A Mechanism for Extensible Mutual Recursion

Peter Vanderbilt
pvanderbilt@acm.org

Abstract
This paper summarizes an approach to extensible mutual recursion using what the author calls, “open modules.” An open module effectively parameterizes its content so that it can be used both for the current module and future, extending modules. This paper discusses some of the issues with extensible mutual recursion, describes an open module construct and gives encodings and typing rules for open modules.¹

1. Problem with extensible mutual recursion
An issue arises when extending mutually recursive classes: the mutual recursion stays with the base classes. The Subject/Observer example is a typical example.

1.1 Base Subject/Observer classes
In the Subject/Observer pattern, a “subject” is an object that transmits messages about changes to its state, while an “observer” is an object that receives such messages. These can be in a many-to-many relationship in that a subject may have many observers and an observer may be monitoring several subjects. For the purposes of this paper, let us assume that the following module definition provides the core functionality of the subject/observer pattern:

```java
module core {
    type Event = { val msg: String }
    class Subject (val id: String) {
        val observers = new Set[Observer];
        func addObserver (observer: Observer): Void = observers += observer;
        func notifyObservers (ev: Event) =
            for (observer < observers)
                observer.notify(self, ev)
    }
    partial class Observer {
        abstract func notify (subj: Subject, ev: Event): Void
    }
    class Logger (val log: Stream) extends Observer {
        impl func notify (subj: Subject, ev: Event) =
            log.printf("event from %s: %s (size=%d)
                subj.id, ev.msg, subj.modelSize())
    }
}
```

This code is expressed in P3, a research programming language similar to C++ [6], Java [4] and Scala [5]. A technical report on P3 is in progress but not currently available, although related technical reports are on the author’s website [8]. Instead, P3 will be explained as needed. In particular, assume that a module construct provides a named scope for the elements within. A module is also recursive, so it permits Subject to refer to Observer before the latter is defined. A type definition creates an alias for the type expression to the right of the equals sign. A class definition for C creates (1) a type C, (2) a constructor “new C(...)” that creates instances of the class and (3) certain (unspecified) mechanisms for extension. If a class is marked partial, it has no new operator. Variable “self” is the special name used within a class to refer to the future object (like “this” in C++, Java or Scala). References to siblings within a class body are via self, although syntactic sugar allows it to be implicit (as shown here).

Module definition (1.1) creates an object named “core” such that core.Subject is a type and core.Observer and core.Logger are classes. The core.Subject class defines a field containing a mutable set of observers, a way to add to that set and a method, notifyObservers, that loops through the set of observers and calls each one’s notify method (passing itself as the first parameter). The core.Observer class defines a single abstract method, notify. The core.Logger class extends core.Observer and implements the notify method. The core.Subject and core.Observer classes are meant to be extended.

Since the contents of object core include types and classes, core is what P3 calls a hybrid, as it is a cross between a regular record and a type group. Type groups are record-like structures containing type definitions; they are similar to Bruce’s type groups [1].

1.2 Extensions of the base classes
Now consider the following module that extends the core classes to implement a model/view/controller pattern:

```java
module mvc {
    class Subject (id: String) extends core.Subject(id) {
        func modelSize(): Int = ...; ... }
    partial class Observer extends core.Observer { ... }
    class Logger (log:Stream) extends core.Logger(log) with Observer {
        override func notify (subj: Subject, ev: Event) =
            log.printf("event from %s: %s (size=%d)\n", subj.id, ev.msg, subj.modelSize()) /* error */
    }
}
```

The ellipses stand for enhancements to these classes that are not relevant to the example. Note that the Subject class has a new method, modelSize, and that the Logger class has overridden its notify method to take advantage of the new method.

1.3 Failure of type safety
However, (1.2) would fail to type because Logger inherits its declaration of notify from mvc.Observer, which inherits it from core.Observer and, so, notify’s fully qualified declaration is as follows:

```java
```

¹Copyright © 2014 by Peter Vanderbilt. This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit creativecommons.org/licenses/by-nc-nd/4.0/.
In particular, the subj parameter of has static type core.Subject which does not have a modelSize method, and so the expression subj.modelSize() would fail to type. An attempt to change the type of subj would fail since it is a contravariant parameter.

The real problem is that (1.2) is not safe. Consider the following code fragment:

```java
val logger = new mvc.Logger(...);
val subj = new core.Subject;
subj.addObserver(logger);
subject.notifyObservers(...)
```

Because mvc.Logger indirectly extends core.Observer, subclass polymorphism allows the logger object to be passed to subject.addObserver(). The call of the last line fails because subject calls logger.notify(subject,...), which calls subject.modelSize(), which does not exist.

Thus the failure of typing correctly disallows the implementation of notify in (1.2) exactly because it is unsafe. However this means that mvc objects cannot use the mvc enhancements, which is contrary to the point of extension. What is needed is a mechanism that allows parallel extension like (1.2), but in a safe way. The next section presents one such mechanism.

2. A solution in P3: Open modules

In P3, the simplest mechanism for achieving extensible mutual recursion is the open module. In contrast to the closed modules of (1.1) and (1.2), open modules are designed to be extended in a way that maintains relationships. In particular, open modules use inheritance and dynamic dispatch so that references to siblings vary with extension.

2.1 Subject/Observer as open modules

Let us reconsider the Subject/Observer example using open modules. The following module definition is an open version of the core definition of (1.1):

```java
open module core { (2.1)
  virtual type +Event = { val msg: String }
  virtual class +Subject (val id: String)
  { /* body of Subject in (1.1) */ }
  virtual partial class +Observer
  { /* body of Observer in (1.1) */ }
  virtual class +Logger (val log: Stream) extends +Observer
  { /* body of Logger in (1.1) */ }

}
```

Similarly, the following module definition is an open version of the mvc definition of (1.2):

```java
open module mvc extends core { (2.2)
  refine class +Subject (id: String)
  extends msuper.Subject(id)
  { /* body of Subject in (1.2) */ }
  refine partial class +Observer
  extends msuper.Observer
  { /* body of Observer in (1.2) */ }
  refine class +Logger (log: Stream) extends +Observer
  extends msuper.Logger(log) with +Observer
  { /* body of Logger in (1.2) */ }

}
```

The bodies of the classes are the same as the corresponding ones in (1.1) and (1.2), so to save space, they are elided. The error indication is not carried forward. Technically, the Logger class must be marked `fragile`, as will be discussed in §4.5, but let us ignore that issue for simplicity.

2.1.1 Explanation

The keyword open at the start of a module, instead of nothing or closed, signals the use of the open variant. The virtual keyword at the start of a field descriptor indicates that it has an implementation which may be overridden. The “+” before a type or class name indicates that its overrides must be a subtype or subclass (as appropriate); this and “with +” will be described further in §2.2.3.

The “extends” core clause on mvc’s definition indicates that it will inherit from core. The keyword refine at the start of a field indicates that it is an override (thus replacing a field that would otherwise be inherited) and has a refined future type. The use of msuper in a class’s extends clause will be discussed in §2.2.5. The Event type of mvc inherits from core since it is not overridden.

To avoid confusion with containment, P3 uses the terms “parent module” and “child module” instead of “super-module” and “sub-module.” Also an “ancestor module” is either the module itself or the parent of an ancestor; similarly for “descendant” and the adjective “proper” can be used to mean “but not itself.” Finally, the term “sibling” is used to refer to a peer entity.

2.1.2 Semantics and typing

Semantically, the core and mvc objects created by (2.1) and (2.2) are functionally equivalent to those created by (1.1) and (1.2) in that they have the same runtime code.

However, while (1.2) as written fails type checking (for good reasons as discussed in §1.3), (2.2) passes type checking. The difference is that, with open modules, mvc.Logger does not extend core.Observer, but rather a shadowed class with the same implementation but with mvc types. In particular, the inherited declaration of notify (which is what failed typing in (1.2)) is as follows:

```java
func notify (subj: mvc.Subject, ev: mvc.Event): Void (2.3)
```

Thus subj has the modelSize() method and, so, type checking goes through.

