Modules, Abstract Types, and Distributed Versioning

Peter Sewell*
Computer Laboratory, University of Cambridge
Peter.Sewell@cl.cam.ac.uk

September 29, 2000

Abstract

In a wide-area distributed system it is often impractical to synchronise software updates, so one must deal with many coexisting versions. We study static typing support for modular wide-area programming, modelling separate compilation/linking and execution of programs that interact along typed channels. Interaction may involve communication of values of abstract types; we provide the developer with fine-grain versioning control of these types to support interoperation of old and new code. The system makes use of a second-class module system with singleton kinds; we give a novel operational semantics for separate compilation/linking and execution and prove soundness.

Contents

*Supported by a Royal Society University Research Fellowship.

1 Introduction

Background Module systems provide an important tool for structuring large programs, both to express their conceptual structure and to support separate compilation and linking. They have been much studied – most relevant to this paper is a line of work on ML-style modules [MTH90], with *structures* (collections of named types, values and substructures) and *functors*, which are parameterised structures. A key issue is the treatment of sharing equality for abstract types. The original ML static semantics involved explicit generation of new type names; the translucent sums/manifest types of Harper and Lillibridge [HL94] and Leroy [Ler94] showed that more type-theoretic treatments were possible and could be expressed using the machinery of singleton kinds. These works treat separate compilation and linking either implicitly or as applications of higher-order functors. Cardelli [Car97] gives a more explicit model of linking, though without abstract types or parameterised modules. Various aspects of modularity and linking have been investigated more recently, e.g. in [BA99, Dro00, Dug00, FF98, GM99, HWC00].

Problem The works cited above focus on development of single sequential programs. In this paper we address issues arising from wide-area distributed programming. There are two key differences:

1. We must deal with whole programs that interact with each other at arbitrary interfaces, not simply programs that interact with the outside world at the fixed types of library functions.
2. In wide-area distributed systems it is often impractical to synchronise software updates, so we must deal with many coexisting versions of programs that interact with each other.

Further, in a wide-area system it becomes particularly important to detect errors early in the software development process; it is therefore worth using static typing as far as possible – and we would like to see how far. The language described here does not involve any run-time checks (save for the implicit equality testing of channel names that is intrinsic to channel-based interaction).

Outline In the following two sections we develop a model language, equipped with static and dynamic semantics, that supports:

1. Separate compilation/linking and execution for modular programs that interact along typed channels.
2. Interaction along channels carrying abstract types.
3. Version control of those abstract types, for interoperation of old and new code.

It consists of a core expression language, a module system, and an imperative command-line language. The core language is taken to be an asynchronous π-calculus, providing a concise form of typed interaction between whole programs. The module system is based on a standard second-class system, with first-order functors and using singleton kinds. It is extended with new channel creation and with a novel type coercion for versioning. The command-line language allows compiling and linking (here merged into a single step) of a module to an object file, and executing such files with 'main' components. The semantics requires careful treatment of new name creation, both for channel names and type names – in brief, abstract types must be compiled to manifest types with new-bound type names; these are used by the versioning coercion. New type names arise in module reduction steps such as

```
[T,e] as ∃X::Type.T′
   ⟶
New Y::EQ(T) in
  [Y,e] as ∃X::EQ(Y).T′
```

Here the first line is an existential package with a hidden representation type `T` and an operation part `e`; the reduction creates a new name `Y` for the representation type, recording it in both the term and signature parts of a manifestly-typed package. The scope of the new type name must extrude to cover both compiled object code (as stored in files) and the running system.

The versioning coercion allows the developer, when building a module expression `m` and assigning the result to a filename `B`, to declare that abstract types provided by `m` should be made compatible (if possible) with the previous version of the module stored in `A`. This is written `B := (m with! A)`.

We prove subject reduction for both build-time and run-time semantics, show the absence of run-time errors, and (in a simple case) relate monolithic and separately-compiled programs, giving a tight link between the singleton-kind system and the type name generation occurring during module reduction. Section 4 concludes with discussion of related work and future directions. The full definitions of the typing judgements, build-time semantics and run-time semantics for the language are given in Appendix A. Details of the soundness proofs are in Appendix B. This is a full version of the paper [Sew01].

The original motivation for this paper arose from work with Pierce, Unyapoth and Wojciechowski on Nomadic Pict [SWP98, WS00, Woj00, US01], a distributed programming language designed to express infrastructure algorithms for location-independent communication between mobile computations. Nomadic Pict is based on a distributed process calculus (Nomadic π) following the concurrent Pict language of Pierce and Turner [PT00], based on the π-calculus [MPW92]. Our work on the language, and on distributed infrastructures expressed in it, has shown a clear need for the support outlined above. The problems are more general, however – similar issues would arise in many other settings where programs interact and have long execution lifetimes (as compared with the development cycle). The exact form of interaction is more-or-less orthogonal to the typing issues. Nomadic Pict has asynchronous message passing to named channels at located agents, but typing would be similar for e.g. Distributed Join-style communication [FGL$^+$96], interaction via persistent references, or RPC mechanisms. For simplicity we here adopt standard π-calculus communication, omitting explicit distribution from the core language. This is not a realistic form of wide-area interaction, but it would be straightforward to extend the system with the Nomadic π distribution and mobility primitives, thereby allowing modularisation of our distributed infrastructures.

We are not here dealing with problems of *numbered versions* (some of which will be familiar to those who regularly encounter DLLs), but regard what we do as a necessary preliminary for a satisfactory treatment. We also do not address dynamic linking [GM99, HWC00, Dro00, Dug00] except in the limited sense that a program might be replaced by another that interacts on an overlapping set of channels, nor do we consider hot code upgrade [AWWV95, HN00], in which new code is given access to the datastructures of that being replaced. These are clearly sometimes required, but inevitably lead to the possibility of late link-time or upgrade-time errors; we are exploring how far one can go without them.

2 Informal Discussion

This section discusses the issues and our solutions informally, leaving rigorous development to §3.

2.1 Interaction along typed channels

We begin by discussing separate compilation/linking for programs that interact by message passing along channels carrying values of simple types. The development is quite straightforward, but it is interesting to see how it must diverge from the single-program case (e.g. as in the flat modules of [Car97]), and it is a necessary preliminary for addressing abstract types later.

To introduce our core language, consider the monolithic π-process

```
new c : int chan in (c!27 | c?x->(c!(x+1)))
```

$C \stackrel{def}{=}$
```
module
  export c : int chan
begin
  newval c : int chan
end
```

$MAIN1 \stackrel{def}{=}$
```
module
  import c  : int chan
  export main : proc
begin
  val main : proc = c!27
  end
```

$MAIN2 \stackrel{def}{=}$
```
module
  import c : int chan
  export main : proc
begin
  val main : proc = c?x->(c!(x+1))
  end
```

Figure 1: Flat modules with `newval`

This declares a new channel `c` of type `int chan` (for messages of type `int`); it has an output `c!27` of 27 along `c` in parallel with an input `c?x->(c!(x+1))` that receives an integer on `c`, binding it to `x`, and then outputs `x+1`.

In a distributed system this process might be split, with the output `c!27` in one program and the input `c?x->(c!(x+1))` in another program that will be executed concurrently, perhaps on a different machine. This is expressed in Figure 1 in the simplest flat module system of [Car97], extended with a `newval` declaration for declaring new channels. The whole system would be built and run by compiling *C*, *MAIN1* and *MAIN2* to give some *C′*, *MAIN1′* and *MAIN2′*, linking *C′* with each of the other two, and executing the two resulting complete programs. To ensure statically that communications on `c` are well-typed, even if some other program declares `c` to be of type (say) `(int*int*int) chan`, a new internal channel identifier must be generated when *C* is compiled. This can be represented by taking *C′* to be

```
module
  export c : int chan
begin
  val c : int chan = z
end
```

where `z` is fresh (in a sense made precise below). At this level of abstraction compilation is simply intra-module type checking and new channel name generation. Linking *MAIN1′* and *MAIN2′* with this will give outputs and inputs on `z`.

Unfortunately, to keep soundness we can only generate a new channel name when the type it will carry is known. To see this, suppose one compiled

$D \stackrel{def}{=}$
```
module
  import type t
  export val c : t chan
begin
  newval c : t chan
  end
```

to give

$D' \stackrel{def}{=}$
```
module
  import type t
  export val c : t chan
begin
  val c : t chan = z2
  end
```

```
C :=     struct
           newval c : int chan
         end
MAIN1 := (functor(U: sig val c:int chan end) struct main = (U.c)!27 end) C
MAIN2 := (functor(U: sig val c:int chan end) struct main = (U.c)?x⇒((U.c)!(x+1)) end) C
```

Figure 2: A sequence of build commands with `newval`

with fresh z2. One could then successfully build two programs that attempt to interact on z2 but have a run-time error, as follows. Take

T3 $\stackrel{def}{=}$	*T1* $\stackrel{def}{=}$

`module`
`  export type t`
`begin`
`  type t=int*int*int`
`end`
	`module`
	`  export type t`
	`begin`
	`  type t = int`
	`end`

and *MAIN3* to be as *MAIN1* but with an output of a triple. Write *T3′*, *T1′* and *MAIN3′* for the results of compiling these, then consider interaction between the two programs built by linking *T3′*, *D′*, *MAIN3′* and linking *T1′*, *D′*, *MAIN2′*.

To avoid dealing with compilation and linking of such modules that declare new channels depending on unresolved imported types, we force a bottom-up build order. We take module expressions m including structures, functors and functor application. Compiling and linking are merged into 'build-time' evaluation of such module expressions and assignment of the resulting module values to filenames. A module being built can refer only to filenames of previously-built modules; it is typechecked with respect to their stored signatures (in examples we sometimes functorise to make signature constraints explicit). Figure 2 recasts the example into a sequence of three assignments, to filenames C, MAIN1 and MAIN2. We pun filenames and free module identifiers (both written in an upright font). Module expressions are reduced call-by-value, with a structure

C2 $\stackrel{def}{=}$

```
struct
   newval c : int chan
end
```

reducing to a value

```
struct
  val c : int chan = z
end
```

for a fresh z.

Now, in what sense must this z be fresh? It should be distinct from all channels generated earlier, either at build-time or at run-time, across the whole distributed system. In an implementation a globally unique bitstring must be generated. We represent this using name binding and scope extrusion, as in the π-calculus. To a first approximation, *C2* above reduces to

```
New z : int chan in
  struct
     val c : int chan = z           (*)
  end
```

In more detail, we idealise the state of the whole system, including both the running computations and the module values stored in the various developers' filesystems produced by earlier builds, as a triple

```
N,F,e
```

which should be read as `New N in (F,e)`. Here `N` is a type environment of new bindings such as `z : int chan`, the `F` models the union of all filesystems[1] as a finite list of pairs of module ids and module values (with no repeated ids – `F` will often be regarded as a partial function), and `e` is the running process expression. We take these triples up to alpha-renaming of the bindings in `N`. Now the effect of developers' command-line build and load commands, and computation steps, can all be regarded as changes of system state. Executing a build command

```
C := struct newval c : int chan end
```

(say) in the state `N,F,e` results in the new state $\mathtt{N}',\mathtt{F}',\mathtt{e}$, where $\mathtt{z} \notin \mathrm{dom}(\mathtt{N})$ and

$$\begin{array}{rcl} \mathtt{N}' & = & \mathtt{N,z:int\ chan} \\ \mathtt{F}' & = & \mathtt{F} \oplus (\mathtt{C} \mapsto \mathtt{struct\ val\ c:int\ chan\ =\ z\ end}) \end{array}$$

The new value of `C` in $\mathtt{F}'$ is obtained by module reduction. Note that for simplicity we are working as much as possible at the source language level, taking a single set of names rather than separate sets of program identifiers and internal channel ids. The `N` binding ensures that code written after a build command cannot mention any of the newly generated names. Note also that `N` is kept to allow proof of soundness; in an implementation it can be discarded.

For the load command, if `U` is a filename referring to a module value `struct val main : proc = e' end` then executing the command

```
run U
```

in the state `N,F,e` results in the new state `N,F,e|e'` in which `e'` is put in parallel with the existing running computation, allowing them to interact.

The computation reductions of a state `N,F,e` are simply those of its process part `e`.

2.2 Interaction with Abstract Types

We now address systems that interact by communicating elements of abstract types. Similar new-name machinery will be required to ensure soundness, though now for type names rather than channel names. We first recall some aspects of ML-style type abstraction, particularly with translucent sums/manifest types in signatures [HL94, Ler94, Ler96, Ler00, Lil97] (for brevity the examples will all be degenerate, without any operation parts or field dependencies). The module

```
A =  struct  type t = int  end
      : sig  type t        end
```

provides a type `A.t` with representation `int`. It has an explicit signature in which no information about the representation is visible, though, so from the outside the type is entirely abstract. In contrast, the module

```
C =  struct  type t = int  end
      : sig  type t = int  end
```

[1] The semantics uses a disjoint union of all filesystems, implicitly extruding the `New N` of `(*)` to the outside, to reduce notational clutter. It would be straightforward to give an equivalent model with explicit extrusion that would keep distinct filesystems.

```
T     := struct type t=int  val x=27   val i=λz.z   end : TSIG
C     := (functor(T:TSIG) struct newval c : T.t chan end) T
MAIN1 := (functor(T:TSIG, C:CSIG(T.t)) struct main = (C.c)!(T.x) end) T C
run MAIN1
wait 6 months
T     := struct type t=int*int  val x=(27,3) val i=λ(z,w).z end : TSIG
MAIN2 := (functor(T:TSIG, C:CSIG(T.t)) struct main = (C.c)?y⇒ ... (T.i)y ... end) T C
run MAIN2
```

where we use abbreviations

```
TSIG   =def sig type t  val x:t  val i:t→int end       CSIG(X) =def sig val c : X chan end
```

Figure 3: Attempted communication and use of elements of a changed abstract type

reveals that its representation type is `int`; the equality `C.t=int` may be used in typechecking the rest of the program. Such manifest types are particularly important in functor signatures. For example

```
functor(U : sig type t end)
  struct type t2=U.t end
   : sig type t2=U.t end
```

allows code using a structure, say `D`, created by applying this anonymous functor to `A`, to depend on the fact that `D.t2` and `A.t` are equal types.

Translucent sums and manifest types were motivated partly by the need to refine SML modules to provide enough type equality information in signatures for separate compilation, partly by concerns of higher-order functors and first-class modules that we do not discuss here, and partly by a desire to move from the generative SML static semantics to a more flexible type-theoretic style. Intuitively, instead of the SML semantics' use of new type names to distinguish between otherwise-identical abstract types, they use the module identifiers (or, more generally, paths) that occur in the source program.

In our distributed setting, we must reintroduce type name generation, albeit in a more controlled form. The broken example in Figure 3 shows why. Here we have two programs, `MAIN1` and `MAIN2`, communicating elements of an abstract type from module `T`, on a channel from `C`. Unfortunately `T` is re-built, with a changed representation type, between the builds of `MAIN1` and `MAIN2`. There will be a run-time error. To prevent this we should detect a build-time error when typechecking the functor application in the build of `MAIN2`, as the values of `C` and `T` there are not compatible. To make this lack of compatibility evident, we generate new type names when evaluating module expressions, reducing an abstract type to a type which is manifestly equal to a fresh type name. For example,

```
struct  type t = int  end          (*)
 : sig  type t        end
```

reduces to

```
New X = int in
  struct  type t = X end
  : sig   type t = X end
```

and the body

```
struct  type t = X end
: sig   type t = X end
```

of this will be a module value whereas `(*)` will not. Nonetheless, we keep the type-theoretic style as much as possible – type checking of a module expression in any individual build command will not involve new name generation. We need `t=X` both in the structure, so that it is correctly propagated during module substitution (when a functor is applied to this structure), and in the signature, so that there is enough type sharing.

