Modules, Abstract Types, and Distributed Versioning

Peter Sewell*

Computer Laboratory, University of Cambridge

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.

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].

*Supported by a Royal Society University Research Fellowship.

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.

$C \stackrel{def}{=}$
```
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`

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 omitted. They can be found, together with proofs, in the technical report [Sew00].

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)))
```

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

```
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`

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 ≝
  module
    import type t
    export val c : t chan
  begin
    newval c : t chan
  end
```

to give

```
D′ ≝
  module
    import type t
    export val c : t chan
  begin
    val c : t chan = z2
  end
```

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 ≝                          T1 ≝
  module                        module
    export type t                 export type t
  begin                         begin
    type t=int*int*int            type t = int
  end                           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 ≝  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
```

[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  ≝ sig type t  val x:t  val i:t→int end      CSIG(X) ≝ sig val c : X chan end
```

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

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

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

The new value of C in 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
```

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

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

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
```

is a dependent function taking structures of signature `SS` and returning modules of signature `S`, which may mention `U`.

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 `mval`s 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.

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 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′`

```
⊢ 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*

```
⊢ 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 =def λU1:[X::Type,X].λU2:[Y::EQ(U1.Type),Y].
             (U1.Type,U2.term) as [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
```

More useful module examples quickly become rather verbose, especially with our cut-down term language; we do not include any here.

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

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′
```

```
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

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` $\longrightarrow$`e′`. 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]`$\twoheadrightarrow$`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` $\in$ `dom(F)` to `sig(F(U))`, and `typify(N)` for the type environment mapping each `X` $\in$ `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)` $\Longrightarrow$ `New N′ in mval` are treated as binders, ensure that all previously-generated names are alpha-converted away from the free names of `m`.

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 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]
```

Applying *Fopaque* to *C*:

```
⊢ Fopaque C

  ⟶ {C/U}[U.Type*U.Type,[U.term,U.term]]
                    as [Y::Type,Y]

  =    [int*int,(6,6)] as [Y::Type,Y]

  ⟶ New Z::EQ(int*int) in
       [Z,(6,6)] as [Y::EQ(Z),Y]
```

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
       s([U.Type*U.Type,(U.term,U.term)]
         as [Y::Type,Y])

       where s = {[Z,6] as [X::EQ(Z),X]/U}

  =   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

We state only the key lemmas and main soundness properties of the semantics.

Lemma 1 (module substitution) *If* $\mathtt{E,U:S,E'} \vdash \mathtt{J}$ *for some judgement* $\mathtt{J}$*, and* $\mathtt{E} \vdash \mathtt{mval:S}$*, then* $\mathtt{E,\{mval/U\}E'} \vdash \mathtt{\{mval/U\}J}$.

Lemma 2 (term substitution) *If* $\mathtt{E,E',E''} \vdash \mathtt{e:T}$ *and* $\mathtt{E} \vdash \sigma\mathtt{:E'}$*, where* σ *is a term substitution, then* $\mathtt{E,E''} \vdash \sigma\mathtt{e} : \mathtt{T}$.

Lemma 3 (module subject reduction) *If we have* $\mathtt{N} \vdash \mathtt{m} : \mathtt{S}$ *and* $\mathtt{N} \vdash \mathtt{m} \Longrightarrow \mathtt{New\ N'\ in\ m'}$ *and* $\mathtt{N,N'}$ *disjoint then* $\mathtt{N,N'} \vdash \mathtt{m':S}$*. Moreover if* $\mathtt{N}$ *atomic and has no module bindings then* $\mathtt{N,N'}$ *is likewise.*

Lemma 4 (process subject reduction) *If* $\mathtt{E}$ *is atomic and has no module bindings,* $\mathtt{E} \vdash \mathtt{e:proc}$ *and* $\mathtt{e} \longrightarrow \mathtt{e'}$ *then* $\mathtt{E} \vdash \mathtt{e':proc}$.

Theorem 1 *If* $\vdash \mathtt{N,F,e\ ok}$ *and* $\mathtt{N,F,e} \xrightarrow{\mathtt{Com}} \mathtt{N',F',e'}$ *then* $\vdash \mathtt{N',F',e'\ ok}$.

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

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*

$$\texttt{m1 = [T1,e1] as [X::K1,T1}'\texttt{]}$$
$$\texttt{m2 = [T2,e2] as [X::K2,T2}'\texttt{]}$$

and

$$\texttt{m2}' \texttt{= (}\lambda\texttt{U1:SS1.m2:SS2) [T1,e1] as [X::EQ(T1),T1}'\texttt{]}$$

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

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

iff

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

Note that one cannot take $\texttt{m2}'$ to be simply $(\lambda\texttt{U1:SS1.m2:SS2) m1}$, as if `K1=Type` and `U1` is used in $\texttt{T2}'$ this cannot generally be given a useful signature. Instead, the abstraction is enforced solely by the argument signature of the functor. 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 $\texttt{U2:=m2}'$ 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 $\texttt{m2}'$.

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 typechecking (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 Wans-

brough, Lucian Wischik, and Paweł Wojciechowski, for discussions and comments.

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.

[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.

[Sew00] Peter Sewell. Modules, abstract types, and distributed versioning. Technical Report 506, University of Cambridge, 2000. Available from `http://www.cl.cam.ac.uk/users/pes20/`.

[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. Extended version of a paper in 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.