Actually, the situation is more general than as just described: with open modules, the typing of module bodies is done in a parameterized context. In particular, references to siblings are qualified by a special variable called “mslf” (instead of being qualified by “core” or “mvc”). In this case, the inherited declaration of notify is as follows, in contrast to (1.3) and (2.3):

```java
func notify (subj: mself.Subject, ev: mself.Event): Void (2.4)
```

The type of mself is derived from programmer-supplied annotations. In (2.2), these annotations are the plus symbols before the type and class names, which gives mself the following declaration:

```java
val mself: {
  type Event <:: { val msg: String }
  type Subject <:: {
    val id: String;
    val observers: Set[mself.Observer];
    func addObserver (mself.Observer): Void;
    func notifyObservers (mself.Event): Void;
    func modelSize(): Int
  }
  type Observer <:: {
    func notify (mself.Subject, mself.Event): Void
  }
  type Logger <:: {
    val log: Stream;
    func notify (mself.Subject, mself.Event): Void
  }
}
```
Not shown are the enhancements elided in (1.2). Variable \texttt{mself} ranges over hybrid objects and is constrained by the hybrid type to the right of the colon. It is a dependent typing since \texttt{mself} appears in its own type. Variable \texttt{mself} will be discussed further in §2.2.5 and its type will be introduced in §2.2.4 and defined in §3.3.5. Declaration (2.5) will be justified in §3.3.7.

When \texttt{mvc} is typed, the notify methods, including those inherited from the core classes, all have the declaration given by (2.4). The implementation of notify in \texttt{mvc.Logger} (see (1.2)) is well typed because variable \texttt{subj} has type \texttt{mself}. Subject which is a subtype of a type that has an appropriate id and modelSize fields. This will be shown more formally in §3.3.7.

When core is typed, \texttt{mself} would have the same type as in (2.5) except without the modelSize declaration nor the elided enhancements. Thus the type of \texttt{mself} in (2.5) is a subtype of the type of \texttt{mself} when typing core. This is a requirement on extension and is what allows core’s implementations to inherit correctly into mvc. This will be discussed further in §3.4.

2.1.3 Observations

Note that in (2.5), \texttt{mself.Subject} and \texttt{mself.Observer} are constrained in a mutually-recursive manner. This means that all future modules will have the mutual recursion desired: each module’s Subject class will refer to its Observer class and vice versa.

An open module, in addition to parameterizing its content, also “ties the knot” in that it yields a module-specific implementation, one with \texttt{mself}+\texttt{m} (for module with name \texttt{m}). This is why the declaration of notify in core is given by (1.3), while it has declaration in mvc is given by (2.3).

Safety is ensured because the mvc classes are not subclasses of their core counterparts, nor are their types in a subtype relation. This means that objects from different modules cannot be mixed, which avoids the unsafe situation discussed in §1.3. Thus an extending module gets to use its enhanced functionality.

2.2 Key mechanisms

Now let us consider some of the key mechanisms of P3’s open modules.

2.2.1 Multi-functionality of open modules

An open module definition for name \texttt{m} is “multi-functional” in the sense that it creates multiple entities:

- The module object itself, \texttt{m}.
- A type, \texttt{(?<<m)}, whose instances are extensions of the current module (including itself). This type is discussed in §2.2.4. It is the type of \texttt{mself} as introduced in §2.1.2.
- Entities that support extension and inheritance. These entities are not directly visible to the programmer, except that they support the \texttt{extends} mechanism. Implementations of these entities will be given in the sections on encoding, 3.2.1 and 3.3.5.

In contrast, a closed module definition creates only \texttt{m}.

2.2.2 Field descriptors

The body of an open module has field descriptor definitions rather than plain definitions as appearing in a closed module. Field descriptor definitions are similar to those phrases found in Java or C++ that start with keywords like \texttt{virtual}, \texttt{abstract} or \texttt{final}. For this paper, all field descriptors are of virtual fields, which means that they have implementations and can be overridden.

2.2.3 Current and future aspects

Because open modules allow fields to be overridden, there needs to be a way to restrict how entities can be overridden in a child module. In P3 this is called the future aspect of a field, in contrast to the current aspect which describes the implementation of a field. The field definitions of a closed module have only current aspects, as those fields cannot be overridden.

For data entities, the future type is the same as the current one. For type definitions, the future aspect is typically some bound on the type’s future value, or no bound at all. For classes, the future aspect is typically a constraint on the type defined by the class, but other specifications are possible (for example see §4.4).

The future types create an invariant over future modules. They allow implementations to be constructed that remain well typed when inherited into future modules. Future types should be restrictive enough to permit such implementations, but not so restrictive as to disallow useful extensions.

In P3, one can write field descriptor definitions using either an explicit current/future form or certain higher-level forms. The explicit descriptor is of the form “\texttt{current \texttt{J}, future \texttt{L}...}” where \texttt{J} is a typed definition specifying the implementation and \texttt{L} is a declaration constraining future overrides.

Among the higher-level forms are the following three for type definitions with their equivalent current/future forms:

\begin{align*}
\text{virtual type } +X &= T \quad \text{current type } X = T; \\
\text{virtual type } X; Z &= T \quad \text{current type } X = T; \\
\text{virtual type } X &= T \quad \text{current type } X = T; \\
\end{align*}

All specify that the current value of type \texttt{X} is \texttt{T}. The first specifies that any future value must be a subtype of the current value (\texttt{mdef} will be described in §2.2.5) while the second has an explicit type \texttt{T} and the third has no constraint.

The following are two higher-level forms for classes with their equivalents:

\begin{align*}
\text{virtual class } +C ... &\approx \text{current class } C \text{ selftype } mself.C ...; \\
\text{virtual class } +C ... &\approx \text{current class } C \text{ selftype } mdef.D selftype } mself.C ...; \\
\end{align*}

Recall that class \texttt{C} defines a type \texttt{C}, so “future type \texttt{C<<C}” requires that any future module has a class \texttt{C} whose type is a subtype of the type of \texttt{C}, i.e. \texttt{C} extends \texttt{C}. Note that \texttt{C}’s other aspects, like its constructor, are unconstrained and cannot be relied on; this will be discussed further in §4.4.

The first equivalence of (2.7) specifies that a class with a plus before its name is equivalent to one whose current aspect is the part without the plus and whose future value must be a class that extends the current class. The second specifies a class, \texttt{C}, that is in “diamond inheritance” with its sibling class \texttt{D}: the current class extends the current \texttt{D} and its future class must extend both the current \texttt{C} and the future \texttt{D}. Both current classes have a \texttt{selftype} specification which narrows the type of self. Special variables \texttt{mself} and \texttt{mdef} are described in §2.2.5.

2.2.4 The “extends” operator

P3 defines an operator “\texttt{<<}” called “extends,” such that \texttt{p<<m} if-and-only-if the fields of module \texttt{p} satisfy the future types of open module \texttt{m}. Informally, we say that “\texttt{p} is an extension of \texttt{m}.” Extends is required to be reflexive, so \texttt{m<<m}, and transitive.

One of the entities created by a definition of open module \texttt{m} is a type with the special P3 syntax “\{?<<m\}” whose instances are those modules in the extends relation with \texttt{m}. Thus \texttt{p}\{?<<m\} is equivalent to \texttt{p<<m}.
For example, definition (2.1) creates the type {{\\texttt{?<\texttt{core}}} which is such that core\\texttt{<<}core and mvc\\texttt{<<}core. Module definition (2.2) creates the type {{\\texttt{?<\texttt{mvc}}} such that mvc\\texttt{<<}mvc (but not core\\texttt{<<}mvc). Note that type {{\\texttt{?<\texttt{mvc}}} is a subtype of {{\\texttt{?<\texttt{core}}}} by transitivity of \ll.

2.2.5 Special variables

Another key mechanism of open modules are the special variables \“mself\”, \“msuper\” and \“mdef\” which may appear in the bodies of modules. They are similar to self (or this) and super for classes and start with \“m\” to make nesting classes within modules easier.

\textbf{mself:} Variable \“mself\” may appear in implementations and types and provides the main way for entities in a module’s body to reference one another in a way that preserves mutual recursion. In particular, unqualified references to entities within a module are syntactic sugar for references qualified by mself. For example, in (2.1) and (2.2), the references to Observer in the subject class are really to mself.Observer and references to Subject in observer are to mself.Subject. In fact, for the purposes of this paper, entities in the body of module \textit{m} may not reference \“m\” directly.

Variable mself ranges over future modules which, for module \textit{m}, are those modules whose fields satisfy \textit{m}’s future declarations, so mself<<\textit{m}. Because the implementations of module \textit{m} are typed with respect to mself<<\textit{m}, \textit{m}’s implementations can be inherited into any future module. This will be made concrete in the encodings of §3.2.1 and §3.3.5.

Within the module object \textit{m} created by a open module definition, mself refers to \textit{m}. For example, in the core object created by (2.1), mself refers to core and has the declaration given by (1.3). Similarly, within the mvc object created by (2.2), mself refers to mvc and has the declaration given by (2.3), even though it was inherited from core. This gives the desired extensible mutual recursion for modules as discussed in §2.1.3.

\textbf{msuper:} For extending modules, there is also a special variable \“msuper\” that refers to the implementations from the parent module. It may appear in implementations and current types, but not future types. Notice that, in (2.2), the Subject class extends msuper.Subject, not core.Subject, and the other classes are similar.

