ \mathit{type})$ $$\frac{\Phi, v{:}k \gg \sigma\ \mathit{type}}{\Phi \gg (\forall v{:}k.\sigma)\ \mathit{type}}$$

Terms: $\Phi \gg e : \sigma$

(var) $$\frac{\Phi\ \mathit{context}}{\Phi \gg x : \sigma}\ (\Phi(x) = \sigma)$$

$(\forall\ I)$ $$\frac{\Phi, v{:}k \gg e : \sigma}{\Phi \gg (\Lambda v{:}k.e) : (\forall v{:}k.\sigma)}$$

$(\forall\ E)$ $$\frac{\Phi \gg e : (\forall v{:}k.\sigma) \quad \Phi \gg u : k}{\Phi \gg e[u] : [u/v]\sigma}$$

$(1\ I)$ $$\frac{\Phi\ \mathit{context}}{\Phi \gg * : 1}$$

$(\times\ I)$ $$\frac{\Phi \gg e_1 : \sigma_1 \quad \Phi \gg e_2 : \sigma_2}{\Phi \gg \langle e_1, e_2 \rangle : \sigma_1 \times \sigma_2}$$

$(\times\ E)$ $$\frac{\Phi \gg e : \sigma_1 \times \sigma_2}{\Phi \gg \pi_i e : \sigma_i}\ (i = 1, 2)$$

$(\rightarrow\ I)$ $$\frac{\Phi, x{:}\sigma_1 \gg e : \sigma_2}{\Phi \gg (\lambda x{:}\sigma_1.e) : \sigma_1 \rightarrow \sigma_2}$$

$(\rightarrow\ E)$ $$\frac{\Phi \gg e_2 : \sigma_1 \rightarrow \sigma_2 \quad \Phi \gg e_1 : \sigma_1}{\Phi \gg e_2\, e_1 : \sigma_2}$$

$(\mathit{type\ eq})$ $$\frac{\Phi \gg e : \sigma_1 \quad \Phi \gg \sigma_1 = \sigma_2\ \mathit{type}}{\Phi \gg e : \sigma_2}$$

Constructors: $\Phi \gg u_1 = u_2 : k$

$(refl)$ $$\frac{\Phi \gg u : k}{\Phi \gg u = u : k}$$

$(symm)$ $$\frac{\Phi \gg u_1 = u_2 : k}{\Phi \gg u_2 = u_1 : k}$$

$(trans)$ $$\frac{\Phi \gg u_1 = u_2 : k \quad \Phi \gg u_2 = u_3 : k}{\Phi \gg u_1 = u_3 : k}$$

$(1\ \eta)$ $$\frac{\Phi \gg u : 1}{\Phi \gg u = * : 1}$$

$(\times\ I =)$ $$\frac{\Phi \gg u_1 = u_1' : k_1 \quad \Phi \gg u_2 = u_2' : k_2}{\Phi \gg \langle u_1, u_2 \rangle = \langle u_1', u_2' \rangle : k_1 \times k_2}$$

$(\times\ E =)$ $$\frac{\Phi \gg u = u' : k_1 \times k_2}{\Phi \gg \pi_i(u) = \pi_i(u') : k_i}\ (i = 1, 2)$$

$(\times\ \beta)$ $$\frac{\Phi \gg u_1 : k_1 \quad \Phi \gg u_2 : k_2}{\Phi \gg \pi_i(\langle u_1, u_2 \rangle) = u_i : k_i}\ (i = 1, 2)$$

$(\times\ \eta)$ $$\frac{\Phi \gg u : k_1 \times k_2}{\Phi \gg \langle \pi_1(u), \pi_2(u) \rangle = u : k_1 \times k_2}$$

$(\rightarrow\ I =)$ $$\frac{\Phi, v{:}k_1 \gg u = u' : k_2}{\Phi \gg (\lambda v{:}k_1.u) = (\lambda v{:}k_1.u') : k_1 \rightarrow k_2}$$

