Towards a Mechanized Metatheory of Standard ML *

Daniel K. Lee   Karl Crary   Robert Harper

Carnegie Mellon University

{dklee,crary,rwh}@cs.cmu.edu

Abstract

We present an internal language with equivalent expressive power to Standard ML, and discuss its formalization in LF and the machine-checked verification of its type safety in Twelf. The internal language is intended to serve as the target of elaboration in an elaborative semantics for Standard ML in the style of Harper and Stone. Therefore, it includes all the programming mechanisms necessary to implement Standard ML, including translucent modules, abstraction, polymorphism, higher kinds, references, exceptions, recursive types, and recursive functions. Our successful formalization of the proof involved a careful interplay between the precise formulations of the various mechanisms, and required the invention of new representation and proof techniques of general interest.

Categories and Subject Descriptors   D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics; Syntax; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Mechanical verification

General Terms   Languages, Verification

Keywords   Standard ML, language definitions, type safety, mechanized metatheory, logical frameworks, Twelf

1. Introduction

A formal definition of a programming language provides a rigorous, implementation-independent description of the semantics of well-formed programs. By giving a precise meaning to programs a formal definition provides the foundation for building a community of users, for ensuring compatibility of implementations, and for proving properties of the language and programs written in it. But a formal definition does not stand on its own, but must be supported by a body of metatheory that establishes both its internal consistency and coherence with external expectations.

The formal definition of a full-scale programming language can easily run into hundreds of pages, as exemplified by The Definition of Standard ML [21]. Verifying the metatheory of such a language taxes, or even exceeds, human capabilities. Absent complete verification, the best alternative is to employ well-established methods, such as type systems and operational semantics, supported by

small case-studies that expose pitfalls. But even using these best practices, errors and inconsistencies arise that are not easily discovered. Moreover, as languages evolve, so must the metatheory that supports it, introducing further opportunities for error.

A promising approach to reducing error is to use mechanized verification tools to ease the burden of proving properties of language definitions. Ideally, a language definition would come equipped with a body of metatheory that is mechanically checked against the definition and that can be extended as need and interest demands. With the development of powerful tools such as mechanical theorem provers and logical frameworks, it is becoming feasible to put this idea into practice. For example, Klein and Nipkow [17] have recently used the Isabelle theorem prover [23] to formalize a large part of the Java programming language and to prove type safety for it.

In this paper we report on the use of the Twelf implementation [27] of the LF logical framework [12] to verify the type safety of the full Standard ML programming language. To our knowledge this is the first mechanical verification of safety for a language of this scale. The first mechanical formalizations of significant subsets of The Definition of Standard ML were performed independently by Syme [36] and VanInwegen and Gunter [39] using HOL [10] for the purpose of establishing determinicity of evaluation. An attempt by VanInwegen [38] to prove type safety was partially successful, but ran into difficulties with the formalism of The Definition of Standard ML, the immaturity of verification tools and methodology at that time, and the unsoundness of the language itself. Our approach draws on intervening experience with logical frameworks [12, 27] and with formalizing language definitions using type-theoretic techniques [16]. Perhaps the most significant lesson to be drawn from VanInwegen's and our experience is that language definitions must be formulated with mechanical verification of metatheory in mind. The formulation of the definition provides the framework for verification, but the demands of verification must also be permitted to influence the definition. Just as programs ought to be written in conjunction with proofs of their key properties, so too must language definitions be developed hand-in-hand with their verification.

2. Overview

Our approach is based on the type-theoretic definition of Standard ML given by Harper and Stone [16]. The Harper-Stone semantics divides the definition of the language into two aspects:

1. *Elaboration*, which translates the *external language*, the abstract syntax of Standard ML, into the *internal language*, a well-behaved type theory based on the translucent sums formalism of Harper and Lillibridge [14]. Elaboration performs type reconstruction, overloading resolution, equality compilation, pattern compilation, and coercive signature matching, resulting in a well-typed term of the internal language.

---

2. *Typing and evaluation*, which enforces type constraints and imposes an operational interpretation on programs. The internal language is a well-behaved, explicitly typed λ-calculus equipped with a transition-based operational semantics on states of an abstract machine.

We will refer to the general structure of definition-by-elaboration as *elaborative semantics,* and to the Harper-Stone semantics for Standard ML in particular as H-S. Note that the utility of elaborative semantics is not limited to ML; it has also been used by Drossopoulou, *et al.* [8, 9] in rigorous accounts of the semantics of Java. The use of elaborative semantics follows in the spirit of Reynolds [32] which explored the semantics of an idealized language that embodied the "essence" of Algol. Elaborative semantics goes further, in that the relationship between a language and its idealized counterpart is made explicit by a formal translation.

In the context of Standard ML, the H-S semantics has been used successfully as the foundation for the TILT compiler for Standard ML [37, 25, 34, 24], and as a basis for language extensions such as recursive modules [5] and modular type classes [7]. For the present purposes, as well as that of the cited work, the utility of elaborative semantics accrues from the isolation of an internal language to which operational meaning is assigned using standard techniques; the external language is given meaning only via its elaboration into internal form.

This permits one to formalize type safety for the internal language as the conjunction of *preservation* (well-formed states transition only to well-formed states) and *progress* (well-formed states are either final or can make a transition). Type safety for the external language then follows from the fact that elaboration yields a well-typed program in the internal language. In the present paper we outline the proof and mechanization of the type safety of the Standard ML internal language, leaving the formalization of elaboration and the proof of well-typing of elaboration to future work. The internal language used is similar enough to the H-S internal language that we believe the elaboration relation given in H-S can be adapted for our IL without serious difficulties. In fact, the safety proof of the internal language represents the bulk of the effort, since the internal language has the same expressive power as Standard ML, lacking only conveniences such as type inference and pattern matching, which can be handled separately. However, we will discuss elaboration, where appropriate, to motivate the internal language's design.

It is natural to ask whether one could carry out a similar mechanization based on The Definition of Standard ML. Our results have no direct bearing on this question, but it may be informative to explain why we did not choose this approach. First, we had in mind VanInwegen's earlier attempt, which was not entirely successful due in part to errors in The Definition itself (the language she studied was not type safe), and in part to complications in handling the machinery used in The Definition (a considerable array of *ad hoc* semantic objects such as finite maps, generative stamps, realizations, and so forth). Second, we drew on our previous experience with the H-S semantics in the implementation of the TILT compiler and in studies of language extensions that supported our belief in the utility of type-theoretic methods in language design. Third, we intended to make use of the Twelf implementation of the LF logical framework for our work, which provides strong support for formalizing and proving properties of type-theoretic languages with purely syntactic notions of binding and scope. The machinery of The Definition is not amenable to formalization in Twelf—more precisely, using such machinery would obviate the advantages afforded by Twelf. Fourth, The Definition is based on an evaluation semantics, which does not support a direct proof of type safety. Instead, one must extend the semantics with spurious "wrong" transitions, only to prove that they are, indeed, spurious. Finally, The Definition relies on a number of informal conventions, such as the handling of overloading and the dynamic significance of signature matching, whose formalization would, we suspect, require a layer of elaboration if fully developed. In short, the structure of The Definition poses problems in general for proving safety, and in particular for proving safety in mechanized form.

Present Work In our present work, we do not utilize the H-S semantics per se, but rather a variation of it that was informed by subsequent work on the type theory of modularity and by our first, failed attempt at verification using H-S verbatim. The internal language of H-S is an extension of Harper and Lillibridge's translucent sums formalism [14] with support for recursive types, mutable references, extensible datatypes, and exceptions. That language is endowed with an operational semantics given as a transition system for an abstract machine with an explicit control stack, as well as a store for reference cells and generated tags. Our first attempts at verification, conducted by the second and third authors with Michael Philip Ashley-Rollman, uncovered numerous minor flaws in the H-S internal language, as would be expected in a formal verification effort, but also ran into apparently insurmountable technical difficulties with proving type safety for the corrected language.[1]

Our effort is based on Dreyer's variant [5] of the H-S semantics. Dreyer's internal language (and ours) is based on Stone and Harper's *singleton kind* formalism [35], rather than translucent sums, and employs phase separation to propagate type information properly while respecting the phase distinction [15] in a language with dependently typed modules. (In addition it treats data abstraction as a computational effect [6], which would permit supporting applicative functors [20], which we do not consider here.) Moreover, we formulate the operational semantics as a transition system using Plotkin's method of structured operational semantics [31]. This supports a direct statement of type safety, and avoids the complications that arose in our initial attempt using the H-S semantics.

The verification of safety for the internal language presents a number of challenges. These may be divided into two broad categories: (1) mathematical challenges, and (2) formalization challenges. The mathematical challenges are those posed by the proof, independently of any mechanization effort. These include delicate matters of staging the key technical lemmas, such as functionality and validity, so that they may be proved in logical progression. These challenges are generally typical of those arising in the metatheories of module calculi.