Variable msuper refers to msuper-parameterized implementations, in contrast to direct references to the parent module, so implementations inherit appropriately. The encodings of §3.3.1 and §3.3.5 will give concrete definitions of msuper and §3.4 will provide further discussion.

\textbf{mdef:} There is a third special variable, \“mdef\,” that refers to the current implementations of a module. It may appear in implementations and types. It too refers to msuper-parameterized implementations, in contrast to direct references to the current module. In fact a module’s mdef is the same as an extending module’s msuper.

A key use of mdef is in \textit{extends} clauses to refer to the current implementation of a sibling class, not some future, unknown implementation as denoted by mself. For instance, in core, the Logger class extends mdef.Logger, not mself.Logger, because it extends the Observer class defined in core, not all possible future implementations of core’s Observer. Similarly, while the mvc Logger class extends msuper.Logger, it also extends mdef.Logger.

Variable mdef is also used in future types to refer to the current implementation. For example, as described in §2.2.3, the \“+X\” and \“+C\” forms of field descriptors have future types that are subtypes of mdef.X or mdef.C, respectively.

2.3 The example in canonical form

Let us revisit the example and see how it would be expressed in canonical form. First, consider the canonical form of the core module, as defined by (2.1):

\begin{verbatim}
open module core {
  current {
    type Event = { val msg: String };
    class Subject (val id: String) selftype mself.Subject {
      val observers = new Set<mself.Observer>;,
      func addObserver (observer: mself.Observer): Void = observers += observer;
      func notifyObservers (ev: mself.Event) =
        for (observer < - observers)
          observer.notify(self, ev)
    }
    partial class Observer selftype mself.Observer {
      abstract func notify (mself.Subject, mself.Event);
      class Logger (val log: Stream) extends mdef.Observer selftype mself.Logger {
        impl func notify
          (subj: mself.Subject, ev: mself.Event)
          = log.printf("event from %s: %s\n", subj.id, ev.msg)
    }
  }
}

future {
  type Event <: mdef.Event;
  type Subject <: mdef.Subject;
  type Observer <: mdef.Observer;
  type Logger <: mdef.Logger \cap mdef.Observer
}
\end{verbatim}

Notice that free sibling references have been explicitly qualified to mself (see §2.2.5) and that field descriptors have been split into \textit{current/future} form (see §2.2.3). Also note that the current Logger class extends mdef.Observer (which is the current Observer class as defined immediately above Logger), while the future Logger type extends mdef.Observer (the future Observer).

The mvc module, as defined by (2.2), is similar:

\begin{verbatim}
open module mvc extends core {
  current {
    class Subject (id: String) extends msuper.Subject(id) selftype mself.Subject
      { func modelSize(): Int = ... ...
      }
    partial class Observer extends msuper.Observer selftype mself.Subject { ... }
    class Logger (log: Stream) extends msuper.Logger(log) with mdef.Observer selftype mself.Logger {
      override func notify
        (subj: mself.Subject, ev: mself.Event)
        = log.printf("event from %s: %s\n", subj.id, ev.msg, subj.modelSize())
    }
  }
}

future {
  type Subject <: mdef.Subject;
  type Observer <: mdef.Observer;
  type Logger <: mdef.Logger \cap mdef.Observer
}
\end{verbatim}

Note that the current classes extend their parent classes via msuper.

3. Encoding and typing modules

This section discusses the encoding and typing of module definitions. The encoding of a module is how it can be represented in lower-level terms. The typing of a module is a logic rule that ex-
presses when the module is well-formed and what can be inferred after its definition.

Handling the general case of open modules is difficult, so let us proceed in stages: first closed modules, then open top-level modules and finally open modules in general.

3.1 Encoding and typing closed modules

To set the stage, let us first consider the encoding and typing of closed modules. This will also introduce some of the key concepts and notations of P3.

3.1.1 Encoding closed modules

A closed module can be encoded as a recursive object:

\[
\text{[closed] module } m \{ \bar{J} \} \quad \mapsto \quad (3.1)
\]

\[
\text{let } \bar{x} = \text{namesDefinedBy}( \bar{J} ) \quad \text{in} \quad \text{val } m = \mu(m) \{ \bar{J} \}
\]

The encoding arrow, \( \mapsto \), indicates that definitions matching the pattern to its left can be encoded by instantiating the pattern to its right. Pattern variable \( \bar{m} \) ranges over identifiers, variables \( \bar{J} \) and \( \bar{J} \) range over lists of typed definitions and variable \( \bar{x} \) ranges over identifier lists. The (blue) square brackets indicate that \( \text{closed} \) is optional. An over-bar indicates a list and, in this case, \( \bar{J}, \bar{J} \) and \( \bar{x} \) are all parallel. The \text{let} clause binds a couple of pattern variables for use in the \text{in} part. The meta-function \text{namesDefinedBy} extracts the names defined by the list of definitions given as its argument. Operator \( \mu \) creates recursion and a curly-bracket-enclosed definition list creates a record; a P3 object is a recursive record with closures. Object \( m \) is in general a hybrid (as discussed in §1.1) as it may have both data and type fields. Each \( \bar{J} \) of \( J \) can be any form of definition, including those starting with \text{val}, \text{func}, \text{type}, or \text{class}.

In general, the phrase \( "c\langle \bar{d} \rightarrow \bar{e} \rangle" \) denotes a substitution that yields expression \( \bar{e} \) with all occurrences of free variables \( \bar{d} \) replaced by corresponding expressions from \( \bar{d} \). In this case, \( "\bar{J}\langle \bar{x} \rightarrow \bar{m}\bar{x} \rangle" \) denotes the substitution such that each free variable, \( x \in \bar{x} \), in definition \( \bar{J} \) is replaced by the corresponding expression, \( m \in \bar{m} \). Thus \( \bar{J} \) is the list of definitions in the body of \( m \) where all free references to sibling entities are explicitly qualified to \( m \). Thus, if encoding (3.1) were applied to (1.1), this substitution arranges that the references to “Subject” in the Observer class would be qualified to “core:Subject” and vice versa.

\textbf{Explanation:} Basically (3.1) says that the definition of the first line can be encoded as a definition of a recursive object, \( m \), which contains the definitions from the body with sibling references qualified to \( m \).

3.1.2 Typing closed modules

The following is a simplified typing rule for closed modules:

\[
\text{MOD-CLOSED:} \quad \frac{\bar{x} \triangleq \text{namesDefinedBy}( \bar{J} ) }{\Gamma; \text{val } m : \{ \bar{L} \} \vdash \bar{J}(\bar{x} \rightarrow \bar{m}\bar{x}) \Rightarrow \bar{L} } \quad (3.2)
\]

Logic variable \( m \) ranges over identifiers, \( \bar{J} \) over lists of typed definitions, \( \bar{L} \) over lists of declarations (parallel to \( J \)) and \( \bar{x} \) over identifier lists (also parallel to \( \bar{J} \)). The term \( \"\{ \bar{L} \}\" \) is a record type. The phrase \( \"\{ \bar{L} \}\Rightarrow \bar{L} \) is a P3 judgement that states that definition \( \bar{J} \), when executed in a context satisfying environment \( \Gamma \), results in a binding satisfying declaration \( \bar{L} \). The operator \( \tau \) is P3’s mechanism for dependent types and is such that \( p:\tau(x)\Rightarrow \bar{p} \) implies \( p\tau(x \rightarrow \bar{x}) \) (for path \( \bar{p} \)). This rule is simplified in that it ignores kinds and the associated premises that ensure that \( \bar{L} \) are well-formed declarations.

Note that logic variable \( \bar{L} \) does not appear in the module definition to the left of :\( \Rightarrow \) in the conclusion. For the purposes of this paper, assume that it can be determined from annotations appearing in \( J \) or by inference (not involving the recursion variable \( m \)).

\textbf{Explanation:} The rule says that, if the two premises are true, then after the execution of \( \"\text{module } m \{ \bar{J} \}\" \), a new variable \( m \) has been bound with a value satisfying the type \( \tau(m)\{ \bar{L} \} \). The first premise simply defines \( \bar{x} \) to be the list of variables defined in the body of the module. The second premise requires that each definition, \( \bar{J} \), in the body of the module yields a binding satisfying the corresponding declaration \( \bar{L} \).

\textbf{Justification:} This rule can be justified by replacing the module definition to the left of :\( \Rightarrow \) in the conclusion with its encoding and showing that the premises lead to the (new) conclusion. Given the encoding for closed modules, (3.1), this comes down to showing that the recursion, \( \mu(m)\{ \bar{J} \} \), has type \( \tau(m)\{ \bar{L} \} \), which follows directly from the second premise.

