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:

```
module core {                                              (1.1)
  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\n", subj.id, ev.msg)
  }
}
```

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.Event is a type and core.Subject, 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:

```
module mvc {                                               (1.2)
  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:

```
func notify (subj: core.Subject, ev: core.Event): Void    (1.3)
```

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:

```
val logger = new mvc.Logger(...);                    (1.4)
val subject = new core.Subject;
subject.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):

```
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):

```
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 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 "submodule." 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:

```
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 "mself" (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):

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

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:

```
val mself: {                                           (2.5)
    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 mself ranges over hybrid objects and is constrained by the hybrid type to the right of the colon. It is a dependent typing since mself appears in its own type. Variable 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 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 mvc.Logger (see (1.2)) is well typed because variable subj has type mself.Subject which is a subtype of a type that has appropriate id and modelSize fields. This will be shown more formally in §3.3.7.

When core is typed, mself would have the same type as in (2.5) except without the modelSize declaration nor the elided enhancements. Thus the type of mself in (2.5) is a subtype of the type of 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), mself.Subject and 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 mself$\leftrightarrow m$ (for module with name 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 m is "multi-functional" in the sense that it creates multiple entities:

- The module object itself, m.
- A type, $\{\{?{<\!\!<}m\}\}$, whose instances are extensions of the current module (including itself). This type is discussed in §2.2.4. It is the type of 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 **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 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 **virtual**, **abstract** or **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 "**current** J_c; **future** L_f" where J_c is a typed definition specifying the implementation and L_f is a declaration constraining future overrides.

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

$$\begin{array}{lcl} \textbf{virtual type } +X = T & \approx & \textbf{current type } X = T; \\ & & \textbf{future type } X <: \mathsf{mdef}.X \\ \textbf{virtual type } X\colon Z = T & \approx & \textbf{current type } X = T; \\ & & \textbf{future type } X\colon Z \\ \textbf{virtual type } X = T & \approx & \textbf{current type } X = T; \\ & & \textbf{future type } X \end{array} \tag{2.6}$$

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

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

$$\begin{array}{l} \textbf{virtual class } +C \text{ ...} \approx \\ \quad \textbf{current class } C \textbf{ selftype } \mathsf{mself}.C \text{ ...}; \\ \quad \textbf{future type } C <: \mathsf{mdef}.C \\ \textbf{virtual class } +C \textbf{ extends } +D \text{ ...} \approx \\ \quad \textbf{current class } C \textbf{ extends } \mathsf{mdef}.D \textbf{ selftype } \mathsf{mself}.C \text{ ...}; \\ \quad \textbf{future type } C <: \mathsf{mdef}.C \cap \mathsf{mself}.D \end{array} \tag{2.7}$$

Recall that class C defines a type C, so "**future type** $C{<:}C'$" requires that any future module has a class C whose type is a subtype of the type of C', i.e. C extends C'. Note that 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, C, that is in "diamond inheritance" with its sibling class D: the current class extends the current D and its future class must extend both the current C and the future D. Both current classes have a **selftype** specification which narrows the type of self. Special variables mself and mdef are described in §2.2.5.

2.2.4 The "extends" operator

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

One of the entities created by a definition of open module m is a type with the special P3 syntax "$\{\{?{<\!\!<}m\}\}$" whose instances are those modules in the extends relation with m. Thus p:$\{\{?{<\!\!<}m\}\}$ is equivalent to $p{<\!\!<}m$.

For example, definition (2.1) creates the type {{?<<core}} which is such that core<<core and mvc<<core. Module definition (2.2) creates the type {{?<<mvc}} such that mvc<<mvc (but *not* core<<mvc). Note that type {{?<<mvc}} is a subtype of {{?<<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.

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 m may not reference "m" directly.

Variable mself ranges over *future modules* which, for module m, are those modules whose fields satisfy m's future declarations, so mself<<m. Because the implementations of module m are typed with respect to mself<<m, 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 m created by a open module definition, mself refers to m. For example, in the core object created by (2.1), mself refers to core and notify has the declaration given by (1.3). Similarly, within the mvc object created by (2.2), mself refers to mvc and notify 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.