$(\rightarrow\ E =)$ $$\frac{\Phi \gg u = u' : k_1 \rightarrow k_2 \quad \Phi \gg u_1 = u_1' : k_1}{\Phi \gg u\, u_1 = u'\, u_1' : k_2}$$

$(\rightarrow\ \beta)$ $$\frac{\Phi \gg u_1 : k_1 \quad \Phi, v{:}k_1 \gg u_2 : k_2}{\Phi \gg (\lambda v{:}k_1.u_2)\, u_1 = [u_1/v]u_2 : k_2}$$

$(\rightarrow\ \eta)$ $$\frac{\Phi \gg u : k_1 \rightarrow k_2}{\Phi \gg (\lambda v{:}k_1.u\, v) = u : k_1 \rightarrow k_2}\ (v \notin \mathrm{Dom}(\Phi))$$

Types: $\Phi \gg \sigma_1 = \sigma_2\ \ type$

$(refl)$ $$\frac{\Phi \gg \sigma\ \ type}{\Phi \gg \sigma = \sigma\ \ type}$$

$(symm)$ $$\frac{\Phi \gg \sigma_1 = \sigma_2\ \ type}{\Phi \gg \sigma_2 = \sigma_1\ \ type}$$

$(trans)$ $$\frac{\Phi \gg \sigma_1 = \sigma_2\ \ type \quad \Phi \gg \sigma_2 = \sigma_3\ \ type}{\Phi \gg \sigma_1 = \sigma_3\ \ type}$$

$(T \subseteq type =)$ $$\frac{\Phi \gg \tau = \tau' : T}{\Phi \gg \mathrm{set}(\tau) = \mathrm{set}(\tau')\ type}$$

$(\times\ type =)$ $$\frac{\Phi \gg \sigma_1 = \sigma_1'\ type \quad \Phi \gg \sigma_2 = \sigma_2'\ type}{\Phi \gg \sigma_1 \times \sigma_2 = \sigma_1' \times \sigma_2'\ type}$$

$(\rightarrow\ type =)$ $$\frac{\Phi \gg \sigma_1 = \sigma_1'\ type \quad \Phi \gg \sigma_2 = \sigma_2'\ type}{\Phi \gg \sigma_1 \rightarrow \sigma_2 = \sigma_1' \rightarrow \sigma_2'\ type}$$

$(\forall\ type =)$ $$\frac{\Phi, v{:}k \gg \sigma = \sigma'\ type}{\Phi \gg (\forall v{:}k.\sigma) = (\forall v{:}k.\sigma')\ type}$$

$(1\ T =)$ $$\frac{\Phi\ context}{\Phi \gg \mathrm{set}(1) = 1\ type}$$

$(\times\ T =)$ $$\frac{\Phi \gg \tau_1 : T \quad \Phi \gg \tau_2 : T}{\Phi \gg \mathrm{set}(\tau_1 \times \tau_2) = \mathrm{set}(\tau_1) \times \mathrm{set}(\tau_2)\ type}$$

$(\rightarrow T =)$ $$\frac{\Phi \gg \tau_1 : T \quad \Phi \gg \tau_2 : T}{\Phi \gg \mathrm{set}(\tau_1 \rightarrow \tau_2) = \mathrm{set}(\tau_1) \rightarrow \mathrm{set}(\tau_2)\ type}$$

Terms: $\Phi \gg e_1 = e_2 : \sigma$

$(refl)$ $$\frac{\Phi \gg e : \sigma}{\Phi \gg e = e : \sigma}$$

$(symm)$ $$\frac{\Phi \gg e_1 = e_2 : \sigma}{\Phi \gg e_2 = e_1 : \sigma}$$

$(trans)$ $$\frac{\Phi \gg e_1 = e_2 : \sigma \quad \Phi \gg e_2 = e_3 : \sigma}{\Phi \gg e_1 = e_3 : \sigma}$$

$(type\ eq =)$ $$\frac{\Phi \gg e_1 = e_2 : \sigma_1 \quad \Phi \gg \sigma_1 = \sigma_2\ type}{\Phi \gg e_1 = e_2 : \sigma_2}$$