More novel are the challenges arising from the demands of formalization of our language in LF and of verifying its metatheory using the Twelf theorem prover. Here our previous experience with logical frameworks was important. Our internal language is carefully formulated to ensure straightforward representation in LF using higher-order abstract syntax and judgements-as-types [12], to full advantage. Using these methods ensures that we avoid the complications encountered by VanInwegen in the treatment of binding operators, since the machinery of binding and scope is provided "for free" by LF. For example, as dicussed in Section 3, Dreyer correlated module variables with underlying type constructor variables using a naming convention that is untenable in higher-order abstract syntax. In our work we maintained the correlation using a hypothetical judgement. Not only does this resolve the issue, but it also results in a more elegant language.

---

[1] Space limitations prevent detailing these difficulties here, but they can be traced to a conflict between the definition of well-formed machine states and side conditions on some typing rules restricting certain terms to be values.

In many cases, such as the issue of variable correlation, LF formalization pushed us to a cleaner development. However, we were not always so fortunate. The formalized proof of functionality—independent of its staging with other theorems—deals with the context in a manner difficult to represent in LF. Consequently, proving functionality required the development of a new proof device for Twelf that adds a detour to the proof. This issue is discussed in Section 6.

One issue fits into both categories. In both progress and preservation, it is essential to establish inversion properties such as that no function type is equal to any non-function type, and that equal function types must have equal domains and equal codomains. Such "obvious" facts are very far from trivial to prove in languages, such as ours, with a rich notion of type equality! The Standard ML type system relies on a formulation of type equality that, among other things, propagates type sharing information across module boundaries [14, 19, 35]. The declarative formulation of type equality in the presence of singleton kinds captures these properties naturally, but at the expense of making it very difficult to verify inversion properties. On the other hand, an algorithmic formulation, as is used in an implementation, makes it easy to establish inversion properties, but very difficult to establish other key properties (including ones as apparently simple as transitivity).

Thus, a crucial ingredient in the canonical forms lemma is a proof of equivalence of the declarative and algorithmic formulations of type equality. This result has already been established by Stone and Harper [35], but their proof cannot currently be expressed in Twelf due to the limits of Twelf's relational metatheory. Consequently, it was necessary to develop a new, syntactic proof based on Watkins's technique of hereditary substitutions [40]. This issue is discussed in Section 4.3.

We begin our discussion with an informal presentation of the internal language (IL) in Section 3. For the purposes of this paper, we mean by "informal" that the presentation is given in English and mathematical notation, not in machine-checkable form. We follow with an informal (in the same sense) account of the IL's metatheory in Section 4. We discuss the IL's formalization in LF in Section 5, and the safety proof's formalization in Twelf in Section 6. We assume no familiarity with LF or Twelf until Section 5. In Sections 5 and 6 we assume familiarity with the methodology of LF encoding [12, 26, 13] and, in the latter, some elementary familiarity with the Twelf meta-logic will also be helpful.

The full Twelf code of our formalization is available at:

```
www.cs.cmu.edu/~crary/papers/2006/tslf.tgz
```

3. The Internal Language

The internal language (IL) of our semantics is an explicitly-typed λ-calculus based on Dreyer's modules formalism [5], enriched with a variety of features to encompass the full expressive power of Standard ML. One simplification compared to Dreyer's language is that we eliminate support for applicative functors, only because they are not necessary for modeling Standard ML.[2]

The IL itself is not a research contribution of this paper; nearly every construct in it appears in some form in prior work [15, 14, 19, 16, 35, 6, 5]. Consequently, our discussion here is a summary, not a thorough discussion. Our purpose is to convey the scope of the language, and to set the scene for discussion of its formalization. Full technical details are included in the companion technical report [18].

The IL is structured into two levels: (1) the core level, which consists of constructors classified by kinds, and terms classified

[2] It would not present serious difficulties to accommodate them to model other languages, such as Objective Caml.

constructors	C	::=	α	
		|	$\langle \rangle$	unit constructor
		|	$\langle C_1, C_2 \rangle$	pairs
		|	$\pi_1 C$	left projection
		|	$\pi_2 C$	right projection
		|	$\lambda \alpha{:}K.C$	abstraction
		|	$C_1\ C_2$	application
		|	Unit	unit type
		|	$C_1 \times C_2$	products
		|	$C_1 \rightarrow C_2$	functions
		|	$C_1 + C_2$	sums
		|	Ref C_1	references
		|	Tag C_1	generative tags
		|	Tagged	tagged expressions
		|	$\mu \alpha{:}\mathbf{T}.C$	recursive types
kinds	K	::=	1	unit kind
		|	$\mathbf{T}$	types
		|	$\mathcal{S}(C)$	singleton kind
		|	$\Pi \alpha{:}K.C$	dependent functions
		|	$\Sigma \alpha{:}K.C$	dependent pairs

Figure 1. Constructor and Kind Syntax

by types (constructors of kind $\mathbf{T}$) and (2) the module level, which consists of modules classified by signatures. The layers are linked by projections that extract the type and term components of a module. The language is designed to enforce the *phase distinction*, which ensures that type equality is independent of term equality by arranging that an admissible type projection can be immediately determined to be a core language type.

Type definitions and type sharing relationships are managed using singleton kinds [35], which are separable from modules, in contrast to the translucent sums formalism used in the Harper-Stone semantics. This formalism isolates the issues of type equality from the other aspects of the language, and is of independent interest from its application here. In particular, the complications related to type equality mentioned in the overview are cleanly isolated from the rest of the metatheory.

Type abstraction is managed using the effects-based techniques of Dreyer, Crary, and Harper [6] in which the imposition of abstraction is regarded as akin to a computational effect. However, in contrast to the DCH formalism, we need only the basic distinction between pure and impure modules, rather than the more sophisticated classification considered there. The reason is simply that the additional sophistication is not required for the semantics of Standard ML.

3.1 Core Language

3.1.1 Constructors and Kinds

The grammar for constructors and kinds[3] is given in Figure 1. The kind $\mathbf{T}$ classifies types, which are themselves used to classify terms.[4] Most of the types and constructors are familiar. The Tag and Tagged types are used to implement Standard ML's `exn` type [16].

The *singleton kind* [35], $\mathcal{S}(C)$, classifies the constructors that are definitionally equivalent to the constructor C. It is used to model

[3] In addition to these constructs, our formalization supports some orthogonal constructs not used by Standard ML in anticipation of future development.

[4] In the terminology of the ML type system [3] these are the *monotypes*; the *polytypes* arise in our formalism as a special case of functor signatures [16].

$$\begin{array}{lll}
\mathcal{S}_1(C) & \stackrel{\text{def}}{=} & 1 \\
\mathcal{S}_{\mathbf{T}}(C) & \stackrel{\text{def}}{=} & \mathcal{S}(C) \\
\mathcal{S}_{\mathcal{S}(D)}(C) & \stackrel{\text{def}}{=} & \mathcal{S}(C) \\
\mathcal{S}_{\Sigma\alpha:K_1.K_2}(C) & \stackrel{\text{def}}{=} & \mathcal{S}_{K_1}(\pi_1 C) \times \mathcal{S}_{[\pi_1 C/\alpha]K_2}(\pi_2 C) \\
\mathcal{S}_{\Pi\alpha:K_1.K_2}(C) & \stackrel{\text{def}}{=} & \Pi\alpha:K_1.\mathcal{S}_{K_2}(C\,\alpha)
\end{array}$$

$$\frac{\Gamma \vdash \pi_1 C : K_1 \qquad \Gamma \vdash \pi_2 C : K_2}{\Gamma \vdash C : K_1 \times K_2}$$

$$\frac{\Gamma \vdash C : \Pi\alpha:K_1.L \qquad \Gamma,\alpha:K_1 \vdash C\,\alpha : K_2 \qquad \alpha \notin \mathrm{Dom}(\Gamma)}{\Gamma \vdash C : \Pi\alpha:K_1.K_2}$$

Figure 2. Higher-Order Singletons and Extensionality

type definitions and type sharing specifications. Singletons create dependencies of kinds on constructors, so function and product kinds take dependent form, $\Pi\alpha:K_1.K_2$ and $\Sigma\alpha:K_1.K_2$, respectively.

The following judgement forms govern constructors and kinds:

- K is a well-formed kind: $\Gamma \vdash K$.

- C has kind K: $\Gamma \vdash C : K$.

- K_1 is a subkind of K_2: $\Gamma \vdash K_1 \leq K_2$.

- C_1 and C_2 are equivalent at kind K: $\Gamma \vdash C_1 \equiv C_2 : K$.

- K_1 and K_2 are equivalent kinds: $\Gamma \vdash K_1 \equiv K_2$.

The grammar of typing contexts is given in Figure 8.

Constructor equivalence is induced by $\beta\eta$ rules for application and projection, together with rules for introducing and eliminating singleton kinds. The introduction rule for singletons, called *self-ification*, assigns to each constructor C of kind $\mathbf{T}$ the singleton $\mathcal{S}(C)$; this is evidently the most precise kind for C. The elimination rule for singletons permits deduction of $\Gamma \vdash C_1 \equiv C_2 : \mathbf{T}$ from $\Gamma \vdash C_1 : \mathcal{S}(C_2)$. Consequently, constructor equivalence is context-sensitive. For example, we have $\alpha:\mathcal{S}(C) \vdash \alpha \equiv C : \mathbf{T}$ because of the kinding assumption on α.

