Type Systems for Modules  
Notes for Meeting #2  

Scott Kilpatrick  
skilpat@mpi-sws.org  
10 Nov 2010

1 Papers

- Xavier Leroy. *Manifest types, modules, and separate compilation*, POPL 94.

2 Summary

The papers for this meeting both address the same problem with (the then-current version of) the SML module system: the disconnect between type information about modules contained in the modules themselves and contained in those modules’ signatures. Both papers introduce a similar addition to the module system, but they formalize corresponding module type systems differently.

At the time, SML structures\(^1\) contained entirely transparent type components; e.g., for any structure named \(M\) that defines a type component \(t\) equal to \(\text{int}\), the projection \(M.t\) would be transparently equal to \(\text{int}\). However, the signature language could only express type components as if they were opaque, e.g. `sig
type t end`. Although signatures were meant to capture the static information about modules, they could not capture the definitions of modules’ type components.

2.1 Leroy 94 [3]

Leroy identifies this problem as one of separate compilation. He argues that for a module \(M_1\) in one file to be compiled separately from — but still used by — a module \(M_2\) in another file, \(M_2\) must know about the types contained in \(M_1\). At first glance, this seems to be exactly what signatures provide because, after all, \(M_2\) could know about \(M_1\)’s (most specific, derived) signature \(S_1\). As the above example shows, however, the signature language is too weak to express the type identities of a module, so \(M_2\) knows only about the existence of \(M_1\)’s type components rather than their definitions.

Leroy introduces the notion of *manifest types* to increase the expressiveness of signatures. A manifest type, as opposed to an abstract type, is a type component in a signature with an explicit definition, e.g. `sig
type t = int end`. With manifest types, signatures become rich enough to express type equivalencies across compilation unit boundaries (i.e. between modules in different files). Manifest types are also shown to express sharing constraints on functor parameters.

He formalizes his system in a small module calculus with a direct type system given by the signature language. The notion of type strengthening (p. 117) is required to type check some desirable expressions. Under strengthening, a named module’s abstract type components can be strengthened into transparent type components whose definitions are “self-projections” from the module’s name. For example, if path \(p\) has type

\(^1\)Except for structures declared with the abstraction keyword in SML/NJ, which were entirely opaque.
sig type t; ... end then it can be strengthened to the (more specific) type sig type t = p.t; ...
end. He also provides a deterministic, syntax-directed type inference algorithm for his nondeterministic
static semantics, but as discussed in Question 3 below, its stated properties are not correct.

The expressiveness of Leroy’s module system — with manifest types represented as a distinct form of
existential — as compared to those dependently typed systems of MacQueen [4] and Harper-Mitchell [2] is
explored in §4. He shows that for all but higher-order functors (§4.3) his system is at least as expressive as
their system, with respect to typing.

2.2 Harper-Lillibridge 94 [1]

Harper and Lillibridge describe the signature problem as one of opacity. Instead of allowing only opaque type
components in signatures, they propose — like Leroy’s manifest types — transparent type components in
signatures. Unlike the abstraction keyword for structures in SML/NJ, their new signature forms allow for
translucent signatures: those signatures whose opacities are specific to each component, rather than the entire
signature. In this way, a single structure can be ascribed signatures of varying opacity, and the generativity
of a structure is thus dictated by the opaque type components of its signature. As with Leroy’s manifest
types, they explain how to more naturally phrase sharing specifications with translucent signatures.

To formalize translucent signatures, they introduce the notion of a translucent sum (§3), which are
“n-ary labeled dependent sums whose types can optionally contain information about the contents of their
constructor fields. Traditional records and weak sums (existentials) are degenerate forms of translucent sums”
(p. 127). A translucent sum type, which corresponds to a signature, contains a list of labeled declarations,
with each describing (1) a kinded type constructor declaration, (2) a kinded type constructor definition, or
(3) a typed term declaration. A translucent sum term, which corresponds to a structure, contains a list
of labeled bindings, with each describing (1) a type constructor definition or (2) a term definition. They
formalize their calculus as a direct type system which, in their words, is based on System F_ω rather than a
dependently typed calculus.

3 Questions

1. Q: Leroy uses stamps to “distinguish identifiers having the same name”. Harper-Lillibridge make an explicit distinction between names (labels) and variables (internal names). Compare the two approaches. Why is any of this necessary?

A: Leroy’s stamp-based approach is under-specified and imprecise, whereas the Harper-Lillibridge approach provides a clearer semantics in which the external name is a fixed label and the internal name is a standard, \( \alpha \)-convertible variable.