The system of §3 uses singleton kinds, due independently to Harper and Leroy, for expressing manifest type declarations in signatures. There are kinds `Type`, of all types, and `EQ(T)` for any type `T`, of all types provably equal to `T`. In the example signatures above `type t` becomes `type t::Type`, and `type t=t'` becomes `type t::EQ(t')`.

2.3 Abstract Type Versioning

The semantics outlined in the previous subsection guarantees soundness but, because every rebuild gives incompatible executables, it would be unusably rigid in practice. For example, one might initially build compatible `MAIN1` and `MAIN2` with the first `T` command and the `C`, `MAIN1` and `MAIN2` commands of Figure 3. Rebuilding `MAIN1` from the same source code, i.e. with the `C`, first `T` and `MAIN1` commands again, would give a version that could not interact with copies of `MAIN2` running elsewhere in the network. `MAIN1` and `MAIN2` would have different versions of both the type `T.t` and the channel `C.c`.

The developer therefore needs a mechanism for forcing a rebuild of a module that provides an abstract type `t` or a channel `c` to produce *the same* abstract type or channel as before. We address only the (more interesting) type part of the problem; the channel part can be dealt with using similar mechanisms. There are four cases:

1. The source code of the module (and all that it depends upon) is syntactically unchanged.
2. The source code is changed, but the representation type of `t` is unchanged and the new code has the same important invariants as the old.
3. The representation type is unchanged but the new code has changed invariants.
4. The representation type is changed.

Naively, one might allow a rebuild to produce the same type only in case 1. This could be entirely automatically checked, but would be too inflexible – we must allow for changed comments, performance improvements, minor bug fixes, and even some changes in functionality. On the other hand, in cases 3 and 4 a rebuild should certainly produce a new type. Distinguishing between cases 2 and 3 clearly cannot be done automatically, so the developer must provide some annotation for the new code, asserting that it has not changed any important invariant of the old abstract type, and hence that values produced by the old and new code will be compatible. Such an annotation should force an automatic check that the representation type is unchanged.

We envisage that large programs will require development environment support for managing these annotations, allowing defaults such as 'always generate a new type', or 'always produce types compatible with the previous build unless the code is syntactically changed', or 'always produce compatible types unless the representation type has changed' – all to be overridden locally as needed. There is an important pragmatic question here, of what expressiveness is really necessary or useful, but we do not investigate it in this paper. Instead, we introduce a minimal form of annotation, that might be generated by the development environment from higher-level defaults, and show how it can be given a sound semantics.

In particular, we allow module expressions containing coercions, e.g. as in the right hand side of the build command

```
A := m with! U                  (*)
```

Suppose `m` is statically type-checkable with signature `sig type t::Type val x:tops end`, and that `U` is a filename that refers to a previously-built module of similar shape. Executing command `(*)` will evaluate `m` to a struct, say

```
struct   type t=trep   val x=e      end
 : sig   type t::Type  val x:tops  end
```

check that its representation type `trep` is compatible with that of `U`, and finally reduce it to a module value containing the same type name as `U`. In more detail, if the state in which `(*)` is executed has an `N` component with `X::EQ(trep2)`, and an `F` component mapping `U` to

```
struct   type t=X       val x=e2 end
: sig    type t::EQ(X)  val x:tops2 end
```

then executing `(*)` will check `trep` and `trep2` are equal and result in a state with `A` mapped to

```
struct   type t=X       val x=e     end
 : sig   type t::EQ(X)  val x:tops  end
```

The notion of type equality required is complicated by the fact that the representation types may themselves involve other abstract types. It is made precise in §3, where it is also shown that it is not necessary to keep the whole development history in order to check it, but only modules actually referred to in coercions.

Note that the coercion must involve a build-time check, during evaluation of module expressions, as `m` must be reduced to a structure to make its representation type available.

3 Formal Development

In this section the previous informal discussion is made precise by giving a language of interacting processes, modules, and commands. It is equipped with build-time and run-time semantics, which are proven sound. The full syntax is given in Figure 4. We begin by discussing the choice of constructs.

Commands There are two command-line commands. Executing the build command `U:=m` type-checks and evaluates the module expression `m` and assigns the resulting module value to filename `U`. Executing `run U`, where `U` is a filename referring to a previously-built main module, loads the process part of that module in parallel with the rest of the running system. The grammar includes also `tau` – computation steps of the running system may be interleaved with build and load commands. We have simplified the system (in an unimportant way) by not representing source code stored in files, and (more significantly) by not dealing with separate builds of module interfaces.

Modules We take as simple a module language as possible: second class, with only first-order functors, without substructures, and with structures that have single type and term parts, not general dependent records. We do include singleton kinds, though – both to allow non-trivial sharing between functor arguments and results, and so that the type equality check needed for `with!` can be expressed. The structure

```
[T,e] as [X::K,T2]
```

is similar to our informal

```
struct  type t=T   val x=e  end
 : sig  type t::K  val x:T2 end
```

It consists of a pair of a type `T` (of kind `K`) and a term `e` (loosely, of type `{T/X}T2`). The kind `K` here might be `Type`, making this a fully abstract structure with representation type `T` hidden, or `EQ(T')` for any `T'=T`, revealing the representation type. The type and term parts of a structure variable `U` can be projected out by `U.Type` and `U.term` (if this can be given a type). The functor

```
λU:SS.m:S
```

Commands

```
Com  ::= U := m              build
         run U               load
         tau                 execute a step
```

Structure signatures

```
  SS ::= [X::K,T]            structure sig
```

Module signatures

```
   S ::= SS
         ΠU:SS.S             functor sig
```

Kinds

```
   K ::= Type                kind of all types
         EQ(T)               kind of types equal to T
```

Types

```
   T ::= X                   variable
         U.Type              type part of a structure
         [T .. T] |  int     tuples and integers
         T chan   |  proc    channels and processes
```

Module expressions

```
   m ::= U                   variable
         [T,e] as SS         structure
         λU:SS.m:S           functor
         m m                 application
         m:SS                seal
         new x:T in m        new channel
         m with! m           version coercion
```

Core expressions

```
   e ::= x                   variable
         U.term              term part of a structure
         n                   integer
         [e .. e]            tuple
         new x:T in e        new channel
         0    |  e|e         nil and parallel processes
         e!e  |  e?pat->e'   output and input processes
```

Core patterns

```
 pat ::= x  | [pat .. pat]
```

Typing environments

```
   E ::= empty  |  E,x:T  |  E,X::K  |  E,U:S
```

`U`, `X` and `x` range over module, type and term variables respectively. Module variables are also used as filenames. The binding is as follows: in a structure sig `X` binds in `T`; in a functor sig `U` in `S`; in a functor `U` in `m` and `S`; in a new (module or core) `x` in `m` or `e`; and in an input the variables of `pat` in `e'`. We work up to alpha conversion.

Figure 4: Syntax

is a dependent function taking structures of signature SS and returning modules of signature S, which may mention U. Functors λU:SS1.([T,e] as SS2):SS2 will often be written λU:SS1.[T,e] as SS2, eliding one copy of SS2 (the explicit annotation is in the syntax just so the filesystem only has to contain module values).

To these standard constructs we add a module-level new channel declaration and our with! coercion. The module-level new allows uses of the newval of §2.1 such as struct ... newval c : int chan end to be expressed as

```
new c:int chan in ([..,c] as [..,int chan])
```

(this is a little awkward, however – a channel carrying an abstract type must be declared in a different structure than the type). The module expression m with! m2, where m and m2 both have structure signatures, attempts to evaluate m and coerce it to provide an abstract type compatible with m2.

Processes The core language is simply an asynchronous π-calculus with tuples, allowing communication on newly-generable channels between parallel outputs and inputs.

System States The state of the whole system is (exactly as in §2.1) a triple

```
N,F,e
```

where N is a type environment of channel and type bindings (such as z : int chan and X::EQ(T)), F models the union of all developers' filesystems as a finite list of pairs of module ids and module values (with no repeated ids), and e is the running process expression. We take state triples up to alpha-renaming of the variables in dom(N). Module values are simply

```
mval ::= [T1,e] as [X::EQ(T1'),T2]
         λU:SS.m:S'
```

We will never need to deal with mvals that have free module variables. Note that module variables and filenames are punned.

3.1 Typing

The type system for modules and processes is largely standard, with judgements:

```
E⊢ K                      E⊢ S sig
E⊢ K <:: K′               E⊢ S < S′ sig
E⊢ K == K′                E⊢ S == S′ sig
E⊢ T::K                   E⊢ m:S
E⊢ T == T′ :: K           E⊢ ok
E⊢ pat : T ▷ E′
E⊢ e:T
```

It includes subkinding, with EQ(T) <:: Type for wellformed types T, a subsignature relation based on this that allows concrete type information to be forgotten, and a self-type rule for manipulating type equalities (expressed with singleton kinds) in signatures. The typing rules are given in Appendix A.1.

3.1.1 Sharing

We first review the standard aspects of the type system – the use of singleton kinds and subkinding to express ML-style sharing; for further explanation we refer the reader to [HL94, Ler94, Ler96, Ler00, Lil97]. The examples use alternate notation T*T′ and (e,e′) for binary products [T,T′] and pairs [e,e′]. One can write structures that are either abstract or concrete:

```
A  =def  [int, 6] as [X::Type, X]            ⊢ A : [X::Type, X]
C  =def  [int, 6] as [X::EQ(int), X]         ⊢ C : [X::EQ(int), X]
```

A here provides an abstract type and a single value of that type; *C* is a pair of type `int` and value $\underline{6}$ of type `int`. To use a structure it must first be bound to a variable – the language allows projections `U.Type` and `U.term` of the type and term parts only of a structure variable `U`, not of an arbitrary module expression. If `U` has an abstract signature, eg in the type environment `U:[X::Type,X]`, then we know only

```
U:[X::Type,X] ⊢ U.Type :: Type
U:[X::Type,X] ⊢ U.term : U.Type
```

This suffices for typechecking a functor *Fopaque* that builds a new abstract type:

```
Fopaque  =def  λU:[X::Type,X].[U.Type*U.Type, (U.term,U.term)] as [Y::Type,Y]

⊢ Fopaque    :  ΠU:[X::Type,X].[Y::Type,Y]
⊢ Fopaque A :  [Y::Type,Y]
```

Fopaque can also be applied to *C*, using the subsignature relation to ignore the manifest type.

```
⊢ Type <:: EQ(int)
⊢ [X::Type,X] < [X::EQ(int),X] sig
⊢ C : [X::Type, X]
⊢ Fopaque C :  [Y::Type,Y]
```

A more interesting variant of *Fopaque* involves type sharing between argument and result with a dependent functor signature, revealing that the type part of its result is the product of the type part of its argument.

```
Ftrans  =def  λU:[X::Type,X].[U.Type*U.Type,(U.term,U.term)] as [Y::EQ(U.Type*U.Type),Y]
⊢ Ftrans : ΠU:[X::Type,X].[Y::EQ(U.Type*U.Type),Y]
```

Ftrans might be applied to a structure variable of an abstract signature:

```
U′:[X::Type,X] ⊢ Ftrans U′ : [Y::EQ(U′.Type*U′.Type),Y]
```

or to a structure variable of a manifest signature (here assuming ⊢ `T::Type`):

```
U′:[X::EQ(T),X] ⊢ Ftrans U′ : [Y::EQ(T*T),Y]
```

To derive this one can first use the subsignature relation to make the argument signature of *Ftrans* match the manifest signature of `U′`

```
⊢ EQ(T) <:: Type
⊢ [X::Type,X] < [X::EQ(T),X] sig
⊢  ΠU:[X::Type,X].[Y::EQ(U.Type*U.Type),Y]
 < ΠU:[X::EQ(T),X].[Y::EQ(U.Type*U.Type),Y] sig
⊢ Ftrans : ΠU:[X::EQ(T),X].[Y::EQ(U.Type*U.Type),Y]
```

then make use of the type equality `U.Type == T` in the result signature of *Ftrans*

```
U:[X::EQ(T),X] ⊢ U.Type == T :: Type
U:[X::EQ(T),X] ⊢ [Y::EQ(U.Type*U.Type),Y] == [Y::EQ(T*T),Y] sig
⊢  ΠU:[X::EQ(T),X].[Y::EQ(U.Type*U.Type),Y]
 < ΠU:[X::EQ(T),X].[Y::EQ(T*T),Y] sig
⊢ Ftrans : ΠU:[X::EQ(T),X].[Y::EQ(T*T),Y]
```

One can express binary functors that require their two arguments to have equal type parts:

$Fb \stackrel{def}{=}$ `λU1:[X::Type,X].λU2:[Y::EQ(U1.Type),Y].(U1.Type,U2.term) as [Z::Type,Z]`
`⊢` *Fb* `: ΠU1:[X::Type,X].ΠU2:[Y::EQ(U1.Type),Y].[Z::Type,Z]`

so

```
⊢ (λU:[X::Type,X].Fb U U) A  : [Z::Type,Z]
⊢ (λU:[X::Type,X].Fb U (Ftrans U)) A : [Z::Type,Z]
⊬ Fb A A : S
⊬ (λU:[X::Type,X].Fb U (Fopaque U)) A : S
```

We show in more detail how the type system allows variables of abstract and concrete structure signatures to be used:

```
Suppose E=E1,U:[X::Type,T]         (and E⊢ ok and X not in dom E)
By m.var and T.Proj  E⊢ U.Type :: Type

By T=.refl and T.T=  E⊢ U.Type :: EQ(U.Type)
By m.self−type       E⊢ U : [X::EQ(U.Type),T]
By the T= rules      E,X::EQ(U.Type)⊢ T == {U.Type/X}T :: Type
By S=.Struct         E⊢ [X::EQ(U.Type),T] == [X::EQ(U.Type),{U.Type/X}T] sig
By S< and m.S<       E⊢ U : [X::EQ(U.Type),{U.Type/X}T]
By e.proj            E⊢ U.term : {U.Type/X}T


Suppose E=E1,U:[X::EQ(T0),T]
By m.var and T.Proj E⊢ U.Type :: EQ(T0)
By T=.EQ            E⊢ U.Type == T0 :: Type

By the T= rules        E,X::EQ(T0)⊢ T == {T0/X}T :: Type
By S=.Struct           E⊢ [X::EQ(T0),T] == [X::EQ(T0),{T0/X}T] sig
By S<, m.var and m.S<  E⊢ U : [X::EQ(T0),{T0/X}T]
By e.proj              E⊢ U.term : {T0/X}T
```

More useful module examples quickly become rather verbose, especially with our cut-down term language. As a minimal example, one can express an abstract type of pairs that provides a once-only channel interface as follows (writing `T -> T′` for `[T, T′ chan] chan`).

```
new mkpair : int*int −> int*int,
    fst    : int*int −> int,
    snd    : int*int −> int
in
  [int*int, (mkpair,fst,snd, ( mkpair?((x,y),result) → result!(x,y)
                             | fst?((x,y),result)  → result!x
                             | snd?((x,y),result)  → result!y) )]
  as
  [X::Type,  (int*int−>X) * (X−>int) * (X−>int) * proc ]
```

3.1.2 New, with!, and system states

Turning now to the new aspects, the `new` and `with!` module constructs have rules

```
E,x : T chan⊢ m : SS
──────────────────────────── m.new
E⊢ new x : T chan in m : SS
```

```
N⊢ (λU:SS1.m:S2) mval               ⟶ {mval/U}m                               m.red.beta
N⊢ [T1,e] as [X::EQ(T1′),T2] : SS   ⟶ [T1,e] as SS                            m.red.seal
N⊢ new x:T in m                     ⟶ New x:T       in m                      m.red.new
N⊢ [T1,e] as [X::Type,T2]           ⟶ New Y::EQ(T1) in [Y,e] as [X::EQ(Y),T2]  m.red.abstype
 (for Y ∉ dom(N))

m =[T1,e ] as [X::Type, T2 ]
m′=[Z, e′] as [Y::EQ(Z),T2′]
N =N1,Z::EQ(TZ),N2
typify(N)⊢ T1==TZ::Type
──────────────────────────────────────────── m.red.with!
N⊢ m with! m′ ⟶  [Z,e] as [X::EQ(Z),T2]
```

Figure 5: Module reduction axioms

```
E⊢ m :[X::K,T2]
E⊢ m′:[Y::EQ(T′),T2′]
────────────────────────────────────m.with
E⊢ (m with! m′) : [X::EQ(T′),T2]
```

The first allows new channel declaration in a structure. The second allows `m` to be coerced to have the same type part as `m′` – statically it always succeeds (if `m′` is a value), leaving the build-time check to determine if the representation types are in fact compatible. Typing for a system state `N,F,e` requires that (1) all module values in the filesystem (which may have free channel or type names created earlier) have the signature they claim; (2) the process `e` is a well-typed `proc`; and (3) `N` has only channel and type names.

```
∀U∈dom(F). N⊢ F(U) : sig(F(U))
N⊢ e : proc
N atomic
N has no module bindings
────────────────────────────────system
⊢ N,F,e ok
```

Here `sig` is the function that extracts the signature of a module value, defined by

```
sig( [T,e] as [X::K,T′] ) = [X::K,T′]
sig( λU:SS.m:S′ )         = ΠU:SS.S′
```

and a type environment `E` is *atomic* if for all term variable bindings `x:T` in `E` there exists `T2` such that `T=T2 chan`.