Because of dependencies, constructor equivalence induces a non-trivial kind equivalence. In addition there is a subkinding relation that is used to "forget" type equivalences due to singletons. Subkinding contains constructor equality, and is closed under the axiom $\mathcal{S}(C) \leq \mathbf{T}$, together with the usual variance rules for product and sum kinds. Constructor formation and equivalence are closed under kind subsumption. Consequently, constructor equivalence is kind-sensitive. For example, the identity operator and the constantly C operator are equivalent at kind $\mathcal{S}(C) \to \mathbf{T}$, but not at the kind $\mathbf{T} \to \mathbf{T}$.

Although singleton kinds can be formed only over constructors of kind $\mathbf{T}$; we can lift them to higher kinds as well. In Figure 2 we give the definition of higher-order singletons, together with two *extensionality* rules that are key to introducing them. For example, if $C : \mathbf{T} \times \mathbf{T}$, then we can use the first extensionality rule to show that $C : \mathcal{S}(\pi_1 C) \times \mathcal{S}(\pi_2 C) = \mathcal{S}_{\mathbf{T} \times \mathbf{T}}(C)$.

3.1.2 Terms

The syntax for the term language is given in Figure 3. The language of terms includes tuples, variants, isomorphisms for recursive types, generative tags, and tagged values. It also includes primitives for raising and handling exceptions, tag checking, and projection of the dynamic component from a module. Terms also include

locations	ℓ	$::=$	$\cdots$	
terms	e	$::=$	x	variables
		$\mid$	$\langle\rangle$	unit term
		$\mid$	$\langle e_1, e_2 \rangle$	pairs
		$\mid$	$\pi_1 e$	left projection
		$\mid$	$\pi_2 e$	right projection
		$\mid$	$\mathbf{fun}\ x\ (y{:}C_1){:}C_2.e$	recursive function
		$\mid$	$e_1\ e_2$	application
		$\mid$	$\mathbf{inl}_C e$	sum intro
		$\mid$	$\mathbf{inr}_C e$	sum intro
		$\mid$	$\mathbf{case}(e_1, x.e_2, y.e_3)$	case
		$\mid$	$\mathbf{loc}\ \ell$	locations
		$\mid$	$\mathbf{ref}\ e$	new reference
		$\mid$	$!\,e$	dereference
		$\mid$	$e_1 := e_2$	assignment
		$\mid$	$\mathbf{tagloc}\ \ell$	tag literals
		$\mid$	$\mathbf{newtag}_C$	new tag
		$\mid$	$\mathbf{tag}(e_1, e_2)$	tag injection
		$\mid$	$\mathbf{iftagof}$	
			$(e_1, e_2, x.e_3, e_4)$	tag check
		$\mid$	$\mathbf{raise}\ e$	raise exception
		$\mid$	$\mathbf{try}(e_1, x.e_2)$	try/handle
		$\mid$	$\mathbf{roll}_C e$	recursive type intro
		$\mid$	$\mathbf{unroll}\ e$	recursive type elim
		$\mid$	$\mathbf{snd}(M)$	module projection

Figure 3. Term Syntax

signatures	K	$::=$	1	unit signature
		$\mid$	$[\![C]\!]$	constructor signature
		$\mid$	$[\![K]\!]$	kind signature
		$\mid$	$\Pi\alpha{:}\sigma_1.\sigma_2$	dependent functions
		$\mid$	$\Sigma\alpha{:}\sigma_1.\sigma_2$	dependent pairs

Figure 4. Signature Syntax

locations (reference literals) and tag literals, which arise during execution but not in user programs.

The typing judgment for terms is written as $\Gamma; \Phi \vdash e : C$, where Γ is a context and Φ is a store typing that assigns types to locations and tags (Figure 8).

3.2 Module Level

The chief technical novelty of the IL's module level (due to Dreyer [5]) are two projection operations, Fst and snd. The former is a judgement that extracts the constructor portion of a (pure) module. The latter is a dynamic operation that projects the term portion of an (appropriately-typed) module. It is important to note that while snd is an ordinary operation, Fst is used only by the semantics, and cannot appear in the syntax of programs. In addition, the static semantics has a meta-operation on signatures (mirroring Fst, and also written Fst) that extracts the kind portion of a signature. This too cannot appear in the syntax of programs.

3.2.1 Signatures

The syntax of signatures is given in Figure 4. There are three basic forms of signature, 1 for the trivial signature, $[\![K]\!]$ for the signature of modules containing a single constructor of kind K, and $[\![C]\!]$ for the signature of modules containing a single term of type C. Signatures are closed under formation of dependent func-

$$\begin{array}{lcl} \mathsf{Fst}(1) & \stackrel{\mathrm{def}}{=} & 1 \\ \mathsf{Fst}(\llbracket C \rrbracket) & \stackrel{\mathrm{def}}{=} & 1 \\ \mathsf{Fst}(\llbracket K \rrbracket) & \stackrel{\mathrm{def}}{=} & K \\ \mathsf{Fst}(\Pi\alpha{:}\sigma_1.\sigma_2) & \stackrel{\mathrm{def}}{=} & 1 \\ \mathsf{Fst}(\Sigma\alpha{:}\sigma_1.\sigma_2) & \stackrel{\mathrm{def}}{=} & \Sigma\alpha{:}\mathsf{Fst}(\sigma_1).\mathsf{Fst}(\sigma_2) \end{array}$$

Figure 5. Static Part of a Signature (Fst)

tions, $\Pi\alpha{:}\sigma_1.\sigma_2$, and dependent products, $\Sigma\alpha{:}\sigma_1.\sigma_2$. These are used to represent functors and sub-structures in Standard ML, respectively. Consequently, functor signatures are partial (generative) in that applications are regarded as impure. (More on this below.)

The astute reader will have noticed that the function and product signatures bind constructor variables, rather than module variables. This is a reflection of the *phase distinction* in Standard ML, which precludes constructors that depend on terms, even indirectly via modules. In general modules consist of a *static part*, which is a type constructor, and a *dynamic part*, which is a term. The syntax expresses the fact that the result signature of a functor can depend only on the static part of its arguments, and similarly that a module in the scope of a sub-module can only depend on the static part of the sub-module.

Only certain forms of module—the *projectible* modules—have a static part that can be determined during type checking. (The class of projectible modules will be discussed in more detail shortly.) When the static part exists, it is a constructor whose kind is determined by the module's signature. The meta-level operation $\mathsf{Fst}(\sigma)$ defined in Figure 5 determines the kind of the static part of a module of signature σ. Hence the rule:

$$\frac{\Gamma \vdash \sigma_1 \quad \Gamma, \alpha{:}\mathsf{Fst}(\sigma_1) \vdash \sigma_2}{\Gamma \vdash \Pi\alpha{:}\sigma_1.\sigma_2}$$

Note that the static part of a functor signature is trivial. This is a reflection of the fact that Standard ML functors are generative, and hence do not have a statically computable static part. (Were we to support pure, or applicative, functors, their static parts would have functional kinds.)

The judgement forms governing signatures are as follows:

- σ is a well-formed signature: $\Gamma \vdash \sigma$.

- σ_1 and σ_2 are equivalent signatures: $\Gamma \vdash \sigma_1 \equiv \sigma_2$.

- σ_1 is a subsignature of σ_2: $\Gamma \vdash \sigma_1 \leq \sigma_2$.

Signature equivalence is induced by constructor and kind equivalence. The subsignature relation is induced by the subkind relation in the usual fashion, except that our functor signatures are invariant, rather than contra- and covariant on the left and right. This reflects from the fact that our intended elaborator [16] employs explicit coercions at functor applications,[5] so the IL does not require subsignatures at those points. It would not be difficult to use the standard rule instead, were the IL to be used with a different elaborator.

3.2.2 Modules

The syntax of modules, given in Figure 6, consists of introductory and eliminatory forms for each form of signature, plus two additional constructs that we shall describe shortly. The introductory

---

[5] The reason for this, in turn, is because much more can happen in a Standard ML signature coercion than just dropping of type information. For example, dynamic fields may be dropped and polymorphic functions specialized.

modules	M	::=	s	
			$\langle\rangle$	unit module
			$[e]$	term module
			$[C]$	constructor module
			$\langle M_1, M_2 \rangle$	pairs
			$\pi_1 M$	left projection
			$\pi_2 M$	right projection
			$\lambda(s/\alpha_s{:}\sigma_1){:}{>}\sigma_2.M$	functor
			$M_1\ M_2$	application
			$M :> \sigma$	sealing
			$\mathtt{let}\ s/\alpha_s = M_1$	
			$\mathtt{in}(M_2 :> \sigma)$	let binding

Figure 6. Module Syntax

forms for atomic signatures are $[C]$ for $\llbracket K \rrbracket$, and $[e]$ for $\llbracket C \rrbracket$. Instead of an elimination form for $\llbracket C \rrbracket$, the static semantics use the aforementioned Fst. Atomic term modules $\llbracket C \rrbracket$ are eliminated using snd.