$(1\ \eta)$ $$\frac{\Phi \gg e : 1}{\Phi \gg e = * : 1}$$

$(\times\ I =)$ $$\frac{\Phi \gg e_1 = e_1' : \sigma_1 \quad \Phi \gg e_2 = e_2' : \sigma_2}{\Phi \gg \langle e_1, e_2 \rangle = \langle e_1', e_2' \rangle : \sigma_1 \times \sigma_2}$$

$(\times\ E =)$ $$\frac{\Phi \gg e = e' : \sigma_1 \times \sigma_2}{\Phi \gg \pi_i(e) = \pi_i(e') : \sigma_i}\ (i = 1, 2)$$

$(\times\ \beta)$ $$\frac{\Phi \gg e_1 : \sigma_1 \quad \Phi \gg e_2 : \sigma_2}{\Phi \gg \pi_i(\langle e_1, e_2 \rangle) = e_i : \sigma_i}\ (i = 1, 2)$$

$(\times\ \eta)$ $$\frac{\Phi \gg e : \sigma_1 \times \sigma_2}{\Phi \gg \langle \pi_1(e), \pi_2(e) \rangle = e : \sigma_1 \times \sigma_2}$$

$(\rightarrow I =)$ $$\frac{\Phi \gg \sigma_1 = \sigma_1' \ type \quad \Phi, x{:}\sigma_1 \gg e = e' : \sigma_2}{\Phi \gg (\lambda x{:}\sigma_1.e) = (\lambda x{:}\sigma_1'.e') : \sigma_1 \rightarrow \sigma_2}$$

$(\rightarrow E =)$ $$\frac{\Phi \gg e = e' : \sigma_1 \rightarrow \sigma_2 \quad \Phi \gg e_1 = e_1' : \sigma_1}{\Phi \gg e\, e_1 = e'\, e_1' : \sigma_2}$$

$(\rightarrow \beta)$ $$\frac{\Phi \gg e_1 : \sigma_1 \quad \Phi, x{:}\sigma_1 \gg e_2 : \sigma_2}{\Phi \gg (\lambda x{:}\sigma_1.e_2)\, e_1 = [e_1/x]e_2 : \sigma_2}$$

$(\rightarrow \eta)$ $$\frac{\Phi \gg e : \sigma_1 \rightarrow \sigma_2}{\Phi \gg (\lambda x{:}\sigma_1.e\, x) = e : \sigma_1 \rightarrow \sigma_2}\ (x \notin \mathrm{Dom}(\Phi))$$

$(\forall I =)$ $$\frac{\Phi, v{:}k \gg e = e' : \sigma}{\Phi \gg (\Lambda v{:}k.e) = (\Lambda v{:}k.e') : (\forall v{:}k.\sigma)}$$

$(\forall E =)$ $$\frac{\Phi \gg e = e' : (\forall v{:}k.\sigma) \quad \Phi \gg u_1 = u_1' : k}{\Phi \gg e[u] = e'[u'] : [u/v]\sigma}$$

$(\forall \beta)$ $$\frac{\Phi \gg u : k \quad \Phi, v{:}k \gg e : \sigma}{\Phi \gg (\Lambda v{:}k.e)[u] = [u/v]e : [u/v]\sigma}$$

$(\forall \eta)$ $$\frac{\Phi \gg e : (\forall v{:}k.\sigma)}{\Phi \gg (\Lambda v{:}k.e[v]) = e : (\forall v{:}k.\sigma)}\ (v \notin \mathrm{Dom}(\Phi))$$

B Common Rules for Structures

Contexts: Φ *context*

$(\Phi, s{:}S)$ $$\frac{\Phi \gg S \ sig}{\Phi, s{:}S \ context}\ (s \notin \mathrm{Dom}(\Phi))$$

Signatures: $\Phi \gg S \ sig$

$([\,]\ sig)$ $$\frac{\Phi, v{:}k \gg \sigma \ type}{\Phi \gg [v{:}k, \sigma] : sig}$$