3.2 Build and Run-Time Semantics

The language semantics is expressed with transitions

$$\texttt{N,F,e} \xrightarrow{\texttt{Com}} \texttt{N}',\texttt{F}',\texttt{e}'$$

labelled by commands `U:=m`, `run U`, and `tau`, expressing how the system state can change. The run-time semantics is straightforward, with transitions labelled by `tau` arising from π-calculus reductions `e` ⟶`e′`. It is defined in Appendix A.3. The build-time semantics, for transitions labelled `U:=m`, is novel. It involves an auxiliary one-step reduction relation for module expressions, discussed in §3.3, written

```
N⊢ m ⟶ New N′ in m′
```

for m reducing to m′ with new channel or type names N′. Note that the New N′ here is part of the judgement, not module syntax. Multistep reductions will be written with the double arrow $\Longrightarrow$. In addition, we identify various error cases. Run-time errors are simply mismatched communications, e.g. `x!3 | x?[y z]->e`. Build-time errors arise when trying `U:=m` for a badly-typed m, or when the dynamic check of a `with!` coercion within m fails. Load-time errors arise when trying `run U` where U is not the filename of a structure containing a process. We omit details of the error cases, but give the main transition rules:

```
typify(N),env(F) ⊢ m : S
N ⊢ F(m) ⟹ New N′ in mval
dom(N) disjoint from fv(m) and dom(N′)
---------------------------------------------- build
N,F,e --U:=m--> (N,N′),F ⊕ (U ↦ mval),e

F(U) = [T,e′] as [X::K,proc]
---------------------------------------------- load
N,F,e --run U--> N,F,e|e′

e ⟶ e′
---------------------------------------------- compute
N,F,e --tau--> N,F,e′
```

We discuss the key aspects of the `build` rule. Firstly, m is typechecked. It may mention previously-built modules in `dom(F)`, so this should be with respect to their signatures. These signatures may involve type variables in `dom(N)` previously generated for abstract types, but the representations of those types should not be visible for typechecking. We write `env(F)` for the type environment mapping each U ∈ `dom(F)` to `sig(F(U))`, and `typify(N)` for the type environment mapping each X ∈ `dom(N)` to `Type`. Secondly, `F(m)` – m with all filenames replaced by the module values they refer to – is reduced to `mval`. This may generate new type or channel names, which are propagated to the resulting state, and involves checking any coercions in m. Finally, the disjointness condition, and the fact that both the N part of `N,F,e` and the N′ part of `N ⊢ F(m) ⟹ New N′ in mval` are treated as binders, ensure that all previously-generated names are alpha-converted away from the free names of m. The full rules for whole systems can be found in Appendix A.4.

3.3 Module reduction

Module reduction is defined by the axioms in Figure 5, with (roughly) evaluation contexts

```
  C  ::=    _ | C m2 | mval C | C:SS
              | C with! mval  | m1 with! C
```

and rule

```
N⊢ m ⟶ New N′ in m′
dom(N′) and fv(C) disjoint
------------------------------
N⊢ C[m] ⟶ New N′ in C[m′]
```

The substitution in `m.red.beta` is nonstandard – it also reduces any projections from structures that are introduced by the substitution. Note the new type name generation of `m.red.abstype`, and the check in `m.red.with!` that the representation type T1 of m is provably equal to the representation type TZ of the module m′ that m is being coerced to. As in `build`, this must be with respect to `typify(N)`. (In fact, we have glossed over a subtlety – `m.red.abstype` should not be used on the left of a `with!`; we ensure this by splitting the reduction relation into two.) The full rules can be found in Appendix A.2.

The dynamic check does require the representation types of any modules that are referred to to be stored (as part of a struct that is inaccessible except to the `with!` check), but as the equality check is wrt. `typify(N)` no other type equalities from the development history are needed.

3.4 Examples

First, some examples of module reduction. For *C* as defined in §3.1.1, the structure

```
[int, 6] as [X::EQ(int), X]
```

is already a module value. For *A* there is a single reduction, creating a new type id, to a module value:

```
⊢ [int,6] as [X::Type,X] ⟶ New Z::EQ(int) in [Z,6] as [X::EQ(Z),X]   by m.red.abstype
```

Applying *Fopaque* to *C*:

```
⊢ Fopaque C ⟶ {C/U}[U.Type*U.Type,[U.term,U.term]] as [Y::Type,Y]     by m.red.beta
            =   [int*int,(6,6)] as [Y::Type,Y]
            ⟶ New Z::EQ(int*int) in [Z,(6,6)] as [Y::EQ(Z),Y]         by m.red.abstype
```

Applying *Fopaque* to *A*, first *A* is reduced (creating a new type id), then there is a beta step, then another new type id is created.

```
⊢ Fopaque A ⟶ New Z::EQ(int) in Fopaque ([Z,6] as [X::EQ(Z),X])
            ⟶ New Z::EQ(int) in {[Z,6] as [X::EQ(Z),X]/U}([U.Type*U.Type,(U.term,U.term)]
                                                             as [Y::Type,Y])
            =   New Z::EQ(int) in [Z*Z,(6,6)] as [Y::Type,Y]
            ⟶ New Z::EQ(int), W::EQ(Z*Z) in [W,(6,6)] as [Y::EQ(W),Y]
```

Now consider the build commands

```
C := C
A := A
```

(here `C` and `A` are module variables, used as filenames, and *C* and *A* are, as above, abbreviations for module expressions). Executing these in the empty state `New empty in (∅,0)` gives the state

```
New Z::EQ(int) in ({C↦C, A ↦ [Z,6] as [X::EQ(Z),X]},0)                    (*)
```

Subsequent build commands can then refer to the type and term components of `A` and `C`, eg in

```
B := [A.Type*C.Type,(A.term,C.term)] as [X::EQ(A.Type*C.Type),X]
```

The module expression on the right-hand side of this will be type checked in the environment

```
Z::Type, C:[X::EQ(int),X], A:[X::EQ(Z),X]
```

Rebuilding *A* will produce an abstract type that is different from that of `A` – executing `A′ := A` in the state `(*)` above results in

```
New Z::EQ(int),Z′::EQ(int) in ({C↦C, A ↦ [Z,6] as [X::EQ(Z),X],
                                     A′ ↦ [Z′,6] as [X::EQ(Z′),X]},0)
```

so executing `U := Fb A A′` will give a type checking error. On the other hand, executing `A′ := A with! A` in the state `(*)` results in

```
New Z::EQ(int) in ({C↦C, A ↦ [Z,6] as [X::EQ(Z),X],
                         A′ ↦ [Z,6] as [X::EQ(Z),X]},0)
```

and here `U := Fb A A′` will succeed. For a more interesting use of `with!`, illustrating the fact that only the representation types must be equal, executing

```
A′ := [int,(7,8)] as [X::Type,X*X] with! A
```

in the state `(*)` results in

```
New Z::EQ(int) in ({C↦C, A ↦ [Z,6] as [X::EQ(Z),X],
                         A′ ↦ [Z,(7,8)] as [X::EQ(Z),X*X]},0)
```

Finally, consider a module that provides an abstract type, implementing it with a representation type that involves an abstract type from another module, eg (again in state `(*)`)

```
A2 := [A.Type*A.Type,(A.term,A.term)] as [X::Type,X]
```

resulting in the state

```
New Z::EQ(int),W::EQ(Z*Z) in ({C↦C, A ↦ [Z,6] as [X::EQ(Z),X],
                                    A2 ↦ [W,(6,6) as [X::EQ(W),X]},0)
```

If one rebuilds `A2`

```
A2′ := m with! A2
```

the `m.red.with!` rule checks that the representation type of `m` is equal to `Z*Z`, in an environment `Z::Type,W::Type`. Simply checking that the representation type of `m` is equal to the underlying representation type `int*int` would be mistaken, as `A` may have been rebuilt, either with the same invariants (and using `with!`) or changed (and not using `with!`).

The examples above are artificial, with useless abstract types, to illustrate the semantics. They should be enough to see how natural examples (which would be unfortunately lengthy) would go, however.

3.5 Soundness

This subsection states only the main lemmas and the soundness properties of the semantics. Full details can be found in Appendix B.

Lemma 5 (Weakening for ⊢) `If E,E′′⊢ J and E,E′⊢ ok and dom E′, dom E′′ disjoint and if J is pat:T ▷ E1 then dom E′,var pat disjoint then E,E′,E′′⊢ J.`

Lemma 15 (Kind, type and sig well–formedness for ⊢)

```
If E⊢ K <:: K′          then  E⊢ K  and  E⊢ K′
If E⊢ K == K′           then  E⊢ K  and  E⊢ K′
If E⊢ T::K              then  E⊢ K
If E⊢ T == T′ :: K      then  E⊢ T::Type and E⊢ T′::Type
If E⊢ S < S′ sig        then  E⊢ S sig and E⊢ S′ sig
If E⊢ S == S′ sig       then  E⊢ S sig and E⊢ S′ sig
If E⊢ pat : T ▷ E′      then  E⊢ T::Type and E,E′⊢ ok
If E⊢ e:T               then  E⊢ T::Type
If E⊢ m:S               then  E⊢ S sig
```

The proof of this involves an alternative characterisation of the kind, type, signature and type environment formation judgements, removing the dependencies on (in)equality judgements.

Lemma 16 (Strengthening of ⊢ by term bindings) `If E,E′,E′′⊢ J and E′ has only term bindings and dom E′ not in J then E,E′′⊢ J.`

Lemma 17 (Substitution – type) If E,Z::KZ,E′⊢ J and E⊢ TZ::Type and KZ is either Type or EQ(TZ) then E,{TZ/Z}E′⊢ {TZ/Z}J.

Lemma 19 (Permutation) If E1,E2,E3,E4⊢ J and dom E2 not in E3 then E1,E3,E2,E4⊢ J.

Lemma 20 (Narrowing) If E,Z::KZ′,E′⊢ J and E⊢ KZ<::KZ′ then E,Z::KZ,E′⊢ J.

Lemma 25 (Cancellation – type equality) If E⊢ T == T′ :: Type and both T and T′ are of the form [T .. T] | chan T | proc | int (not necessarily the same) then either
1) T = [T1..Tn] and T′=[T1′..Tn′] and for i=1..n E⊢ Ti==Ti′::Type,
2) T = chan T1 and T′=chan T1′ and E⊢ T1==T1′::Type,
3) T = proc = T′, or
4) T = int = T′.
The proof of this involves an alternative characterisation of the type equality judgement, reducing it to simple equational reasoning.

Lemma 30 (Substitution – module)
If E,U:S,E′⊢ J and E⊢ mval:S then E,{mval/U}E′⊢ {mval/U}J.

Lemma 31 (Unique decomposition) If N has no module bindings and N⊢ m:S then either m is an mval xor there is a unique j∈{1,2}, Cj, rule r∈m.red.* and instance lr of the redex of r such that m=Cj[lr].

Lemma 35 (Module subject reduction)
If N⊢ m:S and N⊢ m ⟹ New N′ in m′ and N,N′ disjoint then N,N′⊢ m′:S.
Moreover if N atomic and N has no module bindings then N,N′ is likewise.

Lemma 36 (Well–formed term substitutions) If E atomic and has no module bindings, E⊢ e:T and E⊢ pat:T ▷ E′ then {e/pat}′ is defined and E⊢ {e/pat}′ : E′.

Lemma 37 (Substitution – term)
If E,E′,E′′⊢ e:T and E⊢ s:E′, where s is a term substitution, then E,E′′⊢ se : T.

Lemma 38 (Type soundness of structural congruence)
If E⊢ e:proc and e == e′ then E⊢ e′:proc.

Lemma 39 (Process subject reduction)
If E is atomic and has no module bindings, E⊢ e:proc and e⟶ e′ then E⊢ e′:proc.

Lemma 40 (Process soundness)
If E is atomic and has no module bindings and E⊢ e:proc then not (e $\xrightarrow{\texttt{err}}$).

Theorem 1 *If* ⊢ N,F,e ok *and* N,F,e $\xrightarrow{\texttt{Com}}$ N′,F′,e′ *then* ⊢ N′,F′,e′ ok.

Theorem 2 *If* ⊢ N,F,e ok *then there is no transition* N,F,e $\xrightarrow{\texttt{tau}}$ err(runtime error).

There may of course be build- or load-time errors.

3.6 Relating separately-compiled and monolithic programs

A desirable property of systems for separate compilation is that splitting a program into separate compilation units is guaranteed not to change its type-correctness or behaviour. For the language of this paper, a simple version of the property is:

Theorem 3 *If* m1 *and* m2 *are structure expressions*

```
m1 = [T1,e1] as [X::K1,T1′]
m2 = [T2,e2] as [X::K2,T2′]
```

and

```
m2′= (λU1:SS1.m2:SS2) [T1,e1] as [X::EQ(T1),T1′]
```

where SSi *is the signature of* mi*, then*

$$(\exists \texttt{N,F,e. empty,}\emptyset\texttt{,0} \overset{\texttt{U1:=m1}}{\longrightarrow} \overset{\texttt{U2:=m2}}{\longrightarrow} \texttt{N,F,e})$$

iff

$$(\exists \texttt{N,F,e. empty,}\emptyset\texttt{,0} \overset{\texttt{U2:=m2}'}{\longrightarrow} \texttt{N,F,e})$$

Note that one cannot take m2′ to be simply (λU1:SS1.m2:SS2) m1, as if K1=Type and U1 is used in T2′ this cannot generally be given a useful signature. Instead, the abstraction is enforced solely by the argument signature of the functor. The proof of the theorem is in Appendix B.11. The interesting case of the proof is the left-to-right direction for K1=Type – taking also K2≠Type, the execution of U1:=m1 involves new type name generation whereas the execution of U2:=m2′ does not. We therefore have a tight link between the generative view of abstract types, used in the separately-compiled version, and the singleton-kind view, in typechecking m2′.

Intuitively, the two resulting module values F(U2) are syntactically the same modulo certain type equalities, but we do not make this precise here. One might also generalise the result, giving translations between arbitrary module expressions and sequences of structure and functor build commands. This could then be contrasted to the result of Leroy [Ler96] relating the expressiveness of manifest-type-based and stamp-based static semantics for ML-style modules. The latter involves new stamp generation during type-checking (elaboration) of a module expression.

4 Conclusion

In summary, we have provided a solid basis for programming wide-area systems involving interaction at abstract types. It required new constructs – the versioning coercion and build-time channel generation – and novel operational semantics for module reduction and for build/load/compute-time system state changes. We illustrated these, demonstrating how they can be set up coherently, by giving a model language of processes, modules and commands, equipped with build- and run-time semantics, and proving soundness.