Modules of product signature are introduced by pairing, and eliminated by projection, and modules of function signature are introduced by λ-abstraction, and eliminated by application. The syntax of λ-abstraction is unusual in that it introduces *two* variables, s and α_s, standing for the argument module itself, and its static part. The variables s and α_s are "twinned" in the sense that there is an implicit correlation between them, even in the face of alpha-conversion. (As we shall see in Section 5, this is represented in LF by an explicit judgement form.) Typing contexts are extended accordingly with a declaration of the form $s/\alpha_s : \sigma$.

There are two additional constructs, sealing a module with a signature to impose abstraction, written $M :> \sigma$, and a form of let-binding for modules whose syntax is reminiscent of that of λ-abstractions. (The explicit signature information is necessary to circumvent the avoidance problem [6].)

Abstraction is enforced using an effects system that classifies modules into two categories, *pure* (or *projectible*), and *impure* (or *non-projectible*). Sealed modules are impure, and, since functors are generative, so are all functor applications. On the other hand module variables are pure, as are pairs of pure modules.[6]

As the name suggests, projectible modules permit projection of type components. Moreover, if a module expression, M, of signature σ is projectible, then $\mathsf{Fst}(M)$ is a constructor of kind $\mathsf{Fst}(\sigma)$ representing the static part of M. Just as for signatures, $\mathsf{Fst}(M)$ is a meta-level operation, rather than a constructor-forming construct of the language. Its definition is given in Figure 7; in the case of a module variable, s, we define $\mathsf{Fst}(s)$ to be the correlated constructor variable, α_s. Note that while snd may be used only with modules having signature $\llbracket C \rrbracket$, Fst may be used with any projectible module.

The meta-variable κ stands for the purity (projectibility) class of a module, either P or I, according to whether the module is pure or impure. The typing judgment for modules is $\Gamma; \Phi \vdash M :_\kappa \sigma$, where κ is a projectibility class. Projectibility classes are ordered by $\mathtt{P} \sqsubseteq \mathtt{I}$, so that every pure module may be regarded as (vacuously) impure.

3.3 States and Stores

As Standard ML is an imperative language, terms must be considered along with a store that assigns meanings to locations. Loca-

---

[6] Lambda abstractions may be considered pure, but this turns out to be of no importance: since Standard ML functors are first-order and generative, there is no way to use them without incurring an effect.

$$\frac{s/\alpha_s{:}\sigma \in \Gamma}{\Gamma \vdash \mathsf{Fst}(s) \gg \alpha_s} \qquad \overline{\Gamma \vdash \mathsf{Fst}(\langle\rangle) \gg \langle\rangle}$$

$$\overline{\Gamma \vdash \mathsf{Fst}([e]) \gg \langle\rangle} \qquad \overline{\Gamma \vdash \mathsf{Fst}([C]) \gg C}$$

$$\frac{\Gamma \vdash \mathsf{Fst}(M_1) \gg C_1 \quad \Gamma \vdash \mathsf{Fst}(M_2) \gg C_2}{\Gamma \vdash \mathsf{Fst}(\langle M_1, M_2 \rangle) \gg \langle C_1, C_2 \rangle}$$

$$\frac{\Gamma \vdash \mathsf{Fst}(M) \gg C}{\Gamma \vdash \mathsf{Fst}(\pi_1 M) \gg \pi_1 C} \qquad \frac{\Gamma \vdash \mathsf{Fst}(M) \gg C}{\Gamma \vdash \mathsf{Fst}(\pi_2 M) \gg \pi_2 C}$$

$$\overline{\Gamma \vdash \mathsf{Fst}(\lambda(s/\alpha_s{:}\sigma_1){:}{>}\sigma_2.M) \gg \langle\rangle}$$

Figure 7. Static Part of a Projectible Module (Fst)

contexts	Γ	$::=$	$\cdot$	
		$\mid$	$\Gamma, \alpha{:}K$	constructor binding
		$\mid$	$\Gamma, x{:}C$	value binding
		$\mid$	$\Gamma, s/\alpha_s{:}\sigma$	module binding
heap types	Υ	$::=$	$\cdot$	
		$\mid$	$\Upsilon, \ell{:}C$	reference type
tag typings	Θ	$::=$	$\cdot$	
		$\mid$	$\Theta, \ell{:}C$	tag type
store types	Φ	$::=$	(Υ, Θ)	

Figure 8. Contexts and Store Typings

tions are used in two ways in our IL: to identify reference cells and to identify tags (we use the latter to implement `exn` constructors in Standard ML). When used to identify a reference cell, a location is mapped to the value contained by that cell, and when used to identify a tag, a location is mapped to the type expected by that tag. Thus, stores have two components, a *heap* serving the former purpose, and a *tag typing* serving the latter. A state in the IL's abstract machine consists of a term or module being evaluated, and an accompanying store. (The syntax for states and stores is given in Figures 8 and 9.)

Heaps are classified by heap types and stores by store types (Figure 8). Tag typings do not require a classifier; we merely require that they be well-formed, which is the case when each tag type in it is well-formed with kind **T**. Thus, the judgement forms governing states and stores are as follows:

- Θ is a well-formed tag typing: $\vdash \Theta$.
- H has type Υ: $\Phi \vdash H : \Upsilon$.
- S has type Φ: $\Phi \vdash S : \Phi$.
- ς is a well-formed state: $\vdash \varsigma$.

A state (e, S) is well-formed if there exists some store type Φ classifying S, and some type C, such that $\cdot; \Phi \vdash e : C$:

$$\frac{\Phi \vdash S : \Phi \quad \cdot; \Phi \vdash e : C}{\vdash (e, S)}$$

A similar rule exists for typing states in which a module is being evaluated.

heaps	H	$::=$	$\cdot$	
		$\mid$	$H, \ell {\rightsquigarrow} e$	
stores	S	$::=$	(H, Θ)	
states	ς	$::=$	(e, S)	
		$\mid$	(M, S)	

Figure 9. States and Stores

3.4 Dynamic Semantics

The dynamic semantics of the core level involves three main judgement forms:

- Expression e_1 is a value: e_1 `val`.
- Expression e_1 raises the exception e_2: $e_1 \nearrow e_2$.
- Transition between states: $(e_1, S_1) \mapsto (e_2, S_2)$.

These are defined simultaneously with three analogous judgements at the module level.

Our progress theorem proves that for any well typed state (e_1, S_1), one of these three judgements will hold. Note that values are defined using an explicit predicate `val`. Although it is somewhat more common to define values using a sub-syntactic grammar, to do so requires subtyping on syntactic classes, which is not supported by LF.

4. Informal Metatheory

We prove the type safety theorem for our IL in the usual manner. In that proof, complications arise due to dependent typing of constructors, due to stores, due to the interaction of evaluation with type abstraction in modules, and most significantly due to singleton kinds. We outline here the most interesting lemmas on the path to proving type safety.

4.1 Preliminaries

One baseline property that is used throughout the metatheory is *validity* which states that the static semantics deals only with well-formed entities. For example, if C_1 is equal to C_2 at the kind K, then C_1 and C_2 individually have the kind K, and K itself is well-formed. A direct proof of validity is stymied by the asymmetry in some definitional equality rules. For example:

$$\frac{\Gamma \vdash C_1 \equiv C_1' : \Pi\alpha{:}K_1.K_2 \quad \Gamma \vdash C_2 \equiv C_2' : K_1}{\Gamma \vdash C_1 C_2 \equiv C_1' C_2' : [C_2/\alpha]K_2}$$

That $C_1 C_2$ has kind $[C_2/\alpha]K_2$ follows immediately by induction and the appropriate formation rule, but the same is not true for $C_1' C_2'$. For the latter, we need the additional fact that $[C_2'/\alpha]K_2$ is equal to $[C_2/\alpha]K_2$.

Induction and inversion on kind formation gives us that $\Gamma, \alpha{:}K_1 \vdash K_2$, so the desired fact appears to follow from a functionality principle. However, to use this principle at this stage, we must carefully state it so that it can be proven before validity.