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 mself-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.

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 mself-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 **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.Observer, not mself.Observer, 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.Observer.

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):

```
open module core {                                                    (2.8)
  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 ∩ mself.Observer
  }
}
```

Notice that free sibling references have been explicitly qualified to mself (see §2.2.5) and that field descriptors have been split into **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 mself.Observer (the future Observer).

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

```
open module mvc extends core {                                        (2.9)
  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 (size=%d)\n",
          subj.id, ev.msg, subj.modelSize())
    }
  }
  future {
    type Subject <: mdef.Subject;
    type Observer <: mdef.Observer;
    type Logger <: mdef.Logger ∩ mself.Observer
  }
}
```

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 expresses 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:

$$\begin{array}{l} [\textsf{closed}]\ \textsf{module}\ m\ \{\ \bar{J}\ \} \longmapsto \\ \quad \textit{let} \\ \qquad \bar{x} = \textit{namesDefinedBy}(\ \bar{J}\ ) \\ \qquad \bar{J}' = \bar{J}\langle\bar{x}\mapsto m.\bar{x}\rangle \\ \quad \textit{in} \\ \qquad \textsf{val}\ m = \mu(m)\ \{\ \bar{J}'\ \} \end{array} \tag{3.1}$$

The encoding arrow, $\longmapsto$, indicates that definitions matching the pattern to its left can be encoded by instantiating the pattern to its right. Pattern variable 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 **closed** is optional. An over-bar indicates a list and, in this case, $\bar{J}$, $\bar{J}'$ and $\bar{x}$ are all parallel. The *let* clause binds a couple of pattern variables for use in the *in* part. The meta-function *namesDefinedBy* extracts the names defined by the list of definitions given as its argument. Operator μ 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 J^i of $\bar{J}$ can be any form of definition, including those starting with **val**, **func**, **type** or **class**.

In general, the phrase "$e\langle\bar{v}\mapsto\bar{d}\rangle$" denotes a substitution that yields expression e with all occurrences of free variables $\bar{v}$ replaced by corresponding expressions from $\bar{d}$. In this case, "$J^i\langle\bar{x}\mapsto m.\bar{x}\rangle$" denotes the substitution such that each free variable, $x^j\in\bar{x}$, in definition J^i is replaced by the corresponding expression, $m.x^j$. 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.

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:

MOD-CLOSED:

$$\frac{\bar{x} \overset{\triangle}{=} \textit{namesDefinedBy}(\ \bar{J}\ ) \qquad \Gamma;\ \textsf{val}\ m\colon \{\ \bar{L}\ \} \vdash \bar{J}\langle\bar{x}\mapsto m.\bar{x}\rangle :\Longrightarrow \bar{L}}{\Gamma \vdash [\textsf{closed}]\ \textsf{module}\ m\ \{\ \bar{J}\ \} :\Longrightarrow \textsf{val}\ m\colon \tau[m]\ \{\ \bar{L}\ \}}$$

Logic variable m ranges over identifiers, $\bar{J}$ over lists of typed definitions, $\bar{L}$ over lists of declarations (parallel to $\bar{J}$) and $\bar{x}$ over identifier lists (also parallel to $\bar{J}$). The term "$\{\bar{L}\}$" is a record type. The phrase "$\Gamma\vdash J:\Longrightarrow L$" is a P3 judgement that states that definition J, when executed in a context satisfying environment Γ, results in a binding satisfying declaration L. The operator τ is P3's mechanism for dependent types and is such that $p{:}\tau[x]T$ implies $p{:}T\langle x\mapsto p\rangle$ (for path 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 $:\Longrightarrow$ in the conclusion. For the purposes of this paper, assume that it can be determined from annotations appearing in $\bar{J}$ or by inference (not involving the recursion variable m).