The work is a necessary preliminary for more refined treatments of numbered versions, and for extending traditional distributed systems programming with communication of values of abstract types.

Further Related Work We argued in §2.3 that developers must – in a limited way – be able to break the abstraction boundary of an abstract type. The with! coercion does this, allowing new types to be made compatible with old, provided (a) their immediate representation types are equal, and (b) the developer asserts no important invariants have changed. The closest primitive in previous work seems to be the partial revelations of opaque types in Modula 3 [CDG$^+$89], allowing any opaque type to be made concrete (to a specified subtype) within a scope. Turning to formal models, Cardelli [Car97] discusses linking and separate compilation in detail, but without module type components. The MTAL system of Glew and Morrisett [GM99] models linking for typed assembly language. It incorporates abstract types, but has a flat namespace and does not deal with differing versions. There is therefore no need for explicit generation of type names. Other work on separate compilation, notably [BA99, Dro00, Dug00, FF98, HWC00], focusses largely on name space and hierarchy issues, and on dynamic linking.

Future Directions Firstly, it would be interesting to generalise the results of §3.6, as indicated there, to arbitrary programs.

The idea of the module reduction semantics may have other applications. In particular, it should permit a typed operational semantics for Cardelli and Leroy's dot-notation calculi [CL90]. Adding term-level annotations delimiting subterms that originated from abstract types, building on [ZGM99], may allow nice syntactic statements of abstractness properties.

One might hope to address first class modules [HL94, Lil97, Rus98]. Typed semantics for dot-notation would go some way towards this, but note that term-level execution here may involve distributed communication, which may only make sense with respect to a local machine state. It is therefore desirable to be able to stratify execution (in the extreme case of the language of this paper no core reduction is done at build time, trivially preventing such distributed side-effects).

As for extensions, for usability the system would have to be extended to general dependent records and substructures, not simply binary translucent sums. Dealing with named interfaces would allow more direct treatment of traditional IDLs, extended with abstract types. There are many pragmatic issues of what development environment support is required to make the `with!` coercion and its channel-name analogue usable; they will have to be investigated by experiment. Finally, to address interface evolution and numbered versions, the system should be extended with subtyping, with polarity on the types used for interaction (here these are the channel types, for which polarities have been studied eg in [PS96, Ode95]), and with a type-level representation of a partial order.

Acknowledgements I would like to thank Luca Cardelli, James Leifer, Benjamin Pierce, Claudio Russo, Keith Wansbrough, Lucian Wischik, and Paweł Wojciechowski, for discussions and comments.

A Full Language Definition

This appendix contains the full typing rules, build-time semantics and run-time semantics of the language.

A.1 Typing

$\boxed{E\vdash K}$

$$\frac{E\vdash \texttt{ok}}{E\vdash \texttt{Type}}\,\texttt{K.Type} \qquad \frac{E\vdash \texttt{T :: Type}}{E\vdash \texttt{EQ(T)}}\,\texttt{K.EQ}$$

$\boxed{E\vdash K <:: K'}$

$$\frac{E\vdash \texttt{T :: Type}}{E\vdash \texttt{EQ(T) <:: Type}}\,\texttt{K<.EQ} \qquad \frac{E\vdash K == K'}{E\vdash K <:: K'}\,\texttt{K<.K=}$$

$\boxed{E\vdash K==K'}$

$$\frac{E\vdash \texttt{ok}}{E\vdash \texttt{Type == Type}}\,\texttt{K=.Type} \qquad \frac{E\vdash T==T'\ \texttt{:: Type}}{E\vdash \texttt{EQ(T) == EQ}(T')}\,\texttt{K=.T=}$$

$\boxed{E\vdash T::K}$

$$\frac{E\vdash T :: K \qquad E\vdash K <:: K'}{E\vdash T :: K'}\,\texttt{T.K<} \qquad \frac{E\vdash T == T'\ \texttt{:: Type}}{E\vdash T :: \texttt{EQ}(T')}\,\texttt{T.T=} \qquad \frac{E\vdash U{:}[X{::}K,T]}{E\vdash U.\texttt{Type} :: K}\,\texttt{T.Proj}$$

$$\frac{E\vdash \texttt{ok} \qquad E\vdash \texttt{Ti :: Type}\quad \texttt{i=1..n}}{E\vdash \texttt{[T1 .. Tn] :: Type}}\,\texttt{T.rec} \qquad \frac{E\vdash \texttt{T :: Type}}{E\vdash \texttt{T chan :: Type}}\,\texttt{T.chan} \qquad \frac{E\vdash \texttt{ok}}{E\vdash \texttt{proc :: Type} \quad E\vdash \texttt{int :: Type}}\,\texttt{T.proc T.int}$$

$$\frac{E,X{::}K,E'\vdash\ \texttt{ok}}{E,X{::}K,E'\vdash X{::}K}\,\texttt{T.var}$$

$\boxed{E\vdash T == T'\ :: K}$

$$\frac{E\vdash T :: \texttt{EQ}(T')}{E\vdash T == T'\ \texttt{:: Type}}\,\texttt{T=.EQ}$$

and standard rules T=.refl,T=.sym,T=.tran,T=.cong.rec,T=.cong.chan for equivalence and congruence

$\boxed{E\vdash S\ \texttt{sig}}$

$$\frac{E,X{::}K\ \vdash T{::}\texttt{Type}}{E\vdash [X{::}K,T]\quad \texttt{sig}}\,\texttt{S.Struct} \qquad \frac{E\vdash SS\ \texttt{sig} \quad E,U{:}SS\vdash S'\ \texttt{sig}}{E\vdash \Pi U{:}SS.S'\ \texttt{sig}}\,\texttt{S.Fun}$$

$E\vdash S < S' \ sig$

$$\frac{E\vdash K <:: K' \qquad E,X::K\vdash T == T' :: Type}{E\vdash [X::K,T] < [X::K',T'] \ sig}\ \mathrm{S<.Struct}$$

$$\frac{E\vdash SS1' < SS1 \ sig \quad E,U:SS1'\vdash S2 < S2' \ sig}{E\vdash \Pi U:SS1.S2 < \Pi U:SS1'.S2' \ sig}\ \mathrm{S<.Fun}$$

$$\frac{E\vdash S == S' \ sig}{E\vdash S < S' \ sig}\ \mathrm{S<.S=} \qquad \frac{E\vdash S<S' \ sig \quad E\vdash S'<S'' \ sig}{E\vdash S<S'' \ sig}\ \mathrm{S<.tran}$$

$E\vdash S == S' \ sig$

$$\frac{E\vdash K==K' \qquad E,X::K \vdash T == T' :: Type}{E\vdash [X::K,T] == [X::K',T'] \ sig}\ \mathrm{S=.Struct}$$

$$\frac{E\vdash SS1 == SS1' \ sig \quad E,U:SS1'\vdash S2 == S2' \ sig}{E\vdash \Pi U:SS1.S2 == \Pi U:SS1'.S2' \ sig}\ \mathrm{S=.Fun}$$

and standard rules S=.refl, S=.sym, S=.tran for equivalence

$E\vdash ok$

$$\mathrm{E.empty}\ \frac{}{empty\vdash ok} \qquad \mathrm{E.term}\ \frac{E\vdash T :: Type \quad x \text{ not in dom } E}{E,x:T\vdash ok} \qquad \mathrm{E.type}\ \frac{E\vdash K \quad X \text{ not in dom } E}{E,X::K\vdash ok} \qquad \mathrm{E.mod}\ \frac{E\vdash S \ sig \quad U \text{ not in dom } E}{E,U:S\vdash ok}$$

$E\vdash pat : T \rhd E'$

$$\frac{E\vdash T::Type \quad x \text{ not in dom } E}{E\vdash x : T \rhd x:T}\ \mathrm{pat.var} \qquad \frac{E\vdash ok \quad E\vdash pati : Ti \rhd Ei \quad i=1..n \quad pati \text{ all disjoint}}{E\vdash [pat1..patn] : [T1..Tn] \rhd E1,..,En}\ \mathrm{pat.rec}$$

$E\vdash e:T$

$$\frac{E,x:T,E'\vdash ok}{E,x:T,E'\vdash x:T}\ \mathrm{e.var} \qquad \frac{E\vdash e:T \quad E\vdash T==T' :: Type}{E\vdash e:T'}\ \mathrm{e.T=} \qquad \frac{E\vdash ok \quad E\vdash ei:Ti \ i=1..n}{E\vdash [e1 .. en]:[T1 .. Tn]}\ \mathrm{e.rec}$$

$$\frac{E\vdash U : [X::K,T] \quad E\vdash T :: Type}{E\vdash U.term : T}\ \mathrm{e.proj} \qquad \frac{E,x: T\ chan \vdash e : proc}{E\vdash new\ x: T\ chan\ in\ e : proc}\ \mathrm{e.new} \qquad \frac{E\vdash ok}{E\vdash 0 : proc}\ \mathrm{e.nil}$$

$$\frac{E\vdash ei:proc \ i=1,2}{E\vdash e1|e2 : proc}\ \mathrm{e.par} \qquad \frac{E\vdash e1 : T\ chan \quad E\vdash e2 : T}{E\vdash e1!e2 : proc}\ \mathrm{e.out} \qquad \frac{E\vdash e1 : T\ chan \quad E\vdash pat:T \rhd E' \quad E,E' \vdash e2:proc}{E\vdash e1?pat \Rightarrow e2 :proc}\ \mathrm{e.in}$$

$$\frac{E\vdash ok}{E\vdash \underline{n} : int}\ \mathrm{e.int}$$

```
E⊢ m:S
```

```
                            E⊢ m:S
E,U:S,E′⊢ ok                E⊢ S < S′ sig            E,x : T chan ⊢ m : SS
------------------m.var     ------------------m.S<   ------------------------------m.new
E,U:S,E′⊢ U:S               E⊢ m:S′                  E⊢ new x : T chan in m : SS
```

```
E⊢ T :: K
E,X::K⊢ T′ :: Type
E,X::EQ(T) ⊢ e:T′                                    E⊢ U : [X::K,T]
X not free in e                                      E⊢ U.Type :: K′
-------------------------------------m.struct        ------------------m.self−type
E⊢ [T,e] as [X::K,T′] : [X::K,T′]                    E⊢ U : [X::K′,T]
```

```
                                          E⊢ m:ΠU:SS1.S2
                                          E⊢ m′:SS1
E,U:SS1⊢ m:S2                             U not free in S2
-----------------------------m.fun        ------------------m.app
E⊢ λU:SS1.m:S2 : ΠU:SS1.S2                 E⊢ m m′ : S2
```

```
                        E⊢ m :[X::K,T2]
E⊢ m:SS                 E⊢ m′:[Y::EQ(T′),T2′]
-------------m.seal     -----------------------------m.with
E⊢ (m:SS):SS            E⊢ (m with! m′) : [X::EQ(T′),T2]
```

```
⊢ N,F,e ok
```

```
forall U in dom(F). N⊢ F(U) : sig(F(U))
N⊢ e : proc
N atomic
N has no module bindings
---------------------------------------------system
⊢ N,F,e ok
```

A.2 Module Reduction

```
N⊢ m ⟶ᵢ New N′ in m′   i=1,2        N⊢ m --err--> s

N⊢ m ⟹ New N′ in m′                  N⊢ m ==err==> s
```

Here the (1) reductions do not do outermost new type creation, whereas the (2) reductions do. Note that the New here is part of the judgement, not module syntax. We work up to alpha conversion of N′ binding in m′. We elide ‘New empty in’.

```
N⊢ (λU:SS1.m:S2) mval             ⟶1 {mval/U}m                                    m.red.beta
N⊢ [T1,e] as [X::EQ(T1′),T2] : SS  ⟶1 [T1,e] as SS                                m.red.seal
N⊢ new x:T in m                    ⟶1 New x:T       in m                          m.red.new
N⊢ [T1,e] as [X::Type,T2]          ⟶2 New Y::EQ(T1) in [Y,e] as [X::EQ(Y),T2]    m.red.abstype
 (for Y ∉ dom(N))
```

```
m =[T1,e ] as [X::Type, T2 ]
m′=[Z, e′] as [Y::EQ(Z),T2′]
N =N1,Z::EQ(TZ),N2
typify(N)⊢ T1==TZ::Type
------------------------------------------ m.red.with!
N⊢ m with! m′ ⟶1  [Z,e] as [X::EQ(Z),T2]
```

```
m =[T1,e ] as [X::Type, T2 ]
m′=[Z, e′] as [Y::EQ(Z),T2′]
N(Z)=EQ(TZ)
not( typify(N)⊢ T1==TZ::Type )
------------------------------------------------------------------------- m.red.err.with!1
N⊢ m with! m′ --err--> with! failed — representation types not equal

m =[T1, e ] as [X::Type,    T2 ]
m′=[T1′,e′] as [Y::EQ(T1′′),T2′]
not exists Z,TZ such that  T1′ =T1′′=Z  and  N(Z)=EQ(TZ)
------------------------------------------------------------------------- m.red.err.with!2
N⊢ m with! m′ --err--> with! failed — can only with! to an abstract type

m =[T1, e ] as [X::EQ(T1′′′), T2 ]
m′=[T1′,e′] as [Y::EQ(T1′′),  T2′]
------------------------------------------------------------------------- m.red.err.with!3
N⊢ m with! m′ --err--> with! failed — can only apply with! to an abstract
```

```
We omit definitions of errors for bogus applications etc.

Evaluation contexts:

A1   ::=   _ m2   |   mval _   |  _ with! mval  |  m1 with! _  |   _:SS
A2   ::=   _ m2   |   mval _   |                   m1 with! _  |   _:SS

C1   ::=  _   |  A1[C1]        (for the (1) reductions)
C2h  ::=  A2  |  A1[C2h]
C2   ::=  _   |  C2h           (for the (2) reductions)
```

```
                                         N⊢ m ⟶i New N′ in m′
                                         dom(N′) and fv(Ci) disjoint
------------------ m.wred.refl           ------------------------------ m.wred.red
N⊢ m ⟹ m                                 N⊢ Ci[m] ⟹ New N′ in Ci[m′]


N⊢    m ⟹ New N′ in m′
N,N′⊢ m′ ⟹ New N′′ in m′′
N N′ and N′′ have disjoint domains
---------------------------------- m.wred.tran
N⊢ m ⟹ New N′,N′′ in m′′

                                           N⊢ m ⟹ New N′ in m′
                                           N,N′⊢ m ==err==> s
N⊢ m --err--> s                            N and N′ have disjoint domains
-------------------- m.werr.err            ------------------------------ m.werr.tran
N⊢ C1[m] ==err==> s                        N⊢ m ==err==> s
```

A.3 Process reduction

```
e ⟶ e′     e --err-->
```

```
Define a partial function {_/_}′ taking a core expression and pattern
(in which all variables are distinct) and giving a substitution:

{e/x}′                         = {e/x}
{[e1 .. en]/[pat1 .. patn]}′ = {e1/pat1}′ ∪ ... ∪ {en/patn}′
{e/pat}′                       undefined otherwise

Define a structural equivalence == over core expressions to be the
```

least relation generated by axioms:

```
0|e1 == e1  sc.id    e1|e2 == e2|e1 sc.com    e1|(e2|e3) == (e1|e2)|e3 sc.ass
e1| new x:T in e2 == new x:T in e1|e2  if x not free in e1          sc.ext
```

and standard rules sc.cong.par, sc.cong.new, sc.refl, sc.sym, sc.tran.
Note it is important not to have congruence rules for tuples, out, or in.