Lemma 4.1 (Functionality)**.**

1. *If* $\Gamma, \alpha{:}K \vdash K'$,
   and $\Gamma \vdash C_1 \equiv C_2 : K$,
   and $\Gamma \vdash C_1 : K$,
   and $\Gamma \vdash C_2 : K$,
   then $\Gamma \vdash [C_1/\alpha]K' \equiv [C_2/\alpha]K'$.
2. *If* $\Gamma, \alpha{:}K \vdash C' : K'$,
   and $\Gamma \vdash C_1 \equiv C_2 : K$,
   and $\Gamma \vdash C_1 : K$,

and $\Gamma \vdash C_2 : K,$
then $\Gamma \vdash [C_1/\alpha]C' \equiv [C_2/\alpha]C' : [C_1/\alpha]K'.$

The third and fourth premises ($\Gamma \vdash C_1, C_2 : K$) can be dropped, but only after validity has been established:

Lemma 4.2 (Validity for Constructors and Kinds)**.**

1. *If* $\Gamma \vdash C : K$, *then* $\Gamma \vdash K$.
2. *If* $\Gamma \vdash C_1 \equiv C_2 : K$,
   then $\Gamma \vdash C_1 : K$, *and* $\Gamma \vdash C_2 : K$, *and* $\Gamma \vdash K$.
3. *If* $\Gamma \vdash K_1 \equiv K_2$, *then* $\Gamma \vdash K_1$ *and* $\Gamma \vdash K_2$.
4. *If* $\Gamma \vdash K_1 \leq K_2$, *then* $\Gamma \vdash K_1$ *and* $\Gamma \vdash K_2$.

In our preservation theorem, it is necessary to know that typing of terms and modules is not disturbed by the allocation of new references or tags. Thus, we require a lemma that expresses weakening with respect to heap types:

Lemma 4.3 (Weakening with Respect to Heap Types)**.**

1. *If* $\Gamma; (\Upsilon, \Theta) \vdash e : C$ *and* $\ell \notin Dom(\Upsilon)$,
   then $\Gamma; ((\Upsilon, \ell:C'), \Theta) \vdash e : C$.
2. *If* $\Gamma; (\Upsilon, \Theta) \vdash M :_\kappa \sigma$ *and* $\ell \notin Dom(\Upsilon)$,
   then $\Gamma; ((\Upsilon, \ell:C'), \Theta) \vdash M :_\kappa \sigma$.

A similar lemma expresses weakening with respect to tag typings. As an aside, although weakening with respect to the context comes for free with the formalization in LF, weakening with respect to these store type constituents requires explicit proof. This is because, as we will see, our formalization treats the store type not as a context handled implicitly by LF, but as an explicit argument to the term and module formation judgement.

4.2 Modules

As usual, the preservation theorem requires substitution lemmas for term and module values. Term substitution is standard, but the statement of module substitution must account for Fst:

Lemma 4.4 (Module Substitution)**.**

1. *If* $\Gamma, s/\alpha_s{:}\sigma; \Phi \vdash e' : C'$,
   and $\Gamma; \Phi \vdash M :_P \sigma$,
   and $\Gamma \vdash \mathsf{Fst}(M) \gg C$,
   then $\Gamma; \Phi \vdash [C/\alpha_s][M/s]e' : [C/\alpha_s]C'$.
2. *If* $\Gamma, s/\alpha_s{:}\sigma; \Phi \vdash M' :_\kappa \sigma'$,
   and $\Gamma; \Phi \vdash M :_P \sigma$,
   and $\Gamma \vdash \mathsf{Fst}(M) \gg C$,
   then $\Gamma; \Phi \vdash [C/\alpha_s][M/s]M' :_\kappa [C/\alpha_s]\sigma'$.

Note that module substitution requires that the substitutend be pure, so its constructor component can be extracted. Since the let-binding construct permits the module being bound to be impure,[7] we require a lemma saying that module values are necessarily pure. In other words, by the time we are ready to substitute a module, it is permissible to do so, because all of its effects have been resolved.

Lemma 4.5 (Module Values are Pure)**.** *If* M `val` *and* $\cdot; \Phi \vdash M :_\kappa \sigma$, *then* $\cdot; \Phi \vdash M :_P \sigma$.

Finally, since the module language is dependently typed, an issue typical to dependently typed systems arises in the cases of the preservation theorem pertaining to application or to projection from a pair. Since the signature of $F\,M$ depends on Fst of M (and similarly for $\pi_2 M$), when M steps to M', we need to show that their constructor portions are equal.

---

[7] This is a vital feature; without it, abstract types (the type components of impure modules) can never be used.

Lemma 4.6 (Evaluation Preserves Fst)**.**
If $(M, S) \mapsto (M', S')$,
and $\cdot; \Phi \vdash M :_P \sigma$,
and $\cdot \vdash \mathsf{Fst}(M) \gg C$,
then $\cdot \vdash \mathsf{Fst}(M') \gg C'$ *and* $\cdot \vdash C \equiv C' : \mathsf{Fst}(\sigma)$.

4.3 Inversion

As is the case for any language supporting a non-trivial notion of definitional type equality, our safety proof requires a number of inversion properties such as that no function type is equal to any non-function type, and that equal function types have equal domains and equal codomains.

For example, the canonical forms lemma (the key lemma for progress), states that any value e with type $C_1 \rightarrow C_2$ must have the form `fun` $x(y{:}C_1'){:}C_2'.e'$. This is not difficult to prove, provided one can rule out the case that e is actually (say) a pair, which is placed into the function type by virtue of $C_1 \times C_2 \equiv C_1 \rightarrow C_2$.

Similarly, in preservation, when considering the case of the beta-reduction of a function application, we need to know that if `fun` $x(y{:}C_1){:}C_2.e$ has type $C_1' \rightarrow C_2'$, then $C_1 \equiv C_1'$ and $C_2 \equiv C_2'$. Again, this is not difficult to show, provided one may employ inversion on the equality of function types.

Quite a number of such inversion results are required (quadratic in the number of type primitives, which is eight). A few typical instances are as follows:

Lemma 4.7. *Inversion: Contradiction*

1. *It is not the case that* $\Gamma \vdash \mathsf{Unit} \equiv C_1 \times C_2 : \mathbf{T}$.
2. *It is not the case that* $\Gamma \vdash \mathsf{Unit} \equiv C_1 \rightarrow C_2 : \mathbf{T}$.
3. *It is not the case that* $\Gamma \vdash C_1 \times C_2 \equiv C_1' \rightarrow C_2' : \mathbf{T}$.
4. *And so forth.*

Lemma 4.8. *Inversion: Injectivity*

1. *If* $\Gamma \vdash C_1 \times C_2 \equiv C_1' \times C_2' : \mathbf{T}$,
   then $\Gamma \vdash C_1 \equiv C_1' : \mathbf{T}$ *and* $\Gamma \vdash C_2 \equiv C_2' : \mathbf{T}$.
2. *If* $\Gamma \vdash C_1 \rightarrow C_2 \equiv C_1' \rightarrow C_2' : \mathbf{T}$,
   then $\Gamma \vdash C_1 \equiv C_1' : \mathbf{T}$ *and* $\Gamma \vdash C_2 \equiv C_2' : \mathbf{T}$.
3. *And so forth.*

Since our IL includes a transitivity rule over constructors, we cannot obtain any of these results by direct inductive proof. We require some strategy for taming the complexity of definitional equality. The solution is to employ an equivalent but syntax-directed presentation of type equality. We will refer to this alternative presentation as *algorithmic equality*. It is not at all difficult to show that functional types are never algorithmically equal to non-functional types, and that two function types are algorithmically equal exactly when they have equal domains and equal codomains. What remains is to show that algorithmic equality coincides with definitional equality.

Thus far, the story is typical of languages with non-trivial notions of type equality. For our IL, definitional equality is resolutely context sensitive (recall Section 3.1.1) making it far from clear how to use a typical reduction-based account [1, 29]. Nevertheless, a satisfactory algorithm was devised and proven equivalent to definitional equality by Stone and Harper [35].

Unfortunately, while Stone and Harper's algorithm would be satisfactory for our purposes, their proof is not, because it cannot currently be formalized in Twelf. (We discuss why not in Section 6.4.) Consequently, we found it necessary to develop an entirely new proof, based on a somewhat different algorithm. Space considerations preclude a complete discussion of the new proof here. Instead, we summarize the salient points.

The proof is based on Watkins's technique of hereditary substitutions [40]. It in we formulate a canonical presentation of the

singleton kind calculus. (By the singleton kinds calculus, we mean the kind and constructor portions of the IL. For the purposes of this proof, the remainder of the IL can be neglected.[8]) The canonical presentation requires that constructors must be written in canonical form.[9] The key property of the canonical presentation is that a constructor can be written in only one way, up to alpha-equivalence. In other words, definitional equality is identity.

Substitution cannot be defined in the usual manner in the canonical presentation, since it could produce non-canonical constructors. Thus, a notion of *hereditary substitution* is defined that simultaneously substitutes and re-canonizes. It is not at all obvious that any of the usual properties of substitution hold for hereditary substitution, so those properties must be established. Measuring coarsely by lines of Twelf code, this is about half the work.

Once the properties of substitution are established, we may show, by induction on derivations, that equivalence classes of constructors in the singleton kind calculus can be mapped to a unique canonical form. This immediately provides an equivalence algorithm and its completeness proof. For soundness, it remains to show that a constructor is equal to its canonical form, which can be proven by induction over the structure of the canonical form.