Modules: $\Phi \gg M : S$

$([\,]\ I)$ $$\frac{\Phi, v{:}k \gg \sigma \ type \quad \Phi \gg u : k \quad \Phi \gg e : [u/v]\sigma}{\Phi \gg [u, e] : [v{:}k, \sigma]}$$

$(sig\ eq)$ $$\frac{\Phi \gg M : S_1 \quad S_1 = S_2 \ sig}{\Phi \gg M : S_1}$$

Signatures: $\Phi \gg S_1 = S_2 \;\; sig$

$$(refl) \qquad \frac{\Phi \gg S \;\; sig}{\Phi \gg S = S \;\; sig}$$

$$(symm) \qquad \frac{\Phi \gg S_1 = S_2 \;\; sig}{\Phi \gg S_2 = S_1 \;\; sig}$$

$$(trans) \qquad \frac{\Phi \gg S_1 = S_2 \;\; sig \quad \Phi \gg S_2 = S_3 \;\; sig}{\Phi \gg S_1 = S_3 \;\; sig}$$

$$([\,]\; sig =) \qquad \frac{\Phi, v{:}k \gg \sigma = \sigma' \;\; type}{\Phi \gg [v{:}k, \sigma] = [v{:}k, \sigma'] \;\; sig}$$

Modules: $\Phi \gg M_1 = M_2 : S$

$$(refl) \qquad \frac{\Phi \gg M : S}{\Phi \gg M = M : S}$$

$$(symm) \qquad \frac{\Phi \gg M_1 = M_2 : S}{\Phi \gg M_2 = M_1 : S}$$

$$(trans) \qquad \frac{\Phi \gg M_1 = M_2 : S \quad \Phi \gg M_2 = M_3 : S}{\Phi \gg M_1 = M_3 : S}$$

$$(sig\ eq =) \qquad \frac{\Phi \gg M_1 = M_2 : S_1 \quad \Phi \gg S_1 = S_2 \;\; sig}{\Phi \gg M_1 = M_2 : S_2}$$

$$([\,]\; I =) \qquad \frac{\Phi, v{:}k \gg \sigma \;\; type \quad \Phi \gg u = u' : k \quad \Phi \gg e = e' : [u/v]\sigma}{\Phi \gg [u, e] = [u', e'] : [v{:}k, \sigma]}$$

C Module Calculus

Constructors: $\Phi \gg u : k$

$$([\,]\; E_1) \qquad \frac{\Phi \gg M : [v{:}k, \sigma]}{\Phi \gg Fst(M) : k}$$

Terms: $\Phi \gg e : \sigma$

$$([\,]\; E_2) \qquad \frac{\Phi \gg M : [v{:}k, \sigma]}{\Phi \gg Snd(M) : [Fst(M)/v]\sigma}$$

Signatures: $\Phi \gg S \;\; sig$

$$(1\; sig) \qquad \frac{\Phi \;\; context}{\Phi \gg 1 \;\; sig}$$

$$(\Sigma\; sig) \qquad \frac{\Phi, s{:}S_1 \gg S_2 \;\; sig}{\Phi \gg (\Sigma s{:}S_1.S_2) \;\; sig}$$

$$(\Pi\; sig) \qquad \frac{\Phi, s{:}S_1 \gg S_2 \;\; sig}{\Phi \gg (\Pi s{:}S_1.S_2) \;\; sig}$$

Modules: $\Phi \gg M : S$

$$(var) \qquad \frac{\Phi\ context}{\Phi \gg s : S}\ (\Phi(s) = S)$$

$$(1\ I) \qquad \frac{\Phi\ context}{\Phi \gg * : 1}$$

$$(\Sigma\ I) \qquad \frac{\Phi, s{:}S_1 \gg S_2\ sig \quad \Phi \gg M_1 : S_1 \quad \Phi \gg M_2 : [M_1/s]S_2}{\Phi \gg \langle M_1, M_2\rangle : (\Sigma s{:}S_1.S_2)}$$

$$(\Sigma\ E_1) \qquad \frac{\Phi \gg M : (\Sigma s{:}S_1.S_2)}{\Phi \gg \pi_1 M : S_1}$$