$$\frac{\texttt{\{e1/pat\}}' \texttt{ defined}}{\texttt{x!e1 | x?pat->e2} \longrightarrow \texttt{\{e1/pat\}}'\texttt{e2}}\ \texttt{e.red.comm}$$

$$\frac{\texttt{e1} \longrightarrow \texttt{e1}'}{\texttt{e1|e2} \longrightarrow \texttt{e1}'\texttt{|e2}}\ \texttt{e.red.par} \qquad \frac{\texttt{e1} \longrightarrow \texttt{e1}'}{\texttt{new x:T in e1} \longrightarrow \texttt{new x:T in e1}'}\ \texttt{e.red.res}$$

$$\frac{\texttt{e2 == e1} \longrightarrow \texttt{e1}' \texttt{ == e2}'}{\texttt{e2} \longrightarrow \texttt{e2}'}\ \texttt{e.red.sc}$$

$$\frac{\texttt{\{e1/pat\}}' \texttt{ not defined}}{\texttt{x!e1 | x?pat->e2} \stackrel{\texttt{err}}{\longrightarrow}}\ \texttt{e.err.comm} \qquad \frac{\texttt{e2 == e1} \stackrel{\texttt{err}}{\longrightarrow}}{\texttt{e2} \stackrel{\texttt{err}}{\longrightarrow}}\ \texttt{e.err.sc}$$

$$\frac{\texttt{e1} \stackrel{\texttt{err}}{\longrightarrow}}{\texttt{e1|e2} \stackrel{\texttt{err}}{\longrightarrow}}\ \texttt{e.err.par} \qquad \frac{\texttt{e1} \stackrel{\texttt{err}}{\longrightarrow}}{\texttt{new x:T in e1} \stackrel{\texttt{err}}{\longrightarrow}}\ \texttt{e.err.res}$$

A.4 Whole System

$$\boxed{\texttt{N,F,e} \stackrel{\texttt{Com}}{\longrightarrow} \texttt{N}',\texttt{F}',\texttt{e}'}$$

$$\frac{\begin{array}{l}\texttt{typify(N),env(F)} \vdash \texttt{m : S} \\ \texttt{N} \vdash \texttt{F(m)} \Longrightarrow \texttt{New N}' \texttt{ in mval} \\ \texttt{dom(N) disjoint from fv(m) and dom(N}'\texttt{)}\end{array}}{\texttt{N,F,e} \stackrel{\texttt{U:=m}}{\longrightarrow} \texttt{(N,N}'\texttt{),F} \oplus (\texttt{U} \mapsto \texttt{mval}),\texttt{e}}\ \texttt{build}$$

$$\frac{\texttt{F(U) = [T,e}'\texttt{] as [X::K,proc]}}{\texttt{N,F,e} \stackrel{\texttt{run U}}{\longrightarrow} \texttt{N,F,e|e}'}\ \texttt{load} \qquad \frac{\texttt{e} \longrightarrow \texttt{e}'}{\texttt{N,F,e} \stackrel{\texttt{tau}}{\longrightarrow} \texttt{N,F,e}'}\ \texttt{compute}$$

$$\boxed{\texttt{N,F,e} \stackrel{\texttt{Com}}{\longrightarrow} \texttt{err(s)}}$$

$$\frac{\begin{array}{l}\texttt{not (typify(N),sigify(F)} \vdash \texttt{m : S)} \\ \texttt{dom(N) not free in m}\end{array}}{\texttt{N,F,e} \stackrel{\texttt{U:=m}}{\longrightarrow} \texttt{err(source doesn}'\texttt{t typecheck)}}\ \texttt{build.err.1}$$

$$\frac{\begin{array}{l}\texttt{typify(N),sigify(F)} \vdash \texttt{m : S} \\ \texttt{dom(N) not free in m} \\ \texttt{N} \vdash \texttt{F(m)} \stackrel{\texttt{err}}{\Longrightarrow} \texttt{s}\end{array}}{\texttt{N,F,e} \stackrel{\texttt{U:=m}}{\longrightarrow} \texttt{err(s)}}\ \texttt{build.err.2} \qquad \frac{\texttt{e} \stackrel{\texttt{err}}{\longrightarrow}}{\texttt{N,F,e} \stackrel{\texttt{tau}}{\longrightarrow} \texttt{err(runtime error)}}\ \texttt{compute.err}$$

$$\frac{\texttt{F(U) not of the form [T,e}'\texttt{] as [X::K,proc]}}{\texttt{N,F,e} \stackrel{\texttt{run U}}{\longrightarrow} \texttt{err(cannot run a non-proc)}}\ \texttt{run.err.1} \qquad \frac{\texttt{U not in dom F}}{\texttt{N,F,e} \stackrel{\texttt{run U}}{\longrightarrow} \texttt{err(file not found)}}\ \texttt{run.err.2}$$

B Metatheory

This appendix gives the proofs of the results stated in §3.5 and §3.6.

The non-standard substitution {`mval/U`} of an mval for a module variable (used in m.red.beta) is defined by induction over each syntactic category `m,S,e,T,K,E`. If `mval = [T1,e] as [X::EQ(T1'),T2]` simultaneously replace

```
U.Type by T1
U.term by e
U      by  [T1,e] as [X::EQ(T1′),T2] in all other contexts
```

If `mval = λU:S.m:S′` then {`mval/U`} is standard substitution.

The auxiliary functions `typify`, `sig` and `env` are defined as follows.

```
typify(empty)=empty          typify(E,X::K)=typify(E),X::Type
typify(E,x:T)=typify(E)      typify(E,U:S) =typify(E)

sig( [T,e] as [X::K,T′] ) = [X::K,T′]
sig( λU:SS.m:S′ )     = ΠU:SS.S′

env(empty)=empty
env(F,(U,mval)) = env(F), U:sig(mval)
```

Recall that we are not considering judgements `E⊢ J` up to renaming of the variables declared in `E` (as we use module variables for filenames). We do, though, use without proof the fact that if `E⊢ J` and `E′,J′` is a bijective renaming of `E,J` then `E′⊢ J′`. Strictly speaking, we are often doing inductions over renaming-equivalence-classes of derivations, but making that explicit would not add useful rigor. The most routine inductions and cases are omitted throughout.

B.1 Simple admissible rules, ok-ness and weakening

Lemma 1 `The rules below are admissible.`

```
                                    E⊢ K==K′             E⊢ K <:: K′
E⊢ K                E⊢ K==K′        E⊢ K′==K′′           E⊢ K′ <:: K′′
─────────K=.refl    ─────────K=.sym ───────────K=.tran   ──────────────K<.tran
E⊢ K==K             E⊢ K′==K        E⊢ K==K′′            E⊢ K <:: K′′

E⊢ K                  E⊢ T::K            E⊢ U : [X::EQ(T),T′]
──────────────K<.top  ──────────────(*)  ───────────────────────T=.abbrev
E⊢ K <:: Type         E⊢ EQ(T)<::K       E⊢ U.Type == T :: Type
```

`It is also worth noting that an m.self−term rule is admissible.`

Lemma 2 `The rule below is admissible.`

```
E⊢ U : [X::K,T]
E⊢ U.term : T′
───────────────m.self−term
E⊢ U : [X::K,T′]
```

Proof

`Lemma If  E⊢ U.term : T′  then for some X,K′′,T′′ we have  E⊢ U : [X::K′′,T′′]`

```
    and E⊢ T′==T′′ :: Type.
    Proof Simple ind on E⊢ e:T, with only e.proj and e.T= non—vacuous.

By this Lemma and the second premise we have
E⊢ U : [X::K′′,T′′]          b
E⊢ T′==T′′ :: Type.          c

By the first premise and T.Proj    E⊢ U:Type :: K    d
By (b,d) and m.self—type           E⊢ U : [X::K,T′′]
By (c)  by various rules           E⊢ [X::K,T′′] < [X::K,T′] sig
By m.S<                            E⊢ U : [X::K,T′]
```

Lemma 3 If E⊢ S < S′ sig or E⊢ S==S′ sig then S is a struct sig iff S′ is.

Lemma 4 (ok–ness for ⊢) If E,E′⊢ J then E⊢ ok.

Lemma 5 (Weakening for ⊢) If E,E′′⊢ J and E,E′⊢ ok and dom E′, dom E′′ disjoint and if J is pat:T ▷ E1 then dom E′,var pat disjoint then E,E′,E′′⊢ J.

B.2 Well-formedness

To prove that in any derivable judgement E⊢J all kinds, types and signatures in J are well-formed, we define an auxiliary characterisation of the formation judgements. This makes explicit the fact that formation does not depend on the precise kind or signature assumptions in the type environment, and hence (Lemmas 11, 12) that they can be arbitrarily changed. One can then prove Lemma 15; in particular the cases for S < S′ and S==S′.

```
E⊢′ K

E⊢′ ok                  E⊢′ T :: Type
---------K′.Type        ------------------K′.EQ
E⊢′ Type                E⊢′ EQ(T)
```

```
E⊢′ T::Type

E⊢′ ok
E⊢′ Ti :: Type  i=1..n         E⊢′ T :: Type               E⊢′ ok
-----------------------T′.rec  ------------------T′.chan   ---------------T′.proc T′.int
E⊢′ [T1 .. Tn] :: Type         E⊢′ T chan :: Type          E⊢′ proc :: Type
                                                           E⊢′ int :: Type

E,X::K,E′⊢′  ok                E,U:[X::K,T],E′⊢′  ok
-------------------T′.var      ------------------------------------T′.Proj
E,X::K,E′⊢′ X::Type            E,U:[X::K,T],E′⊢′ U.Type :: Type
```

```
E⊢′ S sig

E,X::K ⊢′ T::Type               E⊢′ SS sig  E,U:SS⊢′ S′ sig
-------------------S′.Struct    ----------------------------S′.Fun
E⊢′ [X::K,T]   sig              E⊢′ ΠU:SS.S′ sig
```

`E⊢′ ok`

E′.empty	E′.term	E′.type	E′.mod
	E⊢′ T :: Type x not in dom E	E⊢′ K X not in dom E	E⊢′ S sig U not in dom E
empty⊢′ ok	E,x:T⊢′ ok	E,X::K⊢′ ok	E,U:S⊢′ ok

Lemma 6 (ok–ness for ⊢′) If E,E′⊢′ J then E⊢′ ok.

Lemma 7 (Weakening for ⊢′) If E,E′′⊢′ J and E,E′⊢ ok and dom E′, dom E′′ disjoint then E,E′,E′′⊢′ J.

Lemma 8 (Strengthening for ⊢′) If E,E′,E′′⊢′ J and dom E′ not in ran E′′ or in J then E,E′′⊢′ J.

Lemma 9 (Kind well–formedness in environments)
a) If E,X::K,E′⊢ ok then E,X::K,E′⊢ K.
b) If E,U:[X::K,T],E′⊢ ok then E,U:[X::K,T],E′⊢ K.

Proof
a) By 4 E,X::K⊢ ok, which must be derived by E.type, so E⊢ K. By 5 E,X::K,E′⊢ K.
b) By 4 E,U:[X::K,T]⊢ ok, which must be derived by E.mod, so E⊢ [X::K,T] sig, which must be derived by S.Struct, so (WLG taking X not in dom E) E,X::K⊢ T::Type. By 4 E,X::K⊢ ok, which must be derived by E.type, so E⊢ K. By 5 E,U:[X::K,T],E′⊢ K.

Lemma 10 (⊢′ implies ⊢)
If E⊢′ K then E⊢ K
If E⊢′ T::Type then E⊢ T::Type
If E⊢′ S sig then E⊢ S sig
If E⊢′ ok then E⊢ ok

Proof
[T′.var**]** By ind and T.var E,X::K,E′⊢ X::K. By 9, 1 and T.K< E,X::K,E′⊢X::Type.
[T′.Proj**]** By ind, m.var, T.Proj, 9, 1 and T.K<.

Lemma 11 (Preservation of ⊢′ by kind changes)
If E,X::K,E′⊢′ J and E⊢′ K′ then E,X::K′,E′⊢′ J

Lemma 12 (Preservation of ⊢′ by struct sig changes)
If E,U:SS,E′⊢′ J and E⊢′ SS′ sig then E,U:SS′,E′⊢′ J

Lemma 13 (⊢ implies ⊢′)

```
If E⊢ K              then  E⊢′ K
If E⊢ K <:: K′       then  E⊢′ K and E⊢′ K′
If E⊢ K == K′        then  E⊢′ K and E⊢′ K′
If E⊢ T::K           then  E⊢′ T::Type and E⊢′ K
If E⊢ T == T′ :: K   then  E⊢′ T::Type and E⊢′ T′::Type
If E⊢ S sig          then  E⊢′ S sig
If E⊢ S < S′ sig     then  E⊢′ S sig and E⊢′ S′ sig
If E⊢ S == S′ sig    then  E⊢′ S sig and E⊢′ S′ sig
If E⊢ ok             then  E⊢′ ok
If E⊢ U:[X::K,T]     then  E⊢′ K and E=E1,U:SS,E2 for some E1,SS,E2
```

Proof
[T.Proj**]** By ind E⊢′ K and E equals some E1,U:SS,E2. By 6 E⊢′ ok.
By T′.Proj E⊢′ U.Type :: Type
[T.var**]** Consider E=E1,X::K,E2 and E⊢ X::K. By ind E⊢′ ok. By T′.var E⊢′ X::Type.
By 6 E1,X::K⊢′ ok, which must be derived by E′.type, so E′⊢′ K. By 7 E,X::K,E′⊢′ K.
[S<.Struct**]** By ind E⊢′ K and E⊢′ K′ and E,X::K⊢′ T :: Type and E,X::K⊢′ T′ :: Type.
By S′.Struct E⊢′ [X::K,T] sig. By 11 E,X::K′⊢′ T′ :: Type.
By S′.Struct E⊢′ [X::K′,T′] sig.
[S<.Fun**]** By ind E⊢′ SS1′ sig and E⊢′ SS1 sig and E,U:SS1′⊢ S2 sig and E,U:SS1′⊢ S2′ sig.
By S′.Fun E⊢′ ΠU:SS1′.S2′ sig. By 12 E,U:SS1⊢ S2 sig.
By S′.Fun E⊢′ ΠU:SS1.S2 sig.
[m.var**]** Consider E=E1,U:[X::K,T],E2 and E⊢ U:[X::K,T]. By ind E⊢′ ok
By 6 E1,U:[X::K,T]⊢ ok, which must be derived by E′.mod, so
E1⊢′ [X::K,T] sig, which must be derived by S′.Struct, so (WLG taking X
not in dom E) E1,X::K⊢′ T::Type. By 6 E1,X::K⊢′ ok, which must be
derived by E′.type, so E1⊢′ K. By 7 E1,U:[X::K,T],E2⊢ K.
[m.S<**]** Say S′=[X::K′,T′]. By 3 S is a struct sig, say [X::K,T].
By ind E=E1,U:SS,E2 for some E1,SS,E2. By ind E⊢′ [X::K′,T′] sig
This must be by S′.Struct, so (WLG taking X not in dom E) E,X::K′⊢′ T′::Type.
By 6 E,X::K′⊢′ ok, which must be by E′.type, so E⊢′ K′.

Lemma 14 (Strengthening for ⊢ formation judgements) For J one of
K, T::Type, S sig and ok, if E,E′,E′′⊢ J and
dom E′ not in ran E′′ or in J then E,E′′⊢ J.
Proof This follows from the result 8 for ⊢′ and Lemmas 13,10.

Lemma 15 (Kind, type and sig well–formedness for ⊢)

```
If E⊢ K <:: K′         then  E⊢ K  and  E⊢ K′
If E⊢ K == K′          then  E⊢ K  and  E⊢ K′
If E⊢ T::K             then  E⊢ K
If E⊢ T == T′ :: K     then  E⊢ T::Type and E⊢ T′::Type
If E⊢ S < S′ sig       then  E⊢ S sig and E⊢ S′ sig
If E⊢ S == S′ sig      then  E⊢ S sig and E⊢ S′ sig
If E⊢ pat : T ▷ E′     then  E⊢ T::Type and E,E′⊢ ok
If E⊢ e:T              then  E⊢ T::Type
If E⊢ m:S              then  E⊢ S sig
```

Proof

E⊢ K <:: K' | E⊢ K == K' | E⊢ T::K | E⊢ T == T' :: K | E⊢ S<S' sig | E⊢S==S' sig
All immediate from 13 and 10.