4.4 Type Safety

Once inversion is established, we can prove the canonical forms lemma:

Lemma 4.9 (Canonical Forms)**.**

1. *If e* val *and* $\cdot; \Phi \vdash e : C_1 \times C_2$, *then e has the form* $\langle e_1, e_2 \rangle$.
2. *If e* val *and* $\cdot; \Phi \vdash e : C_1 \rightarrow C_2$, *then e has the form* $\mathtt{fun} f(x{:}C'){:}C''.e'$.
3. *If e* val *and* $\cdot; \Phi \vdash e : \mathsf{Ref}\ C$, *then e has the form* $\mathtt{loc}\ \ell$.
4. *And so forth.*

We now have everything we need to prove progress and preservation.

Theorem 4.10 (Progress)**.**

1. *If* $\vdash (e, S)$ *then either:*
   - *e* val*, or*
   - *$e \nearrow e'$, for some e', or*
   - *$(e, S) \mapsto (e', S')$, for some e', S'.*
2. *If* $\vdash (M, S)$ *then either:*
   - *M* val*, or*
   - *$M \nearrow e$, for some e, or*
   - *$(M, S) \mapsto (M', S')$, for some M', S'.*

Theorem 4.11 (Preservation)**.**

1. *If* $(e, S) \mapsto (e', S')$,
   and $\cdot; \Phi \vdash e : C$,
   and $\Phi \vdash S : \Phi$,
   then there exists some extension Φ' of Φ
   such that $\cdot; \Phi' \vdash e' : C$ *and* $\Phi' \vdash S' : \Phi'$.
2. *If* $(M, S) \mapsto (M', S')$,
   and $\cdot; \Phi \vdash M :_\kappa \sigma$,
   and $\Phi \vdash S : \Phi$,
   then there exists some extension Φ' of Φ
   such that $\cdot; \Phi' \vdash M' :_\kappa \sigma$ *and* $\Phi' \vdash S' : \Phi'$.

This concludes the informal overview of the safety proof.

---

[8] The fact that Fst is a meta-operation and not a syntactic construct is key here.

[9] Canonical forms are those that are beta-normal and eta-long, and do not include any subterms whose natural kind (in the sense of Stone and Harper [35]) is a singleton.

5. Formalization in LF

We encode the IL in LF using the usual LF methodology [12]. That is, syntax is represented using higher-order abstract syntax, and semantics using judgements-as-types. We will not reprise the basic LF methodology here; unfamiliar readers are referred to summaries by Harper, *et al.* [12], Pfenning [26], and Harper and Licata [13]. The examples use Twelf's [27] concrete syntax for LF.

To establish notation, we give the declarations of the type families (that is, the syntactic classes and judgements) in Figure 10, and the declarations of context blocks in Figure 11. Most of the formalization goes through smoothly, leaving little worth remarking on. However, two points do raise interesting issues, and each of them shed some light on the "on-paper" presentation as well. These points are (1) the use of the store type in the static semantics of terms and modules, and (2) the matter of "twinned" variables, arising in the formalization of the Fst operation on modules.

5.1 Store Types in Term and Module Typing

In the informal system, we follow standard practice and write the store type to the left of the turnstile in the term and module typing judgements. However, the store typing is not truly a context, in the sense of a collection of alpha-convertible assumptions.[10] Instead, the store typing indicates a world of discourse, relative to which the term or module is to be considered. Accordingly, while the IL context is identified with the LF context, as usual, the store type is included explicitly in the typing judgements for terms and modules. (Moreover, for technical convenience, the heap type and tag typing are actually supplied as separate parameters.)

However, an explicit store type is *not* given for term and module variable assumptions. Since, as is typical, our IL's store evolves monotonically, an assumption $x : \mathsf{Ref}\ C$, provides a usable reference value not just for the current store, but for all future stores. Put more concretely, if variable assumptions were good only for specific store types, Lemma 4.3 would not hold. In the terminology of modal logic, variable assumptions are *necessary*.

Consequently, the judgement used for terms and modules on the left (that is, for variable assumptions) is different than that used on the right. On the left we use an assumption judgement with no store type parameter:

```
assm/tm : tm -> cn -> type.    %% x : C
assm/md : md -> sg -> type.    %% s : σ
```

The assumption judgements are tied to the typing judgements by hypothesis rules:

```
oftp/var        : oftp HT TT E C
                     <- assm/tm E C.
ofsg/var        : ofsg HT TT pty/p M S
                     <- assm/md M S.
```

Note that the store type in these rules are unconstrained. This is sound because it is an invariant of the type system that whenever a value is substituted for a variable, that value is well-typed relative to the current store. This invariant is typical of type systems for ML-like languages [11, 30], but rarely is it considered explicitly.

5.2 Twinned Variables

A key technical point in the design of the IL is that the constructor and kind language may be considered independently. In particular, it makes no reference to the syntactic class of modules. However, it is clearly necessary for constructors to be able to refer to module assumptions. Harper *et al.* [15] resolved the apparent contradiction

---

[10] The account of Morrisett *et al.,* [22] suggests that perhaps it could be, but we have not attempted to adapt their approach to our work.

```
kd  : type.    %% kinds (K)
cn  : type.    %% constructors (C)
sg  : type.    %% signatures (σ)
tm  : type.    %% terms (e)
md  : type.    %% modules (M)

ht  : type.    %% heap type (Υ)
tt  : type.    %% tag typing (Θ)
st  : type.    %% heap (H)
pty : type.    %% purity level (κ = P or I)

kd-wf  : kd -> type.              %% ⊢ K
ofkd   : cn -> kd -> type.        %% ⊢ C : K
kd-deq : kd -> kd -> type.        %% ⊢ K ≡ K'
kd-sub : kd -> kd -> type.        %% ⊢ K ≤ K'
cn-deq : cn -> cn -> kd -> type.  %% ⊢ C ≡ C' : K

fst-sg : sg -> kd -> type.        %% Fst(σ) ≝ K
fst-md : md -> cn -> type.        %% ⊢ Fst(M) ≫ C

sg-wf  : sg -> type.              %% ⊢ σ
sg-deq : sg -> sg -> type.        %% ⊢ σ ≡ σ'
sg-sub : sg -> sg -> type.        %% ⊢ σ ≤ σ'

oftp   : ht -> tt -> tm -> cn
                     -> type.     %% (Υ, Θ) ⊢ e : C
ofsg   : ht -> tt -> pty
           -> md -> sg -> type.   %% (Υ, Θ) ⊢ M :κ σ

oflt   : st -> ht -> tt -> type
                     %% (Υ, Θ) ⊢ (H, Θ) : (Υ, Θ)  †

val/tm    : tm -> type.           %% e val
val/md    : md -> type.           %% M val
raises/tm : tm -> tm -> type.     %% e ↗ e'
raises/md : md -> tm -> type.     %% M ↗ e
step/tm   : tm -> st -> tt
              -> tm -> st -> tt -> type.
                     %% (e, (H, Θ)) ↦ (e', (H', Θ'))
step/md   : md -> st -> tt
              -> md -> st -> tt -> type.
                     %% (M, (H, Θ)) ↦ (M', (H', Θ'))
```

† In oflt, the heap type and tag typing are given once each, not repeated two and three times as in the informal system.

Figure 10. Encoding in LF (Type Families)

```
%block ofkd-block            %% α : K
   : some {K:kd}
       block {a:cn} {da:ofkd a K}.

%block oftp-block            %% x : C
   : some {C:cn}
       block {x:tm} {dx:assm/tm x C}.

%block ofsg-block            %% s/αs : σ
   : some {S:sg} {K:kd}
       block {s:md} {ds:assm/md s S}
             {a:cn} {da:ofkd a K}
             {dfst:fst-md s a}.
```

Figure 11. Encoding in LF (Context Blocks)

by allowing constructors to refer directly to the module *variables*. When used within a constructor, a module variable s is written s^c, and refers to the constructor portion of the module represented by s. This device was borrowed by Dreyer [5] in his type theory on which we based our IL.

This device poses a problem for our formalization, since we have no license in LF to take a variable with type md and decree that it has type cn. On the other hand, to make $-^c$ into an explicit construct by adding

```
mdToCn : md -> cn.
```

would introduce exactly the dependence of constructors on modules that we wish to avoid.[11]

Another alternative is to introduce a new type for module variables, mdVar, with one injection into modules and one into constructors.

```
mdVarToMd : mdVar -> md.
mdVarToCn : mdVar -> cn.
```

We can avoid introducing a dependence of constructors on modules by ensuring that the only objects of type mdVar are variables. However, such an approach (sometimes called "weak higher-order abstract syntax" [4]) sacrifices a key advantage of LF: the ability to define substitution simply as LF application. Rather, an explicit definition of substitution (and requisite theorems) would be required.

Instead, we decided that the binding of a module variable actually introduces two variables: the module variable itself and a constructor variable representing its Fst part. We do not maintain the connection between the two variables using a convention of spelling (as do Harper *et al.* and Dreyer), since such a spelling convention would do violence to alpha-convertibility, and is therefore incompatible with LF. Instead, as discussed earlier in Section 3.2.2, we explicitly use two binding occurrences to bind two distinct variables, each of which may be freely alpha-varied. (Signatures, which care only about a module's constructor portion, actually bind only the constructor variable.) Although this design seems obvious in retrospect, it took the pressure of formalization in LF to lead us to it.