$$(\Sigma\ E_2) \qquad \frac{\Phi \gg M : (\Sigma s{:}S_1.S_2)}{\Phi \gg \pi_2 M : [\pi_1 M/s]S_1}$$

$$(\Pi\ I) \qquad \frac{\Phi, s{:}S_1 \gg M : S_2}{\Phi \gg (\lambda s{:}S_1.M) : (\Pi s{:}S_1.S_2)}$$

$$(\Pi\ E) \qquad \frac{\Phi \gg M : (\Pi s{:}S_1.S_2) \quad \Phi \gg M_1 : S_1}{\Phi \gg M\ M_1 : [M_1/s]S_2}$$

Non-standard typing rule for structures

$$\frac{\begin{array}{l}\Phi \gg M : [v{:}k, \sigma] \\ \Phi, v{:}k \gg \sigma'\ type \\ \Phi \gg Snd\ M : [Fst\ M/v]\sigma'\end{array}}{\Phi \gg M : [v{:}k, \sigma']}$$

Constructors: $\Phi \gg u_1 = u_2 : k$

$$([\,]\ E_1 =) \qquad \frac{\Phi \gg M = M' : [v{:}k, \sigma]}{\Phi \gg Fst(M) = Fst(M') : k}$$

$$([\,]\ \beta_1) \qquad \frac{\Phi, v{:}k \gg \sigma\ type \quad \Phi \gg u : k \quad \Phi \gg e : [u/v]\sigma}{\Phi \gg Fst([u, e]) = u : k}$$

Terms: $\Phi \gg e_1 = e_2 : \sigma$

$$([\,]\ E_2 =) \qquad \frac{\Phi \gg M = M' : [v{:}k, \sigma]}{\Phi \gg Snd(M) = Snd(M') : [Fst(M)/v]\sigma}$$

$$([\,]\ \beta_2) \qquad \frac{\Phi, v{:}k \gg \sigma\ type \quad \Phi \gg u : k \quad \Phi \gg e : [u/v]\sigma}{\Phi \gg Snd([u, e]) = e : [u/v]\sigma}$$

Signatures: $\Phi \gg S_1 = S_2\ sig$

$$(\Sigma\ sig =) \qquad \frac{\Phi \gg S_1 = S_1'\ sig \quad \Phi, s{:}S_1 \gg S_2 = S_2'\ sig}{\Phi \gg (\Sigma s{:}S_1.S_2) = (\Sigma s{:}S_1'.S_2')\ sig}$$

$$(\Pi\ sig =) \qquad \frac{\Phi \gg S_1 = S_1'\ sig \quad \Phi, s{:}S_1 \gg S_2 = S_2'\ sig}{\Phi \gg (\Pi s{:}S_1.S_2) = (\Pi s{:}S_1'.S_2')\ sig}$$

Non-standard equational rules for signatures

$(1 >)$ $$\frac{\Phi\ \mathit{context}}{\Phi \gg 1 = [v{:}1, 1]\ \mathit{sig}}$$

$(\Sigma >)$ $$\frac{\Phi, v_1{:}k_1 \gg \sigma_1\ \mathit{type} \quad \Phi, v_1{:}k_1, v_2{:}k_2 \gg \sigma_2\ \mathit{type}}{\Phi \gg (\Sigma s{:}[v_1{:}k_1, \sigma_1].[v_2{:}k_2, [\mathit{Fst}(s)/v_1]\sigma_2]) = [v{:}k_1 \times k_2, [\pi_1 v/v_1]\sigma_1 \times [\pi_1 v, \pi_2 v/v_1, v_2]\sigma_2]\ \mathit{sig}}$$

$(\Pi >)$ $$\frac{\Phi, v_1{:}k_1 \gg \sigma_1\ \mathit{type} \quad \Phi, v_1{:}k_1, v_2{:}k_2 \gg \sigma_2\ \mathit{type}}{\Phi \gg (\Pi s{:}[v_1{:}k_1, \sigma_1].[v_2{:}k_2, [\mathit{Fst}(s)/v_1]\sigma_2]) = [v{:}k_1 \rightarrow k_2, (\forall v_1{:}k_1.\sigma_1 \rightarrow [v\, v_1/v_2]\sigma_2)]\ \mathit{sig}}$$