3.2 Encoding and typing open top-level modules

The previous section described typing and encoding of closed modules. Now let us consider open top-level module definitions. These are “top level” in the sense that they do not extend another module. The encoding and typing of these are nice in that they can be expressed in terms of the components of the module, without the need for auxiliary type definitions.

3.2.1 Encoding open top-level module definitions

The following encoding applies to open module definitions:

\[
\text{open module } m \{ \bar{J} \} \quad \mapsto \quad (3.2)
\]

\[
\text{let } \bar{x} = \text{namesDefinedBy}( \bar{J} ) \quad \text{in} \quad \text{val } m = \mu(m) \{ \bar{J} \}
\]

\textbf{Pattern variable \( \bar{m} \) ranges over identifiers, \( \bar{J} \) ranges over field description definition lists, \( \bar{x} \) ranges over identifier lists, \( \bar{J_1} \) ranges over definition lists and \( \bar{L_1} \) ranges over declaration lists. As mentioned above, the field descriptors, \( \bar{J} \), may have only virtual fields. Module name \( m \) may not appear free in \( \bar{J} \) (except through an outer recursion, which will not be discussed further). The substitution is as described in §3.1.1, except that \( m \) is used in place of \( m \). The rewrite to current/future forms is as given in §2.2.3 and instantiates pattern variables \( \bar{J} \) and \( \bar{L} \).

This encoding requires that \( m \) does not appear free in \( \bar{L} \). This restriction is contrary to the example and will be lifted in §3.3. This encoding is also simplified in that it does not deal with kinds nor does it have sufficient type annotations.

\textbf{Explanation:} Encoding (3.2) maps a top-level open module definition to a series of three definitions. The first defines the type \( \{ ?\langle m \rangle \} \) (see §2.2.4) to be a record type, dependent over \( m \), of the future declarations, \( \bar{L}_f \).

The second entity is a “module constructor” function, \( m\_c \), that abstracts over \( m \) and yields a record formed from the current definitions, \( \bar{J}_c \). It is recursive in \( m \) to tie the sibling implementations together as specified. It is parameterized by \( m\_c \) to allow the implementations to be inherited.

This module constructor, \( m\_c \), will be needed for extension: a module, \( f \), extending this one (so \( f \langle m \rangle \) will be constructed by
calling \(m_{-c}(f)\) (perhaps indirectly) in order to create the entities needed for inheritance. This will become more clear when we consider non-top-level modules in §3.3.

The final entity created by encoding (3.2) is the actual module definition, \(m\), which is the object formed by recursively connecting \(mself\) to \(m\) in the body of the function. Note that \(m_{-c}\) provides a parameterized implementation and the last line “ties the knot.”

### 3.2.2 Typing open top-level module definitions

The following rule handles open top-level module definitions:

\[\text{MOD-OPEN-TL:}\]

\[
\begin{align*}
\text{let } m \text{ _-c } & \triangleq \text{namesDefinedBy}(\vec{\mathcal{J}}) \\
\vec{\mathcal{J}} & \triangleq \mathcal{J}(\vec{x} \mapsto \text{mself}, \vec{x}) \approx \text{current } J_0; \text{future } L_f \%
\text{val mslef}: \{L_f\}; \text{val mdef}: \{L_c\} \vdash L_c \Rightarrow L_c \\
\text{Γ} & \vdash \text{open module } m \{\vec{\mathcal{J}}\} : \Rightarrow \Gamma_c
\end{align*}
\]

The pattern variables of (3.2) are logic variables of this rule, with the addition of \(L_c\) which ranges over declaration lists. The rewrite to current/future form is as given in §2.2.2 and instantiates logic variables \(J_i\) and \(L_f\) from \(\vec{\mathcal{J}}\). It must be that \(L_c\) can be determined from annotations appearing in \(J_i\) or by inference (not involving the recursion variables mself nor mdef). Lists \(J_i, L_c, L_f\) are all parallel, which is enabled by the condition that all fields are virtual (and so have both implementation and future type). A judgement “\(L_1 \ll L_2\)” indicates that declaration \(L_1\) implies \(L_2\); it is like a subtype relation, except on declarations. The logic variable \(\Gamma_c\) is defined as follows:

\[
\Gamma_c \triangleq \begin{cases} 
\text{type oper } \{\{< <m\}\} & \leftrightarrow \tau[mself] \{L_c\} \\
\text{val } m_{-c} & \triangleq \tau[mdef] \{L_c\} \\
\text{val } m & : \tau[m] \{\text{mself}, \text{mdef} \Rightarrow m, m\} 
\end{cases}
\]

The first declaration gives the value of the “extends” operator for \(m\) (see §2.2.4). The second gives the type of the module constructor function \(m_{-c}\); it is a dependent type (as signaled by “*”) as \(mself\) and the equivalences of §2.2.3 are used to get it from the current/future form, resulting in (2.8). Logic variables \(J_i\) and \(L_f\) are instantiated to the current and future aspects of core:

\[
\begin{align*}
& \text{\#core } \triangleq \mathcal{F}[\text{the part of (2.8) within current\{\}} \#/ \\
& \text{\#future } \triangleq \mathcal{F}[\text{the part of (2.8) within future\{}} \#/
\end{align*}
\]

Then the body of the encoding is instantiated, yielding the following definitions as being equivalent to core’s definition, (2.1):

\[
\begin{align*}
& \text{type oper } \{\{< <\text{core}\}\} = \tau[mself] \{\text{\#core}\} \#/ \\
& \text{func } \text{core}_{-c}(\text{mself}<<\text{core}) = \mu[\text{mdef}] \{\text{\#core}\} \#/ \\
& \text{val } \text{core} = \mu[\text{core}] \triangleq \#/(\text{\#core}) 
\end{align*}
\]

The first defines \(\{\{< <\text{core}\}\}\) in terms of the future declarations of core, the second defines the module constructor core_{-c} in terms of core’s current definitions and the last defines the module core. The first definition is actually not well formed because \(\text{\#core}\) contains free references to mdef in contradiction to the restriction on (3.2); this will be addressed in §3.3.7.

### Typing:

Now let us consider the typing of core, as defined by (2.1), by applying the typing rule, MOD-OPEN-TL, for open top-level modules as defined in §3.2.2. This instantiates logic variable \(m\) to “core” and \(\vec{\mathcal{J}}\) to the body of core. The first two premises of the rule instantiate \(x, J_i, L_f\) as defined by (3.4), (3.5) and (3.6). The list of current declarations, \(L_c\), is extracted from \(J_c:\)

\[
\begin{align*}
& \text{\#core} \triangleq \begin{cases} 
\text{type } \text{Event} & \leftrightarrow \{\text{val } msg: \text{String}\}; \\
\text{class } \text{Subject} (\text{String}) & \text{selftype } \text{mself}.\text{Subject} \{ \\
& \text{val } \text{id}: \text{String}; 
\end{cases}
\end{align*}
\]

A Mechanism for Extensible Mutual Recursion, version 1.0.1
val observers: Set[mself.Observer];
func addObserver (mselfObserver): Void;
func notifyObservers (mself.Event): Void
}
partial class Observer selftype mself.Observer {
  abstract func notify (mself Subject, mself Event): Void
}
class Logger (Stream) selftype mself.Logger extends mdef Observer {
  val log: Stream
}

For convenience, let us also define \( \Gamma \) to capture the declarations of mself and mdef:

\[
\Gamma_\text{core} \triangleq \begin{cases} \text{val } mself: \{ L_\text{core} \} \\
\text{val } mdef: \{ F_\text{core} \} \end{cases} \quad (3.9)
\]

Note that these are mutually dependent declarations as both mself and mdef may appear in both \( L_\text{core} \) and \( F_\text{core} \).

The third premise, which ends \( J_\text{core} \rightarrow L_\text{core} \), ensures that field implementations (those in the current part of (2.8)) satisfy their declarations (as in (3.9)). For Event, this is trivial. For the classes, showing this is straightforward, except for a couple of places:

- The notifyObserver implementation of class Subject must satisfy its declaration:

\[
\Gamma; L_\text{core} \vdash \text{func } notifyObservers \text{ (ev: mself.Event) } = \\
\text{for (observer } \leftarrow \text{self observers) } \\
\{ \text{observer.notify(self, ev) } \} \\
\implies \text{func } notifyObservers (mself.Event): \text{Void}
\]

Sibling references within the method have been explicitly qualified to self. Variable self has type mself Subject instead of the usual mdef Subject because of the selftype specification. However, self can be widened to mdef Subject because mself \( \{ L_\text{core} \} \) implies mself Subject \( \vdash mdef Subject \). Thus self observers have type Set[mself Observer] (by mdef \( \{ L_\text{core} \} \)), so loop variable observer has type mself Observer, which is a subtype of mdef Observer, which has a notify method as declared by (2.4). Thus the call of observer notify(self, ev) is well typed. Note that this typing depends on the selftype qualification on Subject.