E⊢ pat : T ▷ E' | E⊢ e:T Routine.

E⊢ m:S Induction on derivations:
[m.var] By 4 E,U:S⊢ ok, which must be by E.mod, so E⊢ S sig,
so by 5 E,U:S,E'⊢ S sig.
[m.S<] By the S < S' part.
[m.struct] By S.Struct.
[m.self−type] By ind E⊢ [X::K,T] sig, so (WLG X) E,X::K⊢ T::Type,
so by 13 E,X::K⊢' T::Type. Also by 13 E⊢' K'. By 11 E,X::K'⊢' T::Type, so by
S'.Struct E⊢' [X::K',T] sig and by 10 E⊢ [X::K',T] sig.
[m.fun] By ind and S.Fun.
[m.app] By ind E⊢ ΠU:SS1.S2 sig, so (WLG U) E,U:SS1⊢ S2 sig. By 14 E⊢ S2 sig.
[m.seal] By ind.
[m.with] By ind E⊢ [X::K,T] sig, so (WLG X) E,X::K⊢ T::Type, so by 13 E,X::K⊢' T::Type.
Also by ind E⊢ [Y::EQ(T'),T2'] sig, so E⊢' EQ(T').
By 11 E,X::EQ(T')⊢' T2::Type, so by S'.Struct E⊢' [X::EQ(T'),T2] sig
and by 10 E⊢ [X::EQ(T'),T2] sig.
[m.new] By ind and 14.

B.3 Strengthening, Permutation, and Narrowing

Lemma 16 (Strengthening of ⊢ by term bindings) If E,E',E''⊢ J and E'
has only term bindings and dom E' not in J then E,E''⊢ J.

Lemma 17 (Substitution – type) If E,Z::KZ,E'⊢ J and E⊢ TZ::Type
and KZ is either Type or EQ(TZ) then E,{TZ/Z}E'⊢ {TZ/Z}J.

Lemma 18 (Strengthening of ⊢ by type bindings) If E,X::K,E'⊢ J and
X not in E' or J then E,E'⊢ J.
Proof Immediate from 17, taking cases of K=EQ(T) or (K=Type and T=proc).

Lemma 19 (Permutation) If E1,E2,E3,E4⊢ J and dom E2 not in ran E3
then E1,E3,E2,E4⊢ J.
Proof The only remotely interesting cases are for E⊢ ok – we give just one.
[E.type] We have E1,E2,E3,E4 = E,X::K.
Case E4=empty and E3=empty. Trivial.
Case E4=E4',X::K. By ind E1,E3,E2,E4'⊢ K, so by E.type E1,E3,E2,E4⊢ok.
Case E4=empty and E3=E3',X::K, ie E1,E2,E3'⊢ K.
By ind E1,E3',E2⊢ K. By 4 E1,E3',E2⊢ ok. By 14 E1,E3'⊢ K,
so by E.type E1,E3',X::K⊢ ok. By 5 E1,E3',X::K,E2⊢ ok.

Lemma 20 (Narrowing) If E,Z::KZ′,E′⊢ J and E⊢ KZ<::KZ′
then E,Z::KZ,E′⊢ J.

Proof

[T.var**]** Consider cases of X::K before, equal to, or after Z::KZ′.
Before and after are by ind. For equal to, by ind and T.var E,Z::KZ,E′⊢ Z::KZ.
By weakening 5 E,Z::KZ,E′⊢ KZ<::KZ′. Then use T.K<.
[E.type**]** Consider cases of E′ empty or E′=E1′,X::K

Lemma 21 (Narrowing and Weakening by typify) If typify(E)⊢J and E⊢ ok then E⊢J.

Proof We prove by ind on E′ that E,typify(E′)⊢ J and E,E′⊢ ok implies E,E′⊢J
(in fact using a reversed characterisation of typify).
[E,(empty)**]** Trivial.
[E,(X::K,E′)**]** We have E,X::Type,typify(E′)⊢ J and E,X::K,E′⊢ ok
By formation E⊢ K, so E⊢ K <:: Type, so by 20 E,X::K,typify(E′)⊢ J,
so by ind (as also (E,X::K),E′⊢ ok) E,X::K,E′⊢ J.
[E,(U:S,E′)**]** We have E,typify(E′)⊢ J and E,U:S,E′⊢ ok.
By formation E⊢ S sig, so by weakening 5 E,U:S,typify(E′)⊢J. By ind E,U:S,E′⊢ J.
[E,(x:T,E′)**]** Similar.

B.4 Type cancellation

To prove a cancellation lemma (25) for type equality we define another auxiliary characterisation, of E⊢T==T′::Type judgements, in terms of plain equational logic. This removes the need to chase through the kinding, signature and module judgements in the proof of the cancellation lemma.

$$\frac{\texttt{E,X::EQ(T),E}'\vdash'\texttt{ ok}}{\texttt{E,X::EQ(T),E}'\vdash'\texttt{ X == T :: Type}}\,\texttt{T=}'\texttt{.var}$$

$$\frac{\texttt{E,U:[X::EQ(T1),T2],E}'\vdash'\texttt{ ok}}{\texttt{E,U:[X::EQ(T1),T2],E}'\vdash'\texttt{ U.Type == T1 :: Type}}\,\texttt{T=}'\texttt{.abbrev} \qquad \frac{\texttt{E}\vdash'\texttt{ T :: Type}}{\texttt{E}\vdash'\texttt{ T==T::Type}}\,\texttt{T=}'\texttt{.refl}$$

together with rules T=′.sym, T=′.tran, T=′.cong.rec, T=′.cong.chan

Lemma 22

If E⊢ K <:: K′ then K=Type implies K′=Type
If E⊢ K==K′ then (K=Type iff K′=Type)
If E⊢ [X::K,T] < [X::K′,T′] sig then K=Type implies K′=Type
If E⊢ [X::K,T] == [X::K′,T′] sig then (K=Type iff K′=Type)

Lemma 23 (⊢ implies ⊢′ − type equality)

If E⊢ EQ(T) <::EQ(T′)	then	E⊢′ T==T′::Type
If E⊢ EQ(T) == EQ(T′)	then	E⊢′ T==T′::Type
If E⊢ T::EQ(T′)	then	E⊢′ T==T′::Type
If E⊢ T == T′ :: K	then	E⊢′ T==T′::Type
If E⊢ [X::EQ(T1),T2] < [X::EQ(T1′),T2′] sig	then	E⊢′ T1==T1′::Type
If E⊢ [X::EQ(T1),T2] == [X::EQ(T1′),T2′] sig	then	E⊢′ T1==T1′::Type
If E⊢ U:[X::EQ(T1),T2]	then	E⊢′ U.Type==T1::Type

Proof

[T.K<] By 22 we can take K=EQ(T1) and K=EQ(T1′), then by ind E$\vdash'$ T==T1::Type and E$\vdash'$ T1==T1′::Type, so by T=′.tran E$\vdash'$T==T1′::Type.
[T.var] By 13 and T=′.var.
[T=.refl] By 13 and T′=.refl
[S<.tran] By 22 and ind and T′=.tran.
[m.var] By 13 E,U:[X::EQ(T1),T2],E′$\vdash'$ ok, then use T=′.abbrev.
[m.S<] By 22 we can take S=[X::EQ(T1),T2] and S′=[X::EQ(T1′),T2′].
By ind E$\vdash'$ U.Type==T1::Type and E$\vdash'$ T1==T1′::Type, so by T=′.tran
E$\vdash'$ U.Type == T1′::Type.
[m.self−type] By ind.

Lemma 24 **($\vdash'$ implies $\vdash$ − type equality)** If E$\vdash'$ T==T′::Type then E$\vdash$ T==T′::Type.
Proof Induction, using 10.

Lemma 25 **(Cancellation − type equality)** If E$\vdash$ T == T′ :: Type, if both T and T′ are of the form [T .. T] | chan T | proc | int (not necessarily the same) then either
1) T = [T1..Tn] and T′=[T1′..Tn′] and for i=1..n E$\vdash$ Ti==Ti′::Type,
2) T = chan T1 and T′=chan T1′ and E$\vdash$ T1==T1′::Type,
3) T = proc = T′, or
4) T = int = T′.

Proof
We prove the result for $\vdash'$, by induction on the sequence of equations (ie bindings X::EQ(T) or U:[X::EQ(T1),..]) in E. Let W range over types of the forms X and U.Type, and use an abbreviated notation for type environments.

Base: E has no equations. The result is obvious, as provable equality coincides with syntactic identity.

Ind: E=E1,W=TW,E2 and E2 has no equations.
If T and T′ are of the specified form then {TW/W}T and {TW/W}T′ are.
We have E1,W,E2$\vdash'$ {TW/W}T=={TW/W}T′::Type.
Consider just one direction of the tuple case, ie T = [T1..Tn]
so {TW/W} T = [{TW/W}T1..{TW/W}Tn]
By the induction hypothesis {TW/W}T′ is a tuple of length n,
but T′ is not a variable, so for some T1′..Tn′ T′=[T1′..Tn′]
and {TW/W}T′=[{TW/W}T1′..{TW/W}Tn′].
By the rest of the induction hypothesis
E1,W,E2$\vdash'$ {TW/W}Ti == {TW/W}Ti′ :: Type (for i=1..n)
and by weakening E1,W=TW,E2$\vdash'$ {TW/W}Ti == {TW/W}Ti′ :: Type.
Now observe that E1,W=TW,E2$\vdash'$ Ti == {TW/W}Ti :: Type (and similarly for Ti′)
so E1,W=TW,E2$\vdash'$ Ti==Ti′.

B.5 Deconstruction lemmas

Lemma 26 (Deconstruction – sig equality) If E⊢ [X::K,T] == [X::K′,T′] sig and X not in E then E⊢ K == K′ and E,X::K⊢ T==T′::Type.
Proof Induction, using 20 in **[**S=.tran**]**.

Lemma 27 (Deconstruction – subsig) If E⊢ [X::K,T] < [X::K′,T′] sig and X not in E then E⊢ K <:: K′ and E,X::K⊢ T==T′::Type
Proof Induction, using 20 in **[**S<.tran**]** and 26 in **[**S<.S=**]**.

Lemma 28 (Deconstruction – module)
If E⊢ [T1,e] as [X::K,T2] : [X::K′,T2′] and X not in E then
E⊢ T1::K, E,X::K⊢ T2 :: Type, E,X::EQ(T1)⊢ e : T2, X not free in e,
E⊢ K <:: K′ and E,X::K⊢ T2 == T2′ :: Type

If E⊢ λU:SS1.m:S2 : ΠU:SS1′.S2′ and U not in dom E then
E,U:SS1⊢ m:S2, E⊢ SS1′ < SS1 sig and E,U:SS1′⊢ S2 < S2′ sig.

If E⊢ new x:T in m : SS and x not in dom E then
there exists T0 such that T=chan T0 and E,x:T⊢ m : SS.

If E⊢ m m′ : S then there exist U,SS1,S2 such that
E⊢ m:ΠU:SS1.S2, E⊢ m′:SS1, U not in S2 and E⊢ S2< S sig.

If E⊢ m with! m′ : S then there exist X,K,T2,Y,T′,T2′ such that
E⊢ m :[X::K,T2], E⊢ m′:[Y::EQ(T′),T2′] and E⊢ [X::EQ(T′),T2] < S sig.

If E⊢ (m:S) : S′ then E⊢ m:S and E⊢ S<S′ sig.

We note some easy consequences of the conclusion of 28:
By T.K< E⊢ T1 :: K′. By 1 E⊢ EQ(T1) <:: K. By 20
E,X::EQ(T1)⊢ T2==T2′ :: Type. By e.T= E,X::EQ(T1)⊢ e : T2′.
Hence by m.struct E⊢ [T1,e] as [X::K,T2] : [X::K,T2]
and by S<.Struct E⊢ [X::K,T2] < [X::K′,T2′] sig

Lemma 29 (Deconstruction – term)
If E⊢ e1|e2:T then E⊢ e1:proc and E⊢e2:proc and E⊢T==proc::Type.

If E⊢ new x:T in e : T2 and x not in dom E then
E,x:T⊢ e:T2 and there exists T0 such that T=chan T0.

If E atomic and has no module bindings and E⊢ e:T and E⊢T==[T1..Tn]::Type then
for some e1..en we have e=[e1 .. en] and for i=1..n we have E⊢ ei:Ti.
Proof We show just the last.
[e.var**]** Vacuous, as E atomic and by (25) not(E⊢chan T==[T1..Tn]::Type)
[e.T=**]** By ind hyp.
[e.rec**]** Have E⊢ [e1..em]:[T1′..Tm′] and E⊢ ok and for i=1..n E⊢ ei:Ti′ and
E⊢ [T1′..Tm′] == [T1..Tn]:: Type.
By (25) m=n and for i=1..n E⊢ Ti′ == Ti :: Type.
By e.T= for i=1..n E⊢ ei:Ti.

[e.proj] Vacuous, as E contains no module bindings.
[e.new][e.nil][e.par][e.out][e.in] Vacuous, as the conclusion type is proc in all cases, and by (25) not(E⊢proc==[T1..Tn]::Type)

B.6 Module substitution

Lemma 30 (Substitution – module)
If E,U:S,E′⊢ J and E⊢ mval:S then E,{mval/U}E′⊢ {mval/U}J.
Proof By tedious induction, using 28, 18, 5.

B.7 Unique Context/Redex Decomposition

This subsection gives the proof that module expressions have unique decompositions as an evaluation context and a redex. The complex structure of evaluation contexts for the different judgements makes it rather finickety, unfortunately.