Modules: $\Phi \gg M_1 = M_2 : S$

$([\,]\ \eta)$ $$\frac{\Phi \gg M : [v{:}k, \sigma]}{\Phi \gg \langle \mathit{Fst}(M), \mathit{Snd}(M)\rangle = M : [v{:}k, \sigma]}$$

$(1\ \eta)$ $$\frac{\Phi \gg M : 1}{\Phi \gg M = * : 1}$$

$(\Sigma\ I =)$ $$\frac{\Phi, s{:}S_1 \gg S_2\ \mathit{sig} \quad \Phi \gg M_1 = M_1' : S_1 \quad \Phi \gg M_2 = M_2' : [M_1/s]S_2}{\Phi \gg \langle M_1, M_2\rangle = \langle M_1', M_2'\rangle : (\Sigma s{:}S_1.S_2)}$$

$(\Sigma\ E_1 =)$ $$\frac{\Phi \gg M = M' : (\Sigma s{:}S_1.S_2)}{\Phi \gg \pi_1(M) = \pi_1(M') : S_1}$$

$(\Sigma\ E_2 =)$ $$\frac{\Phi \gg M = M' : (\Sigma s{:}S_1.S_2)}{\Phi \gg \pi_2(M) = \pi_2(M') : [\pi_1(M)/s]S_2}$$

$(\Sigma\ \beta_1)$ $$\frac{\Phi, s{:}S_1 \gg S_2\ \mathit{sig} \quad \Phi \gg M_1 : S_1 \quad \Phi \gg M_2 : [M_1/s]S_2}{\Phi \gg \pi_1(\langle M_1, M_2\rangle) = M_1 : S_1}$$

$(\Sigma\ \beta_2)$ $$\frac{\Phi, s{:}S_1 \gg S_2\ \mathit{sig} \quad \Phi \gg M_1 : S_1 \quad \Phi \gg M_2 : [M_1/s]S_2}{\Phi \gg \pi_2(\langle M_1, M_2\rangle) = M_2 : [M_1/s]S_2}$$

$(\Sigma\ \eta)$ $$\frac{\Phi \gg M : (\Sigma s{:}S_1.S_2)}{\Phi \gg \langle \pi_1(M), \pi_2(M)\rangle = M : (\Sigma s{:}S_1.S_2)}$$

$(\Pi\ I =)$ $$\frac{\Phi \gg S_1 = S_1'\ \mathit{sig} \quad \Phi, s{:}S_1 \gg M = M' : S_2}{\Phi \gg (\lambda s{:}S_1.M) = (\lambda s{:}S_1'.M') : (\Pi s{:}S_1.S_2)}$$

$(\Pi\ E =)$ $$\frac{\Phi \gg M = M' : (\Pi s{:}S_1.S_2) \quad \Phi \gg M_1 = M_1' : S_1}{\Phi \gg M\ M_1 = M'\ M_1' : [M_1/s]S_2}$$

$(\Pi\ \beta)$ $$\frac{\Phi \gg M_1 : S_1 \quad \Phi, s{:}S_1 \gg M_2 : S_2}{\Phi \gg (\lambda s{:}S_1.M_2)\ M_1 = [M_1/s]M_2 : [M_1/s]S_2}$$

$(\Pi\ \eta)$ $$\frac{\Phi \gg M : (\Pi s{:}S_1.S_2)}{\Phi \gg (\lambda s{:}S_1.M\ s) = M : (\Pi s{:}S_1.S_2)}\ (s \notin \mathrm{Dom}(\Phi))$$