- Class Logger’s implementation of notify is similarly type correct for mself Subject and mself Event.Event can be widened by the future type to their mdef counterparts which have the appropriate fields by the current type.

The fourth premise, which ends \( L_\text{core} \vdash L_\text{core} \), ensures that the current declarations (see (3.8)) imply the future ones (those in the future part of (2.8)). For Event, the premise is the following:

\[
(\text{type } \text{Event } \leftrightarrow \{ \text{val } \text{msg: String} \} ) \vdash (3.10) \\
(\text{type } \text{Event } \vdash \text{mdef } \text{Event})
\]

The first line is also the declaration of mdef Event, so (3.10) comes down to showing that “\( \{ \text{val } \text{msg: String} \} \)” is a subtype of itself, which is trivial. While Subject, Observer and Logger have class declarations, they imply type declarations analogous to the first line of (3.10) and, too, come down to showing that a type is a subtype of itself. Actually, the condition doesn’t hold for Logger, which is why it should really be marked fragile as discussed in §4.5.

The last premise defines \( \Gamma_\text{core} \):

\[
\Gamma_\text{core} \triangleq \begin{cases} \text{type } \text{oper } \{ \{ \text{val } \text{msg: String} \} \rightarrow \tau \text{mself} \} \{ L_\text{core} \} \\
\text{func } \text{core}_\text{c} \text{ (mself:} \{ \langle \mu \text{mdef} \rightarrow \text{core} \} \\
\text{val } \text{core} : \tau \text{core} \{ L_\text{core} \{ \text{mself, mdef} \rightarrow \text{core, core} \} \}
\end{cases} \quad (3.11)
\]

These declarations are yielded by the definitions of (3.7) and represent the net effect of module definition (2.1). In the type of core given by the last line, the selftype qualifications appearing in the class declarations drop out because each self-type is the same as the corresponding class type.

Thus rule MOD-OPEN-TL of §3.2.2 establishes that the definition of open module core, as given by (2.1), results in bindings satisfying \( \Gamma_\text{core} \) as defined by (3.11).

3.3 Encoding and typing open modules in general

Now let us consider the encoding and typing of open modules in general, not just top level. These are considerably more complicated, because they need to deal with inherited fields and a more complex internal structure, so let us take it in steps. The first subsection discusses the basic form of the encoding and the next three discuss the internal structure of modules and module types. The next two subsections define an encoding and typing rule for open modules, and the following subsection applies them to the mvc example. We end with a discussion about why open modules work for the example, while straight inheritance does not.

3.3.1 Core of the encoding

A key feature of P3 open modules is that they parameterize their content over future modules. In particular, within a module, references to future (or “late-bound”) sibling fields are qualified to the special variable mself and the module’s body is abstracted with respect to it, so that it can be instantiated to each future module.

For module m extending module n, this abstraction is realized by the “module constructor” function \( m \rightarrow c \), whose general form is as follows:

\[
\begin{align*}
\text{func } \text{m}_\rightarrow \text{c} \text{ (mself } \rightarrow \text{m} \text{) } = \mu \text{mdef} \\
\text{let } \text{val } \text{m} \text{super } = n \rightarrow \text{c} \text{ (mself) } \\
\text{in } \text{m} \text{super with } \{ \text{m}_{\rightarrow \text{c}} \}
\end{align*}
\]

The parent module, \( n \), must have been encoded similarly and, so, has constructor function \( n \rightarrow c \). The operator \( \text{with} \) applied to two record values unifies the fields of the values with those of the right value overriding those of the left. Similarly \( \text{with} \) applied to two record types unifies the declarations with those of the right type overriding. \( n \rightarrow c \) denotes the current definitions of \( m \) with sibling references qualified to mself as discussed above. For a top-level module, (3.12) simplifies to \( m \rightarrow c \) of (3.2).

Current (or “early-bound”) sibling references are qualified to mdef and recursion over it is used to make this linkage. Within this recursion, the parent module’s constructor, \( n \rightarrow c \), is called to create \( n \)'s part of the module, which is bound to the special variable \( \text{m} \text{super} \), which is used both to provide inheritance for any field not overridden and to satisfy references to it within the implementations, \( n \rightarrow c \).

Constructor function \( m \rightarrow c \) is used in two ways. First, it is used to construct module \( m \):

\[
\text{val } m = \mu \text{mdef} (m \rightarrow c)(m)
\]

This builds an object bound to \( m \) such that mself refers to \( m \).

Second, the constructor function \( m \rightarrow c \) is used to provide inheritance to any child module. In particular, it is used in any child module’s constructor, just as \( n \rightarrow c \) was used in it. Similarly, that child’s constructor will be used in its child’s constructor and so on.

3.3.2 Internal module structure

The structure described above can lead to a series of nested calls to module constructors up the chain of ancestors, resulting in a series of “slices.” In particular, say there are modules \( m_0 \) to \( m_k \), where \( m_0 \) is top level and each \( m_{i+1} \) extends \( m_i \). Then the following
and the
in
To save space, \textit{mvars} stands for \textit{“mslice, mdef, msuper”}. Each \(J_i\) denotes the list of module \(m_i\)'s current definitions.

The \textit{let*}-clause defines a series of \(k\)-“slice” objects, \(s_0\) to \(s_k\).

The in-clause yields the last one, \(s_k\). Slice \(s_0\) is as if created by \(m_0\_c(m_1)\) and slices \(s_0\) to \(s_i\) as if created by \(m_{i-1}\_c(m_i)\).

Within each slice, \(s_i\), mself refers to \(m_k\), mdef refers to itself, \(s_i\), and, except for \(i=0\), msuper refers to the “parent” slice, \(s_{i-1}\).

Note that slice objects \(s_0\) to \(s_{k-1}\) contain shadowed entities from modules \(m_0\) to \(m_{k-1}\). These need to exist because they may be referred to via msuper. For example, in module mvc of (2.2), there will be a shadowed slice that contains the entities defined by module core of (2.1) and each class of mvc will inherit implementations from the corresponding class in the shadowed slice. Also notice that these slice objects exist but are not directly accessible via \(m_k\) (which is \(s_k\)).

### 3.3.3 Exact type of a module given its structure

Given that a module has an internal structure as shown by (3.14), what is its type? Recall that it actually has two types, exact and general, as discussed in §3.2.2; let us start with what is its type? Recall that it actually has two types, exact and general type must incorporate the exact types of the slices. Thus (3.15) declares that object \(m_k\) is slice \(s_k\).


\[
\text{val } m_k = \mu(m_k) \text{ let*}
\]

\[
\begin{align*}
\text{val } s_0 &= \mu(s_0) \{ J_0^0 (\text{mself, mdef} \to m_k, s_0) \} \\
\text{val } s_1 &= \mu(s_1) s_0 \quad \text{with} \quad \{ J_1^0 (\text{mvars} \to m_k, s_1, s_0) \} \\
& \quad \vdots \\
\text{val } s_k &= \mu(s_k) s_{k-1} \quad \text{with} \quad \{ J_k^0 (\text{mvars} \to m_k, s_k, s_{k-1}) \}
\end{align*}
\]

Again mvars=“mself, mdef, msuper.” The “∪” operator denotes union and yields a non-discriminated existential type. Here the unions capture the notion that the slices exist and their values are relevant to its own type and the types of later slices. Type \(\{s_0, s_1, \ldots s_k\}\) is the type of things identical to \(s_k\), which, in this case, means that module \(m_k\) is slice \(s_k\).

Declarations \(L_i^0\) are those capturing the result of the level-\(i\) current definitions, \(J_i\). The part before \textit{with} is a type expressing that all fields inherit from the previous slice (which are overridden by corresponding fields after \textit{with}).

Thus (3.15) declares that object \(m_k\) is identical to slice \(s_k\) of a collection of slices, \(s_0\) to \(s_k\), where the type of slice \(s_i\) follows from its implementation as shown in (3.14).