Consider the judgements j

N⊢ m $\longrightarrow_i$ New N′ in m′ i=1,2	N⊢ m $\xrightarrow{\text{err}}$ s

and their axioms r

```
  m.red.beta, m.red.seal, m.red.new, m.red.with!
  m.red.abstype
  m.red.err.with!1, m.red.err.with!2, m.red.err.with!3
```

Let Cj range over the contexts associated with j (given by the C1 and C2 grammars).
For an axiom r, define

```
 MVAL    = {m | exists S. N⊢m:S and m is an mval}
 L(r)    = {m | exists S. N⊢m:S and m is an instance of the redex of r}
```

Lemma 31 (Unique decomposition) If N has no module bindings and N⊢ m:S then either m is an mval xor there is a unique j∈{1,2}, Cj, rule r∈m.red.* and instance lr of the redex of r such that m=Cj[lr].
Proof
We prove that if N has no module bindings then:
1) If Cj/=_ then Cj[m] not in MVAL
2) For any r, L(r) and MVAL are disjoint.
3) For r/=r′, L(r) and L(r′) are disjoint.
4) If exists S such that N⊢ m:S then exactly one of the following hold:
– m is in MVAL
– there is a unique (j,Cj,r,lr) such that m=Cj[lr], r is a rule of j,lr in L(r)

1,2,3 are by inspection. 4 is by induction on typing judgements.
[m.var] Vacuous as N has no module bindings
[m.S<] By induction hyp
[m.struct] Consider m=[T1,e] as [X::K,T2].
Case K=Type Take j=2, Cj=_, r = m.red.abstype, and lr=m.
By (2) m not in MVAL. By the form of rules m /= Cj′[lr′] for any other

```
    (j',Cj',r',lr'), as no other rule has a struct outermost.
Case T=EQ(T1')  m in MVAL.  m is not an instance of m.red.abstype.
    By the form of rules m /= Cj'[lr'] for any other
    (j',Cj',r',lr'), as no other rule has a struct outermost.
[m.self-type] By induction hyp.
[m.fun]  m in MVAL.   By the form of rules m /= Cj[lr] for any
    (j,Cj,r,lr), as no rule or context clause has a functor outermost.
[m.app] Consider m m'.  By the form of m.app E⊢m:S and E⊢m':SS1 for
some functor and structure signatures S, SS1. In all cases (m m') not in MVAL.  (a)

Case m in MVAL.  Here m must be a functor expression
  Case m' in MVAL.  Here m' must be a struct expression, so take
     j=1, Cj=_, r= m.red.beta, lr =m m'.
     By the form of rules (m m')/= Cj'[lr'] for any
     (j',Cj',r',lr'), as no other rule has an app outermost and
     neither m nor m' can be Cj''[lr''] for a redex instance lr''.
  Case there is a unique (j,Cj',r,lr) such that m'=Cj'[lr], r is a rule of j,lr in L(r)
     Take Cj= (m Cj').
     Consider (j',Cj'',r',lr') such that (m m')=Cj''[lr'], r' is a rule of j',lr' in L(r')
        Case Cj''=_, then (m m') in L(r'), but it cannot be as m' not in MVAL. (b)
        Case Cj'' = (Cj''' m'), then m= Cj'''[lr']. This contradicts
           uniqueness for m as m in MVAL. (c)
        Case Cj'' = (mval Cj'''), then m= mval and m'=Cj'''[lr'].
           This contradicts uniqueness for m' unless j=j', Cj'=Cj''',
           r=r', lr=lr', in which case Cj = (m Cj') = Cj''. (d)
     By (a) (m m') not in MVAL, and by (b,c,d) there is no competing 4-tuple.
Case there is a unique (j,Cj',r,lr) such that m=Cj'[lr], r is a rule of j,lr in L(r)
     Take Cj= (Cj' m').
     Consider (j',Cj'',r',lr') with r' a rule of j', lr' in L(r') and (m m')=Cj''[lr']
        Case Cj''=_, then (m m') in L(r'), but it cannot be as m not in MVAL. (b)
        Case Cj'' = (Cj''' m'), then m= Cj'''[lr'].
           This contradicts uniqueness for m unless j=j', Cj'=Cj''',
           r=r', lr=lr', in which case Cj = (Cj' m') = Cj''. (c)
        Case Cj'' = (mval Cj''').  Cannot be, as m not in MVAL (d)
     By (a) (m m') not in MVAL, and by (b,c,d) there is no competing 4-tuple.
[m.seal] Consider (m:SS).  By the rule N⊢m:SS. Applying the induction hypothesis to m,
Case m in MVAL.  As N⊢m:SS, m must be a struct (and as an MVAL, with an EQ kind).
     Take j=1, Cj=_, r= m.red.seal, and lr=(m::SS).
     Consider (j',Cj',r',lr') with r' a rule of j', lr' in L(r') and (m:SS)=Cj'[lr']
        Case Cj'=_, then lr'=(m:SS)=lr. By (3) j'=j,r'=r.
        Case Cj' = (Cj'':SS), then m= Cj''[lr']. But m in MVAL.
     Clearly (m:SS) is not in MVAL, and by the above there is no competing 4-tuple.
Case there is a unique (j,Cj',r,lr) such that m=Cj'[lr], r is a rule of j,lr in L(r)
     Take Cj= (Cj':SS)
[m.with] Consider m with! m'.  By the m.with rule we have N⊢ m : SS
and  N⊢ m': SS' for two struct sigs.  Clearly m with! m' is not in MVAL.
Applying the induction hypothesis to m':
Case m' in MVAL
  Applying the induction hypothesis to m:
  Case m in MVAL. (m with! m') is an instance of r = m.red.err.with!3
     Take j=1, Cj=_, r = m.red.err.with!3,  lr = m with! m'.
```

```
  Case there is a unique (j,Cj,r,lr) such that m=Cj[lr], r is a rule of j,lr in L(r)
      Case j=2 (so r= m.red.abstype)  and Cj=_.    Take j′=1, Cj′=_, lr′= m with! m′,
        and r the unique applicable rule from m.red.with!,  m.red.err.with!1,
        m.red.err.with!2.
      Case j=2 (so r= m.red.abstype)  and Cj=C2h.  Take Cj′ = Cj with! m′.
      Case j=1.   Take Cj′ = Cj with! m′.
Case there is a unique (j,Cj,r,lr) such that m′=Cj[lr], r is a rule of j,lr in L(r)
  Take Cj′= (m with! Cj).
[m.new] Consider m=new x:chan T in m′.
     Take r= m.red.new, Cj=_ and lr=m.
     By (2,3) m not in L(r′) for any r′/=r, and m not in MVAL.
     Consider (j′,Cj′,r′,lr′) with Cj′/=_, r′ a rule of j′, lr′ in
     L(r′) and new x:chan T in m′ = Cj′[lr′].
     Must have Cj′=new x:chan T in Cj′′, but it cannot be.
```

B.8 Subject Reduction - build time

Lemma 32 If

```
   N⊢ ok,
   N has no module bindings,
   forall U in dom(F). N⊢ F(U) : sig(F(U)), and
   typify(N),env(F) ⊢ m:S
then N⊢ F(m) : F(S).
```

Proof

```
Call the premises a,b,c,d. By d, 30 (for the x:T parts), narrowing
(for the X::K parts), and N has no module bindings (for the U:S parts),
we have N,env(F)⊢ m:S  (e).
By b,c each F(U) contains no free module identifiers.
Proceed by induction on the length of F.
For F empty the result is immediate.
Consider             N,env(F,(U,mval))    ⊢ m:S
By defn env          N,env(F),U:sig(mval) ⊢ m:S
By c and 16          N,env(F)⊢ mval : sig(mval)
By 30                N,env(F)⊢  {mval/U}m : {mval/U}S
By the ind hyp       N⊢ F(m):F(S).
```

Lemma 33 For i=1,2, if N⊢ m : S and N⊢ m $\longrightarrow_i$ New N′ in m′ and N,N′ disjoint then N,N′⊢ m′:S. Moreover if N atomic and has no module bindings then N,N′ is likewise.

Proof

```
[m.red.beta]  By 28 have:
N⊢ λU:SS1.m:S2:ΠU:SS1′.S2′                  a
N⊢ mval:SS1′                                b
U not in S2′                                c
─────────────────────────────────m.app
N⊢ (λU:SS1.m:S2)  mval : S2′

and N⊢ S2′ < S.  By Lemma 28  N,U:SS1⊢ m:S2  (d), N⊢ SS1′ < SS1 sig  (e)
and N,U:SS1′⊢ S2 < S2′ sig  (f)
```

```
By (b,e),m.S<        N⊢ mval:SS1                       g
By (d,g,30)          N⊢ {mval/U}m : {mval/U}S2         h
By (f,b,30)          N⊢ {mval/U}S2 < {mval/U}S2′ sig
By m.S<              N⊢ {mval/U}m : {mval/U}S2′
By U not in S2′      N⊢ {mval/U}m : S2′
By m.S<              N⊢ {mval/U}m : S
```

[m.red.seal]

```
Consider N⊢ ([T1,e] as SS : SS′) : S   where SS=[X::EQ(T1′),T2],  SS′=[X::K′,T2′]
and      N⊢ ([T1,e] as SS : SS′) ⟶₁ [T1,e] as SS′  m.red.seal
By 28 N⊢ [T1,e] as SS : SS′ and N⊢ SS′ < S sig
By 28  N⊢ T1::EQ(T1′)                   a
       N,X::EQ(T1′)⊢ T2 :: Type         b
       N,X::EQ(T1)⊢ e : T2              c
       X not free in e                  d
       N⊢ EQ(T1′) <:: K′                e
       N,X::EQ(T1′)⊢ T2 == T2′ :: Type f
By T.K<                 N⊢ T1::K′
By 15                   N⊢ [X::K′,T2′] sig so (cf S.Struct) N,X::K′⊢ T2′ :: Type
By (a) N⊢ EQ(T1)<::EQ(T1′).  By (c) and narrowing and e.T=   N,X::EQ(T1)⊢ e: T2′
By m.struct             N⊢ [T1,e] as [X::K′,T2′] : [X::K′,T2′]
By m.S<                 N⊢ [T1,e] as [X::K′,T2′] : S
```

[m.red.new] Immediate from (28).

[m.red.with!] Consider a redex (m with! m′) and N⊢ m with! m′ : S. Have

```
m  =[T1,e ] as [X::Type,T2  ]                a
m′ =[Z ,e′] as [Y::EQ(Z),T2′]                c
N(Z)=EQ(TZ)                                  f
typify(N)⊢ T1==TZ::Type                      g
------------------------------------------------------------ m.red.with!
N⊢ m with! m′ ⟶₁  [Z,e] as [X::EQ(Z),T2]
ie
N⊢ [T1,e] as [X::Type,T2] with! [Z,e′] as [Y::EQ(Z),T2′]  ⟶₁  [Z,e] as [X::EQ(Z),T2]

and by (28)
  N⊢ [T1,e] as [X::Type,T2]   : [X::K′′,T2′′]              h
  N⊢ [Z,e′] as [Y::EQ(Z),T2′] : [Y::EQ(T1′′′),T2′′′]        i
  N⊢ [X::EQ(T1′′′),T2′′] < S sig                           m
(wlg choose X,Y distinct from dom N and fv e). By well—formedness N⊢ ok.   j
By (j),(f) and sundry rules  N⊢ Z :: EQ(Z)                                 l

By (h),(28), (l) and narrowing    N,X::EQ(Z)⊢ T2 :: Type

By (h) and 28                     N,X::EQ(T1)⊢ e : T2       k
By (f),(j) N⊢Z::EQ(TZ).  By  1    N⊢ EQ(Z)<::EQ(TZ)
By (g) and N—narrowing 21         N⊢ T1==TZ::Type
By sundry rules                   N⊢ EQ(TZ)<::EQ(T1)
By K<.tran                        N⊢ EQ(Z)<::EQ(T1)
By this, (k) and narrowing        N,X::EQ(Z)⊢ e:T2

By assumption X not in e
```

```
By these four and m.struct  N⊢  [Z,e] as [X::EQ(Z),T2] :  [X::EQ(Z),T2]

By (i) and 28               N⊢ EQ(Z) <:: EQ(T1′′′)
By (h), 28 and narrowing    N,X::EQ(Z)⊢ T2==T2′′::Type
By these two and S<.Struct  N⊢ [X::EQ(Z),T2] < [X::EQ(T1′′′),T2′′] sig

then by m.S<        N⊢  [Z,e] as [X::EQ(Z),T2] :  [X::EQ(T1′′′),T2′′]
 and by (m),m.S<    N⊢  [Z,e] as [X::EQ(Z),T2] :  S
```

[m.red.abstype**]**

```
Have  N⊢ [T1,e] as [X::Type,T2] : [X::K′,T2′]
      with no top—level news or module ids free in that struct and Y not in dom N
      wlg take X not in dom N and X/=Y.
By 28
      N⊢ T1::Type                  c
      N,X::Type⊢ T2 :: Type        j
      N,X::EQ(T1)⊢ e : T2          k
      X not free in e              l

      N⊢ Type <:: K′               m
      N,X::Type⊢ T2 == T2′ :: Type  n

By (c) and K.EQ,E.type            N,Y::EQ(T1)⊢ ok.      f
By T.var,T.K<,K<.EQ,T=.refl,T.T=  N,Y::EQ(T1)⊢ Y::EQ(Y)

By (j),weakening and permutation 19 N,Y::EQ(T1),X::Type⊢ T2  :: Type
By f,T.var and 1                    N,Y::EQ(T1)⊢ EQ(Y) <:: EQ(T1)        g
By narrowing 20                     N,Y::EQ(T1),X::EQ(Y)⊢ T2  :: Type

By (k),weakening,permutation        N,Y::EQ(T1),X::EQ(T1)⊢  e : T2
By narrowing 20                     N,Y::EQ(T1),X::EQ(Y)⊢   e : T2

From these three, (l) and m.struct N,Y::EQ(T1)⊢ [Y,e] as [X::EQ(Y),T2 ] : [X::EQ(Y),T2 ]

By (c),1 N⊢EQ(T1)<:: Type. By (m) and K<.tran N⊢EQ(T1)<::K′.
By weakening                     N,Y::EQ(T1)⊢ EQ(T1)<:: K′
By (f),T.var,T=.EQ, K=.T=,K<.K=  N,Y::EQ(T1)⊢ EQ(Y) <:: EQ(T1)
By K<.tran                       N,Y::EQ(T1)⊢ EQ(Y) <:: K′
(could be more direct, as by (d) we really know K′ =Type)

By (n),weakening, permutation and narrowing  N,Y::EQ(T1),X::EQ(Y)⊢ T2 ==T2′  :: Type

By these two and S<.Struct  N,Y::EQ(T1)⊢ [X::EQ(Y),T2] < [X::K′,T2′] sig
By m.S<                     N,Y::EQ(T1)⊢ [Y,e] as [X::EQ(Y),T2] : [X::K′,T2′]
```

Lemma 34 If E⊢ C1[m] : S then there exists S′ such that E⊢ m: S′ and for all E′,m′, if E,E′⊢ m′:S′ then E,E′⊢ C1[m′] : S.
Proof Induction on the pair of C1 and the type derivation.
If C1 = _ the result is immediate, so consider otherwise and take

cases of the last type rule in the derivation.

[m.S<**]**

```
E⊢ C1[m]:S1
E⊢ S1 < S2 sig
────────────────── m.S<
E⊢ C1[m]:S2
```

By the ind hyp there exists S1′ such that E⊢ m: S1′ and for all
E′,m′, if E,E′⊢ m′:S1′ then E,E′⊢ C1[m′] : S1
By weak E,E′⊢ S1 < S2 sig. By m.S< E,E′⊢ C1[m′] : S2.

[C1 m2   /m.app**]**

```
E⊢ C1[m]:ΠU:SS1.S
E⊢ m2:SS1
U not free in S2
────────────────── m.app
E⊢ C1[m] m2 : S
```

By the ind hyp there exists S′ such that E⊢ m: S′ and for all
E′,m′, if E,E′⊢ m′:S′ then E,E′⊢ C1[m′] : ΠU:SS1.S
By weak E,E′⊢ m2:SS1. By m.app E,E′⊢ C1[m′] m2 : S.
[mval C1 /m.app**]** **[**C1 with! mval /m.with**]** **[**m1 with! C1 /m.with**]** **[**C1:SS /m.seal**]**
All similar to the previous case.
[m.var**]** **[**m.struct**]** **[**m.self–type**]** **[**m.fun**]** **[**m.new**]** All
vacuous, as there is no A1 that can match them.

Lemma 35 (Module subject reduction) If N⊢ m:S and
N⊢ m ⟹ New N′ in m′ and N,N′ disjoint then N,N′⊢ m′:S.
Moreover if N atomic and N has no module bindings then so does N,N′.
Proof
[m.wred.refl**]** Immediate
[m.wred.tran**]** By the induction hypothesis twice.
[m.wred.red**]**
Suppose N⊢ Ci[m]:S. By (34) (using the fact that any C2 is also a C1)
there exists S′ such that N⊢ m:S′ and for all N′,m′, if N,N′⊢ m′:S′
then N,N′⊢ Ci[m′] : S. By Lemma 33 N,N′⊢ m′:S, so using (a) N,N′⊢ Ci[m′] : S.

B.9 Subject Reduction - run time

If s is a finite partial function from term variables to expressions and
E′ is a type environment with term variable bindings only
then say E⊢ s : E′ iff dom(s)=dom(E′) and forall x in dom(s). E⊢ s(x) : E′(x)

Lemma 36 (Well–formed term substitutions) If E atomic and has no module bindings,
E⊢ e:T and E⊢ pat:T ▷ E′ then {e/pat}′ is defined and E⊢ {e/pat}′ : E′.
Proof Induction on E⊢ pat:T ▷ E′, using 29.

Lemma 37 (Substitution – term)
If E,E′,E′′⊢ e:T and E⊢ s:E′ then E,E′′⊢ se : T.

Proof Induction on E,E′,E′′⊢ e:T, using 16 and 5.