Non-standard equational rules for modules

$$(1\ I >)\qquad \frac{\Phi\ \mathit{context}}{\Phi \gg * = [*,*]\ [v{:}1,1]}$$

$$(\Sigma\ I >)\qquad \frac{\begin{array}{l}\Phi, v_1{:}k_1 \gg \sigma_1\ \mathit{type} \quad \Phi, v_1{:}k_1, v_2{:}k_2 \gg \sigma_2\ \mathit{type}\\ \Phi \gg u_1 : k_1 \quad \Phi \gg e_1 : [u_1/v_1]\sigma_1\\ \Phi \gg u_2 : k_2 \quad \Phi \gg e_2 : [u_1, u_2/v_1, v_2]\sigma_2\end{array}}{\Phi \gg \langle [u_1, e_1], [u_2, e_2]\rangle = [\langle u_1, u_2\rangle, \langle e_1, e_2\rangle] : [v{:}k_1 \times k_2, [\pi_1 v/v_1]\sigma_1 \times [\pi_1 v, \pi_2 v/v_1, v_2]\sigma_2]}$$

$$(\Sigma\ E_1 >)\qquad \frac{\begin{array}{l}\Phi, v_1{:}k_1 \gg \sigma_1\ \mathit{type} \quad \Phi, v_1{:}k_1, v_2{:}k_2 \gg \sigma_2\ \mathit{type}\\ \Phi \gg u : k_1 \times k_2 \quad \Phi \gg e : [\pi_1 u/v_1]\sigma_1 \times [\pi_1 u, \pi_2 u/v_1, v_2]\sigma_2\end{array}}{\Phi \gg \pi_1[u, e] = [\pi_1 u, \pi_1 e] : [v_1{:}k_1, \sigma_1]}$$

$$(\Sigma\ E_2 >)\qquad \frac{\begin{array}{l}\Phi, v_1{:}k_1 \gg \sigma_1\ \mathit{type} \quad \Phi, v_1{:}k_1, v_2{:}k_2 \gg \sigma_2\ \mathit{type}\\ \Phi \gg u : k_1 \times k_2 \quad \Phi \gg e : [\pi_1 u/v_1]\sigma_1 \times [\pi_1 u, \pi_2 u/v_1, v_2]\sigma_2\end{array}}{\Phi \gg \pi_2[u, e] = [\pi_2 u, \pi_2 e] : [v_2{:}k_2, [\pi_1 u/v_1]\sigma_2]}$$

$$(\Pi\ I >)\qquad \frac{\begin{array}{c}\Phi, v_1{:}k_1 \gg \sigma_1\ \mathit{type} \quad \Phi, v_1{:}k_1, v_2{:}k_2 \gg \sigma_2\ \mathit{type}\\ \Phi, v_1{:}k_1 \gg u : k_2 \quad \Phi, v_1{:}k_1, x{:}\sigma_1 \gg e : [u/v_2]\sigma_2\end{array}}{\begin{array}{r}\Phi \gg (\lambda s{:}[v_1{:}k_1, \sigma_1].[\mathit{Fst}\, s, \mathit{Snd}\, s/v_1, x][u, e]) = [(\lambda v_1{:}k_1.u), (\Lambda v_1{:}k_1.\lambda x{:}\sigma_1.e)] :\\ [v{:}k_1 \to k_2, (\forall v_1{:}k_1.\sigma_1 \to [v\, v_1/v_2]\sigma_2)]\end{array}}$$

$$(\Pi\ E >)\qquad \frac{\begin{array}{l}\Phi, v_1{:}k_1 \gg \sigma_1\ \mathit{type} \quad \Phi, v_1{:}k_1, v_2{:}k_2 \gg \sigma_2\ \mathit{type}\\ \Phi \gg u_1 : k_1 \quad \Phi \gg e_1 : [u_1/v_1]\sigma_1\\ \Phi \gg u : k_1 \to k_2 \quad \Phi \gg e : (\forall v_1{:}k_1.\sigma_1 \to [v\, v_1/v_2]\sigma_2)\end{array}}{\Phi \gg [u, e]\,[u_1, e_1] = [u\, u_1, e[u_1]\, e_1] : [v_2{:}k_2, [u_1/v_1]\sigma_2]}$$