Internal names are necessary because the names need to be \( \alpha \)-convertible, yet external references to components need to be maintained despite any such \( \alpha \)-conversion. Distinct internal names also facilitate dependence on nested components with the same name (see example in [1] at top of p. 128, left column). Without internal distinct names, substitution can lead to problems with capturing. For example, the functor

\[
F(type x: \Omega) = struct 
\text{type } t = \text{int} 
\text{type } u = x 
end
\]

in a context that binds \( type \ t = \text{bool} \), admits an application \( F(type x = t) : \{type \ t = \text{int}, type \ u = t\} \) — note that the application’s type reflects that the \( u \) component is equal to the \( t \) component, 2

2Specifically, this is true only of structures sealed by that signature or of structures about which only signatures are known, such as functor parameters. The notion of sealing is described as “forced coercion” in §4.4.
yet the evaluation yields two different types, \texttt{bool} and \texttt{int}. With a separation between component labels and internal variables, the \texttt{t} in the functor argument (and in the context) would actually be a type variable \( t \), and thus the resulting signature would refer to that \( t \) rather than its own \( t \) component.

2. \textbf{Q:} Leroy writes: “The reason why general projections (and even projections restricted to values, as in Harper and Lillibridge [10]) are inadequate is that we have abstract types and therefore must account for type generativity.” Do you believe that?

\textbf{A:} His statement lacks a clear interpretation. Leroy’s system only permits projections from named paths whereas the Harper-Lillibridge system permits projections from any module value. A module’s type, thanks to manifest types, now accounts for type generativity per component, so more fine-grained control of projection could be allowed. For example, if the functor application \( m_1(m_2) \) has type \texttt{sig type t; type u = int end}, there should be no reason to prohibit the projection \((m_1(m_2)).u\) as \texttt{int} simply because the \( t \) component is abstract.

3. \textbf{Q:} The “completeness” property that Leroy claims in section 3.5 is false. Can you find a counterexample? (There’s a very simple one.) Is there an easy fix?

\textbf{A:} The problem lies in the newly restricted use of the subsumption typing rule (which says that if \( m : M \) and \( M <: M' \) then \( m : M' \)), which in the syntax-directed rules may only be applied, along with strengthening, on functor arguments (and module bindings). Moreover, in the paragraph below the typing rules for module expressions on p. 115, Leroy explains that functor applications are ill-typed if either the functor has a dependent signature (i.e., if the functor parameter appears in the result signature) or the argument is not a path. Then a functor application with exactly these restrictions seems an interesting test case.

Consider the following example. \( S \) is a metavariable for the signature \texttt{sig type t end} and \texttt{Id} is a functor variable with type \texttt{X:S \rightarrow S with type t = X.t} bound in the context \( \Gamma \). In the old system, we could prove

\[ \Gamma \vdash \texttt{Id(struct type t = int end):S} \]

by using subsumption on \texttt{Id} to weaken its type to the non-dependent \( S \rightarrow S \), which then means that the application is well-typed. In the new system, however, we cannot weaken \texttt{Id}’s type to be non-dependent and moreover the argument is not a path, so we are stuck with an ill-typed application. Therefore the syntax-directed typing judgment is \textit{not} complete.

4. \textbf{Q:} Critique section 4 of Leroy’s paper.

\textbf{A:} Leroy’s stripped-down calculus in §4 makes no mention of type generativity or sharing. \textit{Response needed.}

5. \textbf{Q:} Harper and Lillibridge’s calculus is very similar in many ways to Leroy’s. Can you enumerate the differences?

\textbf{A:} \textit{Response needed.}

6. \textbf{Q:} Unlike Leroy’s calculus (which ignores reordering), Harper and Lillibridge’s subtyping relation accounts for the three main aspects of signature matching in ML: dropping components, reordering of components, and forgetting the identity of type components. Yet it does not fully account for signature matching of the sort found in modern SML or OCaml. Can you come up with two closed signatures \( S_1 \) and \( S_2 \), such that \( S_1 \) matches \( S_2 \) in SML or OCaml, but where the encodings of \( S_1 \) and \( S_2 \) in Harper-Lillibridge are not in the subtype relation?

\textbf{A:} The encoding of sharing specifications mentioned by Harper and Lillibridge (first paragraph of §4.5) produces asymmetric signatures, whereas the sharing specification in SML is treated symmetrically. For example, consider the two SML signatures \( S_1 \) and \( S_2 \) defined as follows:
Their representations in the Harper-Lillibridge calculus are as follows:

\[ S_1 := \{ t \triangleright t :: \Omega, \quad u \triangleright u :: \Omega = t \} \]

\[ S_2 := \{ u \triangleright u :: \Omega, \quad t \triangleright t :: \Omega = u \} \]

\( S_1 \) matches \( S_2 \) in SML, but under their subtyping rules, \( S_1 \not\leq S_2 \) since the type components have distinct opacities between the two sums.

7. Q: Harper and Lillibridge prove that their type system is undecidable. In more informal, less technical terms, can you explain what is the key aspect of the language that makes it undecidable? How might you restrict the language to recover decidability?

A: Response needed.

References