Lemma 38 (Type soundness of structural congruence)
If E⊢e:proc and e == e′ then E⊢e′:proc.
Proof
We show that if E⊢e:proc and (e == e′ or e′==e) then E⊢e′:proc, by induction on the derivation of structural congruence.
[sc.id] (a) If E⊢ 0|e1 : proc. By (29) E⊢ e1:proc.
(b) If E⊢ e1:proc then by well–formedness E⊢ok, so by e.nil E⊢0:proc, and by e.par E⊢0|e1:proc.
[sc.com] [sc.ass] Similar.
[sc.ext] WLG take x not in dom E. (a) If E⊢ e1| new x:T in e2 : proc then by 29 E⊢ e1:proc and E⊢ new x:T in e2:proc.
By Lemma 29 E,x:T⊢ e2:proc and there exists T0 such that T=chan T0.
By weakening E,x:T⊢ e1|e2:proc. By e.new E⊢ new x:T in e1|e2 : proc.
(b) If E⊢ new x:T in e1|e2 : proc by 29 E,x:T⊢e1|e2:proc
and T=chan T0 for some T0. By 29 E,x:T⊢e1:proc
and E,x:T⊢e2:proc. By strengthening 16 E⊢e1:proc.
By e.new and e.par E⊢e1|new x:T in e2 : proc.
[sc.cong.par] By 29 and the ind hyp.
[sc.cong.new] By 29 and the ind hyp.
[sc.refl] Immediate. [sc.sym] by the ind hyp. [sc.tran] by the ind hyp twice.

Lemma 39 (Process subject reduction) If E atomic and has no module bindings, E⊢e:proc and e⟶ e′ then E⊢ e′:proc.
Proof
By induction on derivation of e⟶ e′.
[e.red.comm] Wlg take vars(pat) and dom E disjoint.
By 29 E⊢ x!e1 : proc and E⊢ x?pat.e2 :proc.
Clearly for some T,T′,E′ we have E⊢ x: chan T′, E⊢ x:chan T, E⊢ e1:T′, E⊢pat:T▷ E′, and E,E′⊢e2:proc. Clearly also E⊢ chan T == chan T′ :: Type.
By 25 E⊢ T==T′::Type so E⊢e1:T. By 36 {e1/pat}′ is defined and E⊢ {e1/pat}′ : E′.
By 37 E⊢ {e1/pat}′e2:proc.
[e.red.par] By 29, the ind hyp and e.par.
[e.red.res] Wlg take x not in dom E.
By 29, the ind typ (noting that E,x:chan T0 is atomic) and e.new
[e.red.sc] By 38 twice and the ind hyp.

Lemma 40 (Process soundness) If E atomic and has no module bindings and E⊢e:proc then not (e $\xrightarrow{\text{err}}$).
Proof We show (e $\xrightarrow{\text{err}}$ and E atomic and has no module bindings and E⊢e:proc) implies false, by induction on e $\xrightarrow{\text{err}}$.
[e.err.comm] Wlg take vars(pat) and dom E disjoint.
By 29 E⊢ x!e1 : proc and E⊢ x?pat.e2 :proc.
Clearly for some T,T′,E′ we have E⊢ x: chan T′, E⊢ x:chan T, E⊢ e1:T′, E⊢pat:T▷ E′, and E,E′⊢e2:proc. Clearly also E⊢ chan T == chan T′ :: Type
By 25 E⊢ T==T′::Type so E⊢e1:T
By 36 {e1/pat}′ is defined, so not (e $\xrightarrow{\text{err}}$).
[e.err.sc] By 38 and the ind hyp.
[e.err.par] By 29 twice and ind.
[e.err.res] Wlg take x not in dom E.

By 29, the ind typ (noting that E,x:chan T0 is atomic) and e.new

B.10 Subject Reduction and Soundess - whole systems

Theorem 1 If ⊢ N,F,e ok and N,F,e $\xrightarrow{\texttt{Com}}$ N′,F′,e′ then ⊢ N′,F′,e′ ok.
Proof
[build**]** By 32 N⊢ F(m) : F(S). By this and 35 N,N′⊢ mval : F(S)
and moreover N,N′ atomic and contains no module bindings.
By well—formedness N,N′⊢ ok. Now consider typing clause of ⊢ (N,N′),F+(U,mval),e ok
— For U consider cases of mval a struct or a functor, then apply
28 and m.struct or 28 and m.fun respectively.
— For U′ in dom(F+(U,mval))—{U}, the clause follows by weakening.
The proc clause of ⊢ (N,N′),F+(U,mval),e ok follows by weakening.

[load**]** wlg suppose X not in dom N. The typing and N clauses of N,F,e|e′ are immediate.
For the proc clause, by hypothesis N⊢ e:proc.

By hypothesis	N⊢ [T,e′] as [X::K,proc] : [X::K,proc]
By 28	N,X::EQ(T)⊢ e′ : proc and X not free in e′
By strengthening 18	N⊢ e′:proc
By e.par	N⊢ e|e′:proc

[compute**]** The typing and N clauses of N,F,e′ are immediate.
The proc clause follows from 39.

Theorem 2 If ⊢ N,F,e ok then there is no transition N,F,e $\xrightarrow{\texttt{tau}}$ err(runtime error).
Proof Immediate from the system rule and 40.

B.11 Proof of Theorem 3

Lemma 41 If E,Y::Type,U:[X::EQ(Y),T],E′ ⊢ J and Y not in T
then E,U:[X::Type,T],{U.Type/Y}E′ ⊢ {U.Type/Y}J.
Proof
[T.var**][**E.mod**]** Straightforward.
[m.var**]** Case before, after: routine.
Case same: by ind E,U:[X::Type,T],{U.Type/Y}E′ ⊢ ok.
By m.var E,U:[X::Type,T],{U.Type/Y}E′ ⊢ U:[X::Type,T]
By T.Proj,T=.refl,T.T= E,U:[X::Type,T],{U.Type/Y}E′ ⊢ U.Type :: EQ(U.Type)
By m.self—type E,U:[X::Type,T],{U.Type/Y}E′ ⊢ U:[X::EQ(U.Type),T]

Theorem 3 If m1 and m2 are structure expressions

m1 = [T1,e1] as [X::K1,T1′] with SS1 = [X::K1,T1′]

m2 = [T2,e2] as [X::K2,T2′] SS2 = [X::K2,T2′]

and

m2′ = (λU1:SS1.m2:SS2) [T1,e1] as [X::EQ(T1),T1′]

then

($\exists$N,F,e. empty,$\emptyset$,0 $\xrightarrow{\texttt{U1:=m1}}$ $\xrightarrow{\texttt{U2:=m2}}$ N,F,e) iff ($\exists$N,F,e. empty,$\emptyset$,0 $\xrightarrow{\texttt{U2:=m2}'}$ N,F,e).

Proof For the left–to–right direction, first consider K1=Type.
By the build rule there exist SS1′,Y,mval1,SS2′,N2,mval2 such that

a empty $\vdash$ m1 : SS1′
b mval1 = [Y,e1] as [X::EQ(Y),T1′]
c empty $\vdash$ m1 $\Longrightarrow$ New Y::EQ(T1) in mval1
d Y::Type,U1:[X::EQ(Y),T1′] $\vdash$ m2 : SS2′
e Y::EQ(T1) $\vdash$ {mval1/U1}m2 $\Longrightarrow$ New N2 in mval2
f {Y} disjoint from fv(m2) and dom(N2)
g N= Y::EQ(T1),N2

By (a,28) empty $\vdash$ m1 : SS1
By (d,28) Y::Type,U1:[X::EQ(Y),T1′] $\vdash$ m2 : SS2
By (f,41) U1:[X::Type, T1′] $\vdash$ m2 : SS2
By m.fun empty $\vdash$ λU1:[X::Type,T1′].m2:SS2 : ΠU1:[X::Type,T1′].SS2
By sundry rules empty $\vdash$ λU1:[X::Type,T1′].m2:SS2 : ΠU1:[X::EQ(T1),T1′].{T1/U1.Type}SS2
By m.app empty $\vdash$ (λU1:[X::Type,T1′].m2:SS2) [T1,e1] as [X::EQ(T1),T1′]
: {T1/U1.Type}SS2
ie empty $\vdash$ m2′ : {T1/U1.Type}SS2

There is then some N′ and mval2′ such that empty$\vdash$ m2′ $\Longrightarrow$ New N′ in mval2′
so empty,$\emptyset$,0 $\xrightarrow{\texttt{U2 := m2}'}$ N′,{U2$\mapsto$mval2′},0

Now consider K1=EQ(T). m1 is a value, so by the build rule there exist SS1′,SS2′,mval2 such that

a empty $\vdash$ m1 : SS1′
b U1:SS1$\vdash$ m2 : SS2′
c empty$\vdash$ {m1/U1}m2 $\Longrightarrow$ New N in mval2
d F={U1$\mapsto$m1,U2$\mapsto$mval2} and e=0

By (a,28) empty $\vdash$ m1 : SS1
By (b,28) U1:SS1$\vdash$ m2 : SS2
By m.fun empty $\vdash$ λU1:SS1.m2:SS2 : ΠU1:SS1.SS2
By sundry rules empty $\vdash$ λU1:SS1.m2:SS2 : ΠU1:SS1.{T/U1.Type}SS2
By m.app empty $\vdash$ (λSS1.m2:SS2) m1 : {T/U1.Type}SS2
ie empty $\vdash$ m2′ : {T/U1.Type}SS2

There is then some N′ and mval2′ such that empty$\vdash$ m2′ $\Longrightarrow$ New N′ in mval2′
so empty,$\emptyset$,0 $\xrightarrow{\texttt{U2 := m2}'}$ N′,{U2$\mapsto$mval2′},0

For the right–to–left direction, by the build rule there exists SS2′,mval2′ such that

empty $\vdash$ m2′ : SS2′
empty $\vdash$ m2′ $\Longrightarrow$ New N in mval2′

By 28 there exist SS1′′ and SS2′′ with U1 not in SS2′′ such that
empty $\vdash$ [T1,e1] as [X::EQ(T1),T1′] : SS1′′
empty $\vdash$ (λU1:SS1.m2) : ΠU1:SS1′′.SS2′′
By 28 U1:SS1$\vdash$ m2 : SS2,   empty$\vdash$ SS1′′ < SS1 sig and U1:SS1′′$\vdash$SS2 < SS2′′ sig
By m.S< empty$\vdash$ [T1,e1] as [X::EQ(T1),T1′] : SS1   (recall SS1=[X::K1,T1′])
so   empty$\vdash$ [T1,e1] as SS1 : SS1

There is then some N1 and mval1 such that empty$\vdash$ m1 $\Longrightarrow$ New N1 in mval1
so empty,$\emptyset$,0 $\stackrel{\text{U1:=m1}}{\longrightarrow}$N1,{U1$\mapsto$mval1},0.

By weakening   typify(N1),U1:SS1$\vdash$ m2 : SS2
By refl or m.red.abstype   typify(N1)$\vdash$ sig(mval1) < SS1 sig
By narrowing for modules   typify(N1),U1:sig(mval1)$\vdash$ m2 : SS2

There is then some N2 and mval2 such that N1$\vdash$ {mval1/U1}m2 $\Longrightarrow$ New N2 in mval2
and empty,$\emptyset$,0 $\stackrel{\text{U1:=m1}}{\longrightarrow}$ $\stackrel{\text{U2:=m2}}{\longrightarrow}$(N1,N2),{U1$\mapsto$mval1,U2$\mapsto$mval2},0.

References

[AWWV95] J. L. Armstrong, M. C. Williams, C. Wikström, and S. R. Virding. *Concurrent Programming in Erlang.* Prentice Hall, 2nd edition, 1995.

[BA99] Matthias Blume and Andrew W. Appel. Hierarchical modularity. *ACM Transactions on Programming Languages and Systems*, 21(4):813–847, July 1999.

[Car97] L. Cardelli. Program fragments, linking, and modularization. In *POPL'97*, pages 266–277, January 1997.

[CDG+89] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 report (revised). Technical report, DEC SRC, September 1989. SRC-052.

[CL90] L. Cardelli and X. Leroy. Abstract types and the dot notation. In *Programming Concepts and Methods, IFIP State of the Art Reports*, pages 479–504. North Holland, March 1990. Also appeared as SRC Research Report 56.

[Dro00] Sophia Drossopoulou. Towards an abstract model of Java dynamic linking and verification. In *Preliminary Proceedings of the Third Workshop on Types in Compilation (TIC 2000). CMU Technical Report CMU-CS-00-161*, September 2000.

[Dug00] Dominic Duggan. Sharing in typed module assembly language. In *Preliminary Proceedings of the Third Workshop on Types in Compilation (TIC 2000). CMU Technical Report CMU-CS-00-161*, September 2000.

[FF98] M. Flatt and M. Felleisen. Units: Cool modules for hot languages. In *PLDI 98*, pages 236–248, 1998.

[FGL+96] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *Proceedings of CONCUR '96. LNCS 1119*, pages 406–421. Springer-Verlag, August 1996.

[GM99] Neal Glew and Greg Morrisett. Type-safe linking and modular assembly language. In *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 250–261, New York, NY, January 1999. ACM.

[HL94] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Conference record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 123–137, January 1994.

[HN00] Michael Hicks and Scott Nettles. Active networking means evolution (or enhanced extensibility required). In *Proceedings of the Second International Working Conference on Active Networks*, October 2000. To appear.

[HWC00] Michael Hicks, Stephanie Weirich, and Karl Crary. Safe and flexible dynamic linking of native code. In *Preliminary Proceedings of the Third Workshop on Types in Compilation (TIC 2000). CMU Technical Report CMU-CS-00-161*, September 2000.

[Ler94] Xavier Leroy. Manifest types, modules, and separate compilation. In *Conference Record of POPL '94: 21ST ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon*, pages 109–122, New York, NY, January 1994. ACM.

[Ler96] Xavier Leroy. A syntactic theory of type generativity and sharing. *Journal of Functional Programming*, 6(5):667–698, 1996.

[Ler00] Xavier Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, May 2000.

[Lil97] Mark Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems.* PhD thesis, CMU, 1997. CMU-CS-97-122.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Parts I + II. *Information and Computation*, 100(1):1–77, 1992.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* The MIT Press, Cambridge, Mass., 1990.

[Ode95] Martin Odersky. Polarized name passing. In *Proceedings of FSTTCS '95. LNCS 1026*, December 1995.

[PS96] Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–454, 1996.

[PT00] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. In *Proof, Language and Interaction: Essays in Honour of Robin Milner.* MIT Press, 2000.

[Rus98] Claudio V. Russo. *Types For Modules.* PhD thesis, University of Edinburgh, LFCS, 1998. ECS-LFCS-98-389.

[Sew01] Peter Sewell. Modules, abstract types, and distributed versioning. In *Proceedings of POPL 2001*, January 2001. To appear.

[SWP98] Peter Sewell, Paweł T. Wojciechowski, and Benjamin C. Pierce. Location independence for mobile agents. In *Proceedings of the Workshop on Internet Programming Languages (Chicago)*, May 1998. Full version appeared in LNCS 1686.

[US01] Asis Unyapoth and Peter Sewell. Nomadic Pict: Correct communication infrastructure for mobile computation. In *Proceedings of POPL 2001*, January 2001. To appear.

[Woj00] Paweł T. Wojciechowski. *Nomadic Pict: Language and Infrastructure Design for Mobile Computation.* PhD thesis, Computer Laboratory, University of Cambridge, June 2000. Available as Technical Report 492.

[WS00] Paweł T. Wojciechowski and Peter Sewell. Nomadic Pict: Language and infrastructure design for mobile agents. *IEEE Concurrency*, 8(2):42–52, April–June 2000. Invited submission for ASA/MA 99.

[ZGM99] Steve Zdancewic, Dan Grossman, and Greg Morrisett. Principals in programming languages: A syntactic proof technique. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, volume 34.9 of *ACM Sigplan Notices*, pages 197–207, N.Y., September 27–29 1999. ACM Press.