### 3.3.4 General type of a module given its declarations

The general type of a module is based on its future declarations. Since non-refined fields inherit their future declarations, the general type is a function of the future declarations of \textit{all} the slices. Since each future declaration may have mdef referring to its slice, the general type must incorporate the exact types of the slices. Thus the following is effectively the definition of the general type:

\[
\text{type oper } \{\{?<<m_k\}\} \leftrightarrow \tau[m_k] \quad (3.16)
\]

\[
\begin{align*}
\cup \{ s_0 \colon \{L_0^0 (\text{mdef} \to s_0)\} \\
\cup \{ s_1 \colon \{x_0 \to s_0, x_0\} \quad \text{with} \quad \{L_1^0 (\text{mdef}, \text{msuper} \to s_1, s_0)\} \\
& \quad \vdots \\
\cup \{ s_k \colon \{x_{k-1} \to s_{k-1}, x_{k-1}\} \quad \text{with} \quad \{L_k^0 (\text{mdef}, \text{msuper} \to s_k, s_{k-1})\} \}
\end{align*}
\]

The pattern variables of (3.2) apply here, with the addition of \(n\) which ranges over identifiers and must refer to a previously defined module that has been encoded by this encoding or by the variant for top-level modules given later. The declarationsOf meta-function extracts a declaration from a typed definition.

The first part of the encoding defines four type constructors and types that relate to the exact and general types defined by (3.15) and (3.16). The encoding then uses these types to annotate the module constructor definition (3.12) and the module definition (3.13). This encoding also removes the restriction that future types may not reference mdef. The encoding is as follows:

\[
\text{open module } m \text{ extends } n \{ \bar{J} \} \quad (3.17)
\]

\[
\begin{align*}
\text{let } \bar{x} &= \text{namesDefinedBy}(J) \\
\bar{J}(x \mapsto \text{mself}, \bar{x}) &\cong \text{current } \bar{J}_f; \text{ future } \bar{J}_f \\
\bar{L}_e &= \text{declarationsOf}(\bar{J}_e)
\end{align*}
\]

\[
\text{in } \quad \text{*/ type definitions (3.18) and (3.19) */}
\]

\[
\begin{align*}
\text{func } m_{-c} (\text{mself} <\ll m_f) ; m_{-eTc}[\text{mself}] &= \mu(\text{mdef}) \\
\text{let } \text{val } \text{msuper} : n_{-eTc}[\text{mself}] &= n_{-c} (\text{mself}) \\
\text{in } \text{msuper with } \{ \text{val } \text{msuper} = \text{msuper}, \bar{J}_e \} \\
\text{val } m_{-eT} &= \mu(m) \quad m_{-c}(m)
\end{align*}
\]

The names of these types are built from \(m\) (the module name), an underscore, \(e\) (for "exact," see §3.3.3), \(T\) (for "type") and a number of \(c\)s indicating the number of parameters. A definition of the form "\(\text{type } \bar{X}[\bar{Y}]=T\)" creates a binding from variable \(X\) to the type function \(\bar{Y}[T]\). The type functions defined above are from type groups to hybrid types. When a hybrid is used in a context where a type is expected (as in "\(m_{-eTc}[\text{mself}]\)" in (3.17)), only its type-group projection matters.
The types of (3.17) are such that \( m_\text{eTcc}[m_k,s_i,s_{i-1}] \) is the type of slice \( s_i \) in (3.15) and \( m_\text{eTcc}[m_k,s_i] \) and \( m_\text{eTcc}[m_k] \) are both types of slice \( s_i \) with the additional \( \_\text{super} \) field (which gives access to \( s_{i-1} \) to \( s_0 \)). Type \( m_\text{eT} \) is the exact type of module object \( m_L \). Note that, because \( s_i\leftrightarrow m_k \) and \( s_{i-1}\leftrightarrow s_{i-1} \), there is no need for the explicit unions of (3.15).

The following two types define the "extends" operator for \( m \):

\[
\text{type } m_\text{eTcc}[mself, mdef] = n_\text{gTcc}[mself, mdef, \_\text{super}] \quad \text{with } \{ \bar{L}_f \}
\]

\[
\text{type operator } \{\{<><m]\} = \tau[mself] \quad \text{with } \{ \bar{m}_\text{eTcc}[mself, mdef] \}
\]

The \( g \) is for "general" (see §3.3.4). These two definitions, together with \( m_\text{eTcc} \) from (3.18), represent the parent module and \( \_\text{super} \), so the only explicit union remaining is the bottom one.

**Explanation:** The essential part of encoding (3.17) is to map an open module definition for \( m \) to a function \( m \rightarrow m \_\text{c} \) over mself and a module object \( m \), just as discussed in §3.3.1. The constructor function \( m \_\text{c} \) is crafted so that \( m \)'s implementations, \( \bar{J}_e \), are executed with mdef and msuper bound to appropriate, mself-parameterized entities. Variable mdef provides access to sibling entities, while msuper gives access to those entities defined by the parent module. The module object, \( m \), is basically the fixed point of \( m \_\text{c} \) and, so, "ties the knot" so that mself refers to \( m \). The encoding also defines a bunch of type constructors and types, which are used to give precise type annotations to \( m \_\text{c} \) and other entities as described in the next section.

**Top-level variant:** For an encoding of a top-level module (where there is no "extends" \( n \) clause), everything is the same except there is no \( m_\text{eTcc} \) and \( m_\text{eTcc} \) and \( m_\text{gTcc} \) as follows:

\[
\text{type } m_\text{eTcc}[mself, mdef] = \{ \bar{L}_e \}
\]

\[
\text{type } m_\text{gTcc}[mself, mdef] = \{ \bar{L}_f \}
\]

Also the constructor has no let clause nor use of \( \_\text{super} \):

\[
\text{func } m_\_c (mself<<m) : m_\text{eTcc}[mself] = \mu(mdef) \{ \bar{J}_e \}
\]

Note that in this case, (3.17) is equivalent to a typed version of the top-level encoding, (3.2).

### 3.3.6 Typing open module definitions in general

A typing rule for open modules that extend others requires considerably more formality than does typing closed or top-level open modules. It can be based on encoding (3.17):

\[
\text{MOD-OPEN:}
\]

\[
\bar{x} \triangleq \text{namesDefinedBy } \bar{J} \quad \bar{J}(\bar{x} \rightarrow mself, \bar{x}) \approx \text{current } \bar{J}_e; \text{future } \bar{L}_f
\]

\[
\Gamma_c \triangleq \Gamma^\# \text{ declarations of (3.18) and (3.19) */}
\]

\[
\Gamma^\# \text{ definitions of } \Gamma_v \text{ from (3.20) and } \Gamma_c \text{ from (3.21) */}
\]

\[
\Gamma_c; \Gamma_v; \bar{J}_e := \bar{L}_e; \Gamma_c; \Gamma_v; \bar{L}_f := \bar{L}_f
\]

\[
\Gamma; m_\text{super} : \text{mdef} \rightarrow \text{mself} \rightarrow \text{mSuper}
\]

\[
\Gamma \vdash \text{open module } m \text{ extends } n \{ \bar{J} \} \quad \Rightarrow \quad \Gamma_v; \bar{J}_e
\]

The pattern variables of (3.17) are logic variables of this rule, with the addition of \( \bar{L}_e \), \( \Gamma_c \), \( \Gamma_v \), and \( \bar{J}_e \). \( \bar{L}_e \) ranges over declaration lists and is determined from annotations appearing in \( \bar{J}_e \) or by inference (not involving mself nor mdef). Logic variable \( \Gamma_c \) is instantiated to the list of declarations formed by substituting "e" for "n" in (3.18) and (3.19). Lists \( \bar{J}_e, \bar{L}_e, \) and \( \bar{L}_f \) are all parallel, which is enabled by the condition that all fields are virtual or refinements (and so have both implementation and future type). The last premise is tentative and may change in future research.

Logic variable \( \Gamma_v \) is defined to have the declarations for mself, mdef and msuper used in the main typing clauses:

\[
\Gamma_v \triangleq \text{val mself: } m_\text{gTcc}[mself, mdef]; \text{val mdef: } m_\text{eTcc}[mself, mdef, msuper]; \text{val msuper: } n_\text{eTcc}[mself, msuper]
\]

Note that these declarations are mutually dependent.

Logic variable \( \Gamma_e \) is defined to have the declarations for the module constructor function, \( m \_\text{c} \), and the module itself, \( m \):

\[
\Gamma_e \triangleq \text{func } m \_\_c (*\text{mself}<<m) : m_\text{eTcc}[mself]; \text{val } m : m_\text{eT}
\]

**Explanation:** This rule says that to check that an open, non-top-level module is well typed, follow these steps:

- Instantiate \( m \) to the name of the module, \( n \) to the name of parent module and \( \bar{J} \) to its body. Also extract the names of entities in \( \bar{J} \), qualify sibling references to mself and split the resulting body into current/future form, instantiating \( \bar{J}_e \) and \( \bar{L}_f \). Determine \( \bar{L}_e \) from the annotations on \( \bar{J}_e \), as before.