For example, functors (module lambdas) are encoded using:

```
md/lam : sg -> (cn -> sig)
            -> (md -> cn -> md) -> md.
```

The first argument is the functor's domain, the second its codomain, which can depend on the constructor portion of the argument, and the third its body, which can depend on the argument and its constructor portion.

Once introduced, we can cleanly maintain the pairing between module and constructor variables using a hypothetical judgement. The type fst-md M C (recall Figure 10) represents the judgement ⊢ Fst(M) ≫ C, expressing the relationship between a module and its constructor portion. When we introduce twinned variables, we simply introduce a fst-md hypothesis at the same time. For example, the typing rule for functors is:

---

[11] It might be possible to use a typing rule to ensure that mdToCn was used only with module variables, but since Twelf's subordination relation would still be affected, it is unclear how much advantage we would still receive from the separation. In any case, we had a better idea.

```
ofsg/md/lam
  : ofsg HT TT pty/p
    (md/lam S1 S2 M) (sg/pi S1 S2)
    <- fst-sg S1 K1
    <- sg-wf S1
    <- ({s:md} assm/md s S1
          {a:cn} ofkd a K1
          -> fst-md s a
          -> ofsg HT TT Y (M s a) (S2 a)).
```

When typing the body of the functor, in addition to introducing the twinned variables s and a, with signature S1 and kind K1, we also introduce an assumption of type fst-md s a, indicating that a is the constructor portion of s. The resulting block of assumptions added to the context while typing the body is ofsg-block, given in Figure 11.

6. Verification with Twelf

We verified the IL's metatheory using the Twelf meta-logical framework [27]. We will not reprise the Twelf methodology here; unfamiliar readers are referred to summaries by Crary and Sarkar [2], Harper and Licata [13], and Pfenning [26] (in increasing order of detail).

Once we devised the appropriate formalization of the IL in LF, the process of verifying the metatheory within Twelf went smoothly for the most part. Many of the proofs are very similar to how the metatheory would be developed on paper. However, several interesting issues did arise, which we discuss here.

6.1 Functionality and Explicit Contexts

The functionality lemma (Lemma 4.1) expressed in Twelf takes the form:

```
funct/kd-wf : ({a:cn} ofkd a K -> kd-wf (K' a))
              -> cn-deq C1 C2 K
              -> ofkd C1 K
              -> ofkd C2 K
              %% outputs begin here
              -> (kd-deq (K' C1) (K' C2))
              -> type.
%mode funct/kd-wf +D1 +D2 +D3 +D4 -D5.  12
```

The difficulty appears with the very first argument. This argument refers to a derivation of kd-wf (K' a) that is permitted to depend on some a:cn such that ofkd a K. Furthermore, it can refer to other variables residing in the context. However, note that no variables in the context can refer to the variable a; its entire scope is the first argument. In other words, implicit in the theorem statement is the fact that a *appears last* in the context.

This is satisfactory for all our uses of the lemma, but it is not a strong enough induction hypothesis for the proof to go through. Any rule that adds a new assumption to the context will break the invariant, and such new assumptions cannot in general be moved to before a because our constructors are dependently kinded.

Consequently, it is necessary to strengthen the induction hypothesis so that that variable of interest need not appear last. To expressing the strengthened hypothesis in LF requires a novel device we call *explicit contexts*.

The idea is to formulate an alternative formalization of the IL in which contexts are made explicit, rather than identified with the LF context as usual. More precisely, we reformulate the static semantics of constructors, kinds, and signatures; terms and modules have

no bearing on the former and require no functionality lemma in their own right. Importantly, the alternative formalization changes *only* the static semantics; the syntax remains unchanged. Thus, the explicit system is talking about the same objects as the original system.

For example, constructor formation is formalized in the explicit system by the judgement:

```
eofkd : cxt -> cn -> kd -> type.   %% Γ ⊢ C : K
```

where contexts are formalized simply as lists of assumptions:

```
cxt        : type.
cxt/nil  : cxt.                    %% ·
cxt/cons : cxt -> cn -> kd -> cxt. %% Γ, α:K
```

In the explicit system one can prove functionality using a direct proof by induction, but this is not enough. We must also prove that the explicit system is equivalent to the standard (implicit context) system. (The explicit system is clunkier to work with, so we wish to use it as little as possible. We certainly do not wish to use it for our entire verification.) Consequently, for each judgement, we prove an "implication" and "explication" theorem. For example:

```
implicate-closed/ofkd
  : eofkd cxt/nil C K -> ofkd C K -> type.
%mode implicate-closed/ofkd +D1 -D2.

explicate-closed/ofkd
  : ofkd C K -> eofkd cxt/nil C K -> type.
%mode explicate-closed/ofkd +D1 -D2.
```

Note that the explication theorem always produces a derivation using an empty explicit context. (This must be so, since we do not have access to the LF context.) However, we do need to use functionality in non-empty contexts. Therefore, the explicit context system is formulated so that it can refer to the implicit (LF) context, as well as the explicit one. In practice, then, the explicit context is used for bindings accumulated during the proof, while the implicit context is used for pre-existing bindings.

6.2 Evaluation Contexts

In the course of proving preservation for modules, we need to establish a lemma regarding beta reduction of module pairs:

Lemma 6.1.

1. If $\Gamma; \Phi \vdash \pi_1\langle M_1, M_2\rangle :_\kappa \sigma_1$ then $\Gamma; \Phi \vdash M_1 :_\kappa \sigma_1$.
2. If $\Gamma; \Phi \vdash \pi_2\langle M_1, M_2\rangle :_\kappa \sigma_2$ then $\Gamma; \Phi \vdash M_2 :_\kappa \sigma_2$.

The proof of this lemma is complicated by an extensionality rule for modules (recall from Section 3.1.1 and Figure 2 a similar rule for constructors):

$$\frac{\Gamma; \Phi \vdash \pi_1 M :_{\text{P}} \sigma_1 \qquad \Gamma; \Phi \vdash \pi_2 M :_{\text{P}} \sigma_2}{\Gamma; \Phi \vdash M :_{\text{P}} \sigma_1 \times \sigma_2}$$

This rule makes it possible for the module in question to grow by the addition of projections as we consider smaller derivations. Consequently, it is necessary to strengthen the induction hypothesis:

Lemma 6.2. *Let evaluation contexts be defined by:*

$$\Psi ::= [\,] \mid \pi_1\Psi \mid \pi_2\Psi$$

Then:

1. If $\Gamma; \Phi \vdash \Psi[\pi_1\langle M_1, M_2\rangle] :_\kappa \sigma$, then $\Gamma; \Phi \vdash \Psi[M_1] :_\kappa \sigma$.
2. If $\Gamma; \Phi \vdash \Psi[\pi_2\langle M_1, M_2\rangle] :_\kappa \sigma$, then $\Gamma; \Phi \vdash \Psi[M_2] :_\kappa \sigma$.

3. *If $\Gamma; \Phi \vdash \langle M_1, M_2 \rangle :_\kappa \sigma$, then there exists σ_1 and σ_2 such that $\Gamma; \Phi \vdash C_1 :_\kappa \sigma_1$, and $\Gamma; \Phi \vdash C_2 :_\kappa \sigma_2$, and $\Gamma \vdash \sigma_1 \times \sigma_2 \leq \sigma$.*

Our represention of evaluation contexts in LF is based on the observation that an evaluation context is simply an LF function of type `md -> md`. We then used a judgement to restrict attention to the members of that type that actually represent evaluation contexts:

```
psi-md      : (md -> md) -> type.
psi-md/eps : psi-md ([s] s).
psi-md/pj1 : psi-md ([s] md/pj1 (F s))
                <- psi-md F.
psi-md/pj2 : psi-md ([s] md/pj2 (F s))
                <- psi-md F.
```

With this definition in hand, we may state the induction hypothesis. For example, the first clause is:

```
module-beta/pj1
  : psi-md F
    -> ofsg HT TT Y
       (F (md/pj1 (md/pair M1 M2))) S
    %% outputs begin here
    -> ofsg HT TT Y (F M1) S
    -> type.
%mode module-beta/pj1 +D1 +D2 -D3.
```

6.3 Subderivations

In various places in the proof we utilize subderivation lemmas in which a judgement is asserted to be derivable by a subderivation of an input derivation. For example, the following lemma arises when proving canonical forms for modules:

Lemma 6.3. *If $\Gamma; \Phi \vdash \pi_1 M :_\kappa \sigma$ is derivable, then there exist $\Sigma\alpha{:}\sigma_1.\sigma_2$ and $\kappa' \sqsubseteq \kappa$ such that $\Gamma; \Phi \vdash M :_{\kappa'} \Sigma\alpha{:}\sigma_1.\sigma_2$ is derivable by a subderivation.*

This arises when, given a module value M whose signature is a sum, we wish to show that M must be a pair. We cannot prove that directly by induction, because the extensionality rule mentioned above makes it possible to ascribe such a signature to M by virtue of ascribing signatures to $\pi_1 M$ and $\pi_2 M$. Using the lemma, we can show that there exists a subderivation that gives M another sum signature, and then we can proceed by induction.