- Show that the implementations are well typed: that \( \bar{J}_e \) satisfy their corresponding declarations, \( \bar{L}_e \). Show this is an environment containing \( \Gamma_v \) (which contains the declarations of (3.18) and (3.19)) and \( \Gamma_v \) (defined by (3.20)). \( \Gamma_v \) gives the types of mself, mdef and msuper using the types declared by \( \Gamma_e \).

- Show that the current declarations imply the future ones: \( \bar{L}_e \approx \bar{L}_f \). Show this in the same environment discussed above.

- Show \( m_\text{self}: n_\text{gTcc}[mself, msuper] \) which comes down to showing that any overridden future declaration must imply (be a refinement of) the one it overrides. Show this in the environment discussed above.

- Conclude that the module definition yields entities satisfying the declarations given by \( \Gamma_e \) (from (3.18) and (3.19)) and \( \Gamma_v \) as defined by (3.3).

Note that the implementations, \( \bar{J}_e \), are typed with mself free but bounded by mself<<m. This means the implementations are type correct for \( m \) and for any future module correctly extending \( m \). Also note that any entities referenced via mdef or msuper are also parameterized by mself, so, for instance, a class \( C \) extending msuper.C will inherit appropriately mself-parameterized fields. Thus subclass relationships hold in that subclasses yield subtypes and inheritance is sound.

**Justification:** To justify rule MOD-OPEN, replace the module definition in the conclusion by its encoding as given by (3.17) and show that the premises lead to the conclusion. The let-clause of the encoding corresponds to the first two premises of the rule and to the instantiation of \( \bar{L}_e \). The conclusion of the rule asserts that the module definition results in \( \Gamma_v; \Gamma_e \). The declarations of \( \Gamma_v \) follow directly from the first line of the encoding’s in-clause (except for kind checks which are beyond the scope of this paper), and those of \( \Gamma_e \) follow directly from the annotations on \( m \_\text{c} \) and \( m \).

What remains is to show that the last three premises of MOD-OPEN imply that the implementations of \( m \_\text{c} \) and \( m \) satisfy their annotations. Note that \( \Gamma_e \) (as defined by (3.20)) gives the types of mself, mdef and msuper in the body of \( m \_\text{c} \).

The last premise ensures that mself<<n (using msuper as the witness for the union), which ensures that mself is an appropriate argument to n_..c. Thus n_..c(mself) (in the let-clause of m_..c) has type n_..eTcc[mself] since n was encoded by this same encoding.

The premise ending with \( \bar{J}_e \approx \bar{L}_e \) together with the definition of m_..eTcc (and with a little juggling of _super), ensures that the whole let-expression in the body of m_..c has type
A Mechanism for Extensible Mutual Recursion, version 1.0.1

10

Then the body of encoding (3.17) is instantiated, yielding the following definitions as being equivalent to mvc’s definition, (2.2):

\[
\text{type mvc}_e\text{Tc} = \{ \text{val } \bar{x}_{\text{core}} \leftrightarrow \text{msuper.} \bar{x}_{\text{core}} \} \quad (3.27)
\]

\[
\text{type mvc}_e\text{Tc} = \{ \text{val } \text{msuper.} \bar{x}_{\text{core}} \leftrightarrow \text{msuper.} \bar{x}_{\text{core}} \} \quad (3.28)
\]

\[
\text{val mvc.} \text{mvc}_e\text{Tc} = \text{mu(mvc)} \quad (3.29)
\]

**Typing:** Finally, let us consider the typing of the example modules giving the typing rule MOD-OPEN of §3.3.6. The typing of module core goes through as described previously in §3.2.3, except that declarations for all the entities defined by (3.22) would be in the conclusion.

Applying rule MOD-OPEN to the definition of module mvc, as defined by (2.2), instantiates log variables m to “mvc” n to “core,” f to the body of mvc. The first two premises of the rule instantiate \( \bar{x}, \bar{J}, \bar{L} \) as defined by (3.23), (3.24) and (3.25). The list of current declarations, \( \bar{J}_c \), is extracted from \( \bar{J}_c \), yielding (3.26). Logic variables \( \Gamma_t \) and \( \Gamma_c \) are instantiated as follows:

\[
\Gamma_t = \text{/* declarations of the type fields of (3.27) */} \quad (3.30)
\]

\[
\Gamma_c = \text{func mvc.} \text{mvc}_e\text{Tc} = \text{mu(mvc)} \quad (3.31)
\]

The instantiation of \( \Gamma_c \) comes down to the following:

\[
\Gamma_{\text{mv}} = \text{val mvc:} \quad (3.32)
\]

\[
\text{val mvc:} \quad (3.33)
\]

Because Event was not defined by mvc, its future declaration is inherited and its current declaration states that it is equal to its value from core. Note that class mdef.Logger extends both msuper.Logger and mdef.Observer, both of which extend msuper.Observer.

The premise of MOD-OPEN ending \( \bar{J}_c \Rightarrow \bar{L}_c \), comes down to showing that the definitions in the current part of (2.9) satisfy the corresponding declarations in (3.26). Since most of the new implementation of mvc was elided, there is little to check. One requirement is that the self-type of each subclass is a subtype of its superclass’s self-type, which is trivial as they are identical. Another requirement is that the overridden implementation of the Logger class satisfies its declaration, which is established below.

Establishing the premise ending \( \bar{L}_c \cup \bar{L}_f \) is similar to that of §3.2.3 for core, except that there is no check for Event since it was not overridden.
The final premise comes down to ensuring, for each refinement, that its declaration in $L^{\text{mdef}}_{\text{mvc}}$ (see (3.26)) implies its declaration in $L^{\text{msuper}}_{\text{mdef}}$ (see (3.8)). For the Subject field, this comes down to

$$\Gamma; \Gamma_v \vdash (\text{type } \text{Subject} <: \text{msuper}\text{.Subject}) <: (\text{type } \text{Subject} <: \text{msuper}\text{.Logger})$$

This follows from the declaration of mdef.Subject which specifies that it extends msuper.Subject (see (3.30) and (3.26)). The other fields are similar.

Thus rule Mod-Open of §3.3.6 establishes that the definition of open module mvc, as given by (2.2), results in bindings satisfying $\Gamma^{\text{mdef}}_{\text{mvc}}$ and $\Gamma^{\text{msuper}}_{\text{mdef}}$ as defined by (3.28) and (3.29).

**Typing notify’s implementation:** As discussed above, the premise of Mod-Open ending $J_c := L_c$ requires that the overridden implementation of class Logger (in $J^{\text{mdef}}$ which is the current aspect of (2.9)) must satisfy its declaration (in $L^{\text{mdef}}_{\text{mvc}}$ as defined by (3.26)). This comes down to showing that the overridden implementation of method notify in mvc Logger’s class satisfies its inherited declaration.

Recall that notify was the problematic method in the original example. Its implementation was unsound with closed modules, as discussed in §1.3, but sound with open modules as discussed in §2.1.2. Let us delve deeper into the typing of notify’s implementation and justify the discussion of §2.1.2.

The implementation of mvc Logger’s notify method was first given in (1.2) and repeated in canonical form in (2.9):

```plaintext
func notify (subj: mself.Subject, ev: mself.Event) (3.31)
= log.printf("event from \%s: \%s (size=\%d)\n", subj.id, ev.msg, subj.modelSize());
```

Its declaration is inherited from msuper.Observer (through diamond inheritance from mdef.Logger, see (3.26), (3.30) and (3.8)):

```plaintext
```

To show that (3.31) yields (3.32), the following judgement must be established:

$$\Gamma; \Gamma_v \vdash \text{val self: mself.Logger; val subj: mself.Subject; val ev: mself.Event}$$

$$\vdash \text{self.log.printf("event from \%s: \%s (size=\%d)\n", subj.id, ev.msg, subj.modelSize()); Void}$$

Recall that “self” is the special variable for a class’s future object and that self. log is the explicit form of sibling reference. Type mself.Logger is a subtype of mdef.Logger which has log:Stream; let us assume that Stream has an appropriate printf method, so self.log.printf(fmt args) has a Void return if args have types matching fmt.

Variable subj has type mself.Subject, which is a subtype of mdef.Subject, which has appropriate id and modelSize fields (where id: String comes from msuper.Subject). Similarly, variable ev has type mself.Event, which is a subtype of mdef.Event, which has msg: String.

Thus it is established that the overridden notify method satisfies its declaration, which means that the Logger class satisfies its declaration, which means that the premise ending $J_c := L_c$ is established.

With respect to the discussion of §2.1.2, the declaration of mself given by (2.5) follows from the declarations of $\Gamma^{\text{msuper}}_{\text{mdef}}$ (given by (3.30)) by extracting the type declarations from (3.26) and (3.8) (and dropping the non-type entities), replacing the mdef and msuper references with their values and simplifying the with type expressions. Declaration (2.4) is the same as (3.32).