Subderivation requirements are expressed in Twelf using a `%reduces` directive [28]:

```
subder/md/pj1
  : ofsg HT TT Y (md/pj1 M) S
    %% outputs begin here
    -> ofsg HT TT Y' M  (sg/sgm S1 S2)
    -> pty-sub Y' Y
    -> type.
%mode subder/md/pj1 +D1 -D2 -D3.

... proof ...

%reduces D2 < D1 (subder/md/pj1 D1 D2 _).
```

The `%reduces` directive causes Twelf to check that the output `ofsg` derivation is always a subderivation of the input one. Thereafter, that information is used automatically by Twelf's theorem checker when it checks that inductions are valid.

6.4 Inversion

As we discussed in Section 4.3, a crucial component of the metatheory is a collection of inversion lemmas for constructors that are proved using an algorithmic presentation of definitional equality. The core of the inversion arguments is the proof that the algorithm is sound and complete for definitional equality.

Prior to this work, Stone and Harper [35] gave an algorithm for the singleton kind calculus and proved it sound and complete. Unfortunately, it was not possible to utilize that proof in this work, because it relied crucially on a logical relation.

In Twelf, every theorem takes the form of a logic program accompanied by a mode declaration indicating which parameters are inputs and which are outputs. Thus, every Twelf theorem is a simple implication from a universally quantified collection of inputs to an existentially quantified collection of outputs. In particular, it is not possible to nest implications, either on the left or on the right. Since the method of logical relations requires arbitrary nesting of implication on both the left and the right, there is no way to express a logical relations argument in the Twelf meta-logic.[13]

Consequently, it was necessary to develop a *syntactic* proof of correctness of an algorithm for the singleton kind calculus. The proof was summarized previously in Section 4.3. In the proof's formalization, we made heavy use of the explicit context technique from Section 6.1 while establishing the properties of hereditary substitution [40], for reasons similar to those that motivated it for functionality.

7. Conclusion

This work establishes a new high-water mark in the verification of the safety of programming languages. Although many safety proofs exist for various core calculi, none before have existed for any full-scale programming language, due to the daunting complexity of such languages. For the languages we actually use, we have always settled for much less: at most, a general agreement that the language's core aspects have been studied carefully, and that any errors that might exist in the language as a whole must be minor. Thus it is unsurprising that numerous minor errors have been uncovered in supposedly type-safe languages.

Our aim is to place Standard ML on as solid a footing as any core calculus, using the techniques of elaborative semantics and mechanical verification in Twelf to deal with the complexities of a full-scale programming language. In this work we mechanize the proof of type safety of an internal language with equivalent expressive power to Standard ML. It remains, in our ongoing work, to define elaboration of Standard ML into this internal language, along the lines suggested by Harper and Stone [16].

Acknowledgments

References

[1] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* Elsevier, 1984.

[2] Karl Crary and Susmit Sarkar. Foundational certified code in a metalogical framework. In *Nineteenth International Conference on Automated Deduction*, Miami, Florida, 2003. Extended version published as CMU technical report CMU-CS-03-108.

---

[13] Sarnat and Schürmann [33] have suggested that some logical relations argument can be formulated by Twelf by reflecting the logical relation into the object language, but their technique does not appear to generalize to Stone and Harper's proof.

[3] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Ninth ACM Symposium on Principles of Programming Languages*, pages 207–212, Albuquerque, New Mexico, January 1982.

[4] Joëlle Despeyroux, Amy Felty, and André Hirschowitz. Higher-order syntax in Coq. In *1995 International Conference on Typed Lambda Calculi and Applications*, volume 905 of *Lecture Notes in Computer Science*, Edinburgh, April 1995. Springer-Verlag.

[5] Derek Dreyer. *Understanding and Evolving the ML Module System*. PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, May 2005.

[6] Derek Dreyer, Karl Crary, and Robert Harper. A type system for higher-order modules. In *Thirtieth ACM Symposium on Principles of Programming Languages*, pages 236–249, New Orleans, Louisiana, January 2003.

[7] Derek Dreyer, Robert Harper, Manuel M.T. Chakravarty, and Gabriele Keller. Modular type classes. Technical Report TR-2006-03, University of Chicago, April 2006.

[8] Sophia Drossopoulou and Susan Eisenbach. Towards an operational semantics and proof of type soundness for Java. In *Formal Syntax and Semantics of Java*. Springer-Verlag, March 1998.

[9] Sophia Drossopoulou, Tanya Valkevych, and Susan Eisenbach. Java type soundness revisited, September 2000. Technical report, Imperial College London.

[10] Michael J. C. Gordon and Tom F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, 1993.

[11] Robert Harper. A simplified account of polymorphic references. *Information Processing Letters*, 51(4):201–206, 1994. Follow-up note in *Information Processing Letters*, 57(1), 1996.

[12] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.

[13] Robert Harper and Daniel R. Licata. Mechanizing language definitions. Submitted for publication, April 2006.

[14] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Twenty-First ACM Symposium on Principles of Programming Languages*, pages 123–137, Portland, Oregon, January 1994.

[15] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *Seventeenth ACM Symposium on Principles of Programming Languages*, pages 341–354, San Francisco, January 1990.

[16] Robert Harper and Chris Stone. A type-theoretic interpretation of Standard ML. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. The MIT Press, 2000. Extended version published as CMU technical report CMU-CS-97-147.

[17] Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. Technical Report 0400001T.1, National ICT Australia, Sydney, March 2004.

[18] Daniel K. Lee, Karl Crary, and Robert Harper. Mechanizing the metatheory of Standard ML. Technical Report CMU-CS-06-138, Carnegie Mellon University, School of Computer Science, 2006.

[19] Xavier Leroy. Manifest types, modules and separate compilation. In *Twenty-First ACM Symposium on Principles of Programming Languages*, pages 109–122, Portland, Oregon, January 1994.

[20] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *Twenty-Second ACM Symposium on Principles of Programming Languages*, San Francisco, January 1995.

[21] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, Massachusetts, 1997.

[22] Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract models of memory management. In *Conference on Functional Programming Languages and Computer Architecture*, pages 66–77, La Jolla, California, June 1995.

[23] Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5:363–397, 1989.

[24] Leaf Petersen. *Certfying Compilation for Standard ML in a Type Analysis Framework*. PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, 2005.

[25] Leaf Petersen, Perry Cheng, Robert Harper, and Chris Stone. Implementing the TILT internal language. Technical Report CMU-CS-00-180, Carnegie Mellon University, School of Computer Science, December 2000.

[26] Frank Pfenning. Computation and deduction. Lecture notes, available electronically at `http://www.cs.cmu.edu/~twelf`.

[27] Frank Pfenning and Carsten Schürmann. *Twelf User's Guide, Version 1.4*, 2002. Available electronically at `http://www.cs.cmu.edu/~twelf`.

[28] Brigitte Pientka and Frank Pfenning. Termination and reduction checking in the logical framework. In *Workshop of Automation of Proofs by Mathematical Induction*, June 2000.

[29] Benjamin Pierce and Martin Steffen. Higher-order subtyping. *Theoretical Computer Science*, 176(1–2):235–282, 1997.

[30] Benjamin C. Pierce. *Types and Programming Languages*, chapter 13. The MIT Press, 2002.

[31] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[32] John C. Reynolds. The essence of Algol. In J. W. de Bakker and J. C. van Vliet, editors, *Proceedings of the International Symposium on Algorithmic Languages*, pages 345–372, Amsterdam, October 1981. North-Holland.

[33] Jeffrey Sarnat and Carsten Schürmann. A proof-theoretic account of logical relations. Submitted for publication, 2006.

[34] Christopher A. Stone. *Singleton Kinds and Singleton Types*. PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, August 2000.

[35] Christopher A. Stone and Robert Harper. Extensional equivalence and singleton types. *ACM Transactions on Computational Logic*, 2006? To appear. An earlier version appeared in the 2000 Symposium on Principles of Programming Languages.

[36] Donald Syme. Reasoning with the formal definition of Standard ML in HOL. In *Sixth International Workshop on Higher Order Logic Theorem Proving and its Applications*, pages 43–60, Vancouver, August 1993.

[37] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *1996 SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–192, May 1996.

[38] Myra VanInwegen. *The Machine-Assisted Proof of Programming Language Properties*. PhD thesis, University of Pennsylvania, Philadelphia, Pennsylvania, May 1996.

[39] Myra VanInwegen and Elsa Gunter. Hol-ml. In *Sixth International Workshop on Higher Order Logic Theorem Proving and its Applications*, pages 61–74, Vancouver, August 1993.

[40] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework: The propositional fragment. In S. Berardi, M. Coppo, and F. Damiani, editors, *Types for Proofs and Programs*, volume 3085 of *Lecture Notes in Computer Science*, pages 355–377. Springer-Verlag, 2004. Papers from the Third International Workshop on Types for Proofs and Programs, April 2003, Torino, Italy.