### 3.4 Observations

Let us end the main part of this paper with a few observations about the example: why it failed before and why it works with open modules. The problem with the implementation of section 1 was that the overridden implementation of notify in (1.2) was unsafe because the mvc classes extended the corresponding core ones. As illustrated by (1.4), because it was possible to mix objects from different modules, a core subject could call an mvc.Logger notify method, which would call back on the subject’s modelSize method, which did not exist. Type checking caught this unsoundness because the inherited type of parameter subj was core Subject, which did not have a modelSize method. An attempt to refine the method failed because the parameter is contravariant. Thus, to be safe, type checking disallows the enhanced functionality of the mvc extensions.

The open module implementation of section 2 allows use of the mvc enhancements by disallowing core and mvc objects to be mixed. In particular, the expression subject.addObserver(logger) in (1.4) fails to type because subject has type core.Subject, but logger has type mvc.Logger, which is not a subtype of the required type, core.Observer. The override of notify is well typed because the inherited type of parameter subj is mvc.Subject, which has the needed modelSize method. Let us look at this in more detail:

- There are two levels of inheritance in the mvc module: (1) the core entities inherit into mvc as mvc...super; and (2) the entities within each class of mvc...super inherit into the corresponding class of mvc.
- Note that each mvc class, mvc C, extends mvc...super C, not core C. Classes core C and mvc...super C are each instantiations of core...C (msuper C), with mself equal to core and mvc, respectively. Thus these two classes have the same implementations, but different types. Both classes are well-typed because core...C is typed with mself<<core and both core<<core and mvc<<core.
- Since class mvc C does not extend core C, type mvc C is not necessarily a subtype of core C. In fact, since the mvc classes have mutual references in contravariant positions, they are not subtypes of their core counterparts. This is why core and mvc objects may not mix.
- The unsafe situation of §1.3 can not be recreated by replacing “new core.Subject” in the second line of (1.4) with the functionally equivalent “new mvc...super.Subject” because the mvc...super classes have self-type specifications that are not satisfied and, so, can not be instantiated. Recall that the self-type specification on Subject is needed to allow the call of observer.notify(self.ev) to type check.
- The situation is actually more general than as described above. Within the body of mvc, special variable msuper refers to the entities inherited from core, while mdef refers to the entities defined by mvc. Variable msuper ranges over future modules, which must be such that mself<<mvc. Within object mvc, mself and msuper refer to mvc and msuper becomes mvc...super.
- For each class C, msuper C inherits into mdef C. The implementations of msuper are well-typed because mself<<mvc implies mself<<core.
- The overridden notify method is in class mdef.Logger, which extends msuper.Observer (through diamond inheritance). Both classes are mself-parameterized and, in particular, the inherited type of the subj parameter is mself.Subject, which has the needed modelSize method by mself<<mvc.
The key is that, in going from core to mvc, the type of parameter subj remains the same (mself.Subject), so no refinement is needed. Instead the type of mself has narrowed, which means that the core implementations remain well typed, while the mvc implementations get to use the new functionality.

4. Other issues and future work

For this final section, let us examine some of the shortcomings of the mechanisms of this paper and discuss how they might be handled.

4.1 Kinds

P3 uses kinds to track the structure of type-level values. Since hybrids can act as type groups, they too have kinds. To be complete, the encodings of this paper should have kind annotations on all type-level definitions (such as those of (3.18) and (3.19)), as well as on all path-starting hybrids including mself, mdef, msuper and m. Also the type rules should ensure that all type definitions are consistent with their kind annotations and that all type paths are well-kindred.

4.2 Other field descriptor modes

The modules of this paper were only allowed fields with descriptor modes virtual and refine. However, the current/future framework described in §2.2.3 is designed to handle other modes. In particular, an override would be equivalent to just a current aspect, with an inherited future aspect. A final field would be equivalent to a current aspect with a future aspect that says that the future value equals the current one (or has the same type as the current one).

It is also possible to permit abstract fields and partial modules. An abstract field would have just a future aspect, but no current one. An imp field would have just a current aspect and inherit the future aspect. A partial module would not yield a module object, but it could be extended. Only partial modules would be allowed to have unimplemented abstract fields.

A problem with handling the features above is that the current and future aspects become no longer parallel and, so, notations like \[ L_c \cap L_f \] no longer make sense. An appropriate notation needs to be developed.

4.3 Module polymorphism

Given that objects from different modules cannot mix, it would be nice to have some way of writing software that can operate over different modules. This notion of abstracting over collections of entities is often called “family polymorphism.” In P3, the “extends” type, \( \{ {?}<<m\} \), coupled with facilities from the underlying language, permits a form of family polymorphism.

For example, the following function takes a subject/observer module, som, and a subject from that module and sets up a logger specific to that subject:

```java
func addLogger (*som<<core, subj: som.Subject) { (4.1)
    val logStream = new PrintWriter("log." + subj.id)
    val logger = new som.Logger(logStream)
    subj.addObserver(logger)
}
```

Because this code is not correct as will be discussed in §4.4, let us assume declaration (4.3) and its implication (4.4). The "*" before parameter som indicates that som can be the start of a path-dependent type. In particular, the type of the second parameter, som.Subject, depends on som and, so, has as instances the Subject objects from the som module. Within the body of the function, som.Logger denotes the Logger class that is the sibling of som.Subject. Thus “addLogger(core,s)” takes a core.Subject instance and hooks it up to a core.Logger instance, while “addLogger(mvc,s)” takes an mvc.Subject and hooks it up to an mvc.Logger.

Given final fields as discussed in §4.2, the Subject and Observer classes of (2.1) could be modified to each have a field that refers to the containing module:

```java
final val outer = mself
```

In this case, (4.1) could be written with one parameter:

```java
func addLogger (subj: \{ {?}<<core\}.Subject) { (4.2)
    val logStream = new PrintWriter("log." + subj.id)
    val logger = new subj.outer.Logger(logStream)
    subj.addObserver(logger)
}
```

The P3 type "\{ {?}<<core\}.Subject” is a wildcard type whose instances are the union of instances of m.Subject for all m<<core. Thus this function works for any subject from any module extending core (including core). It gets the appropriate Logger class via subj.outer.Logger.

4.4 “Hazy” classes

In §2.2.3, the future declaration of a class was given as a type declaration. For example, the future declaration of the Logger class in core was:

```java
future type Logger <<: mdef.Logger \& mself.Observer
```

However, this specification is somewhat unintuitive. But more importantly, it leaves out key functionality. For example, the code of (4.1) needs the Logger class to be concrete and able to be instantiated by calling “new m.Logger(s)” with a stream s, for any module m<<core. For this purpose, P3 has an experimental kind of class qualifier called “hazy,” so called because not much is known about such classes as compared to regular classes. For the Logger example, consider the following future declaration:

```java
future hazy class Logger (Stream) extends mdef.Logger with mself.Observer
```

This would imply the subtype relation above, plus the following declaration of the new operator:

```java
future operator new Logger (Stream): mself.Logger
```

This would allow (4.1) to be type correct.

Currently a hazy class can not be extended except by another hazy class. Future research might allow this restriction to be lifted in controlled ways.

4.5 Field descriptor “fragile”

There are cases where a module’s field’s implementation is unsafe when inherited into some future extensions. For example, the Logger classes of (2.1) and (2.2) would be unsafe if a future Observer class had additional abstract methods which the current Logger classes did not implement. This situation manifests itself in the typing rules when the current declaration of a field does not imply its future declaration. In the case of Logger, this comes down to the following:


given mdef.Logger \leftrightarrow (mdef.Observer with { })

and mself.Observer <<: mdef.Observer

This fails because mdef.Observer<<mself.Observer can not be established from the assumptions.

This is an area of future research but tentatively P3 labels such fields with fragile (instead of virtual) and arranges to type them with mself<<m (or maybe mself<<mdef), instead of mself<<m.
Such fields must be overridden in an extension of the containing module.

4.6 Module declarations

The usual style in P3 is that each form of definition has a corresponding form of declaration. For instance, for each type, class, val or func definition in \( \Gamma^\text{core} \) (see the current part of (2.8)), there is a corresponding type, class, val or func declaration in \( \Gamma^\text{core} \) (see (3.8)). The “yields” operator, :=⇒, provides the logical relation between each pair. However, this paper does not present a form of declaration for module, but instead has :=⇒ go to a series of lower-level declarations. It is future research to define a module declaration construct, give its encoding and provide typing rules that yield module declarations. It is also future research to define field descriptors (see §2.2.2) for modules and their mapping to current/future form (which would enable nested modules).

References