Explanation: The rule says that, if the two premises are true, then after the execution of "**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, J^i, in the body of the module yields a binding satisfying the corresponding declaration L^i.

Justification: This rule can be justified by replacing the module definition to the left of $:\Longrightarrow$ 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:

$$\begin{array}{l} \textsf{open module}\ m\ \{\ \bar{\mathcal{J}}\ \} \longmapsto \\ \quad \textit{let} \\ \qquad \bar{x} = \textit{namesDefinedBy}(\ \bar{\mathcal{J}}\ ) \\ \qquad \bar{\mathcal{J}}\langle\bar{x}\mapsto \textsf{mself}.\bar{x}\rangle \approx \textsf{current}\ \bar{J}_c;\ \textsf{future}\ \bar{L}_f \\ \quad \textit{in} \\ \qquad \textsf{type oper}\ \{\!\{?{<\!\!<}m\}\!\} = \tau[\textsf{mself}]\ \{\ \bar{L}_f\ \} \\ \qquad \textsf{func}\ m\_\textsf{c}\ (\textsf{mself}{<\!\!<}m) = \mu(\textsf{mdef})\ \{\ \bar{J}_c\ \} \\ \qquad \textsf{val}\ m = \mu(m)\ m\_\textsf{c}(m) \end{array} \tag{3.2}$$

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

This encoding requires that mdef does not appear free in $\bar{L}_f$. 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.

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

The second entity is a "module constructor" function, $m\_\textsf{c}$, that abstracts over mself and yields a record formed from the current definitions, $\bar{J}_c$. It is recursive in mdef to tie the sibling implementations together as specified. It is parameterized by mself$<\!\!<m$ to allow the implementations to be inherited.

This module constructor, $m\_\textsf{c}$, will be needed for extension: a module, f, extending this one (so $f{<\!\!<}m$) will be constructed by calling $m\_\mathsf{c}(f)$ (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 object, 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:

MOD-OPEN-TL:

$$\frac{\begin{array}{c} \bar{x} \triangleq \textit{namesDefinedBy}(\ \bar{\mathcal{J}}\ ) \\ \bar{\mathcal{J}}\langle \bar{x} \mapsto \mathsf{mself}.\bar{x} \rangle \approx \textbf{current}\ \bar{J}_c;\ \textbf{future}\ \bar{L}_f \\ \Gamma;\ \textbf{val}\ \mathsf{mself}\text{: } \{\bar{L}_f\};\ \textbf{val}\ \mathsf{mdef}\text{: } \{\bar{L}_c\} \vdash \bar{J}_c :\Longrightarrow \bar{L}_c \\ \Gamma;\ \textbf{val}\ \mathsf{mself}\text{: } \{\bar{L}_f\};\ \textbf{val}\ \mathsf{mdef}\text{: } \{\bar{L}_c\} \vdash \bar{L}_c \lhd\!: \bar{L}_f \\ \textit{/* define } \Gamma_c \textit{ as given in (3.3) */} \end{array}}{\Gamma \vdash \textbf{open module}\ m\ \{\ \bar{\mathcal{J}}\ \} :\Longrightarrow \Gamma_c}$$

The pattern variables of (3.2) are logic variables of this rule, with the addition of $\bar{L}_c$ which ranges over declaration lists. The rewrite to current/future form is as given in §2.2.3 and instantiates logic variables $\bar{J}_c$ and $\bar{L}_f$ from $\bar{\mathcal{J}}$. It must be that $\bar{L}_c$ can be determined from annotations appearing in $\bar{J}_c$ or by inference (not involving the recursion variables mself nor mdef). Lists $\bar{J}_c$, $\bar{L}_c$ and $\bar{L}_f$ are all parallel, which is enabled by the condition that all fields are virtual (and so have both implementation and future type). A judgement "$L_1 \lhd\!: L_2$" indicates that declaration L_1 implies L_2; it is like a subtype relation, except on declarations. The logic variable Γ_c is defined as follows:

$$\begin{array}{l} \Gamma_c \triangleq \\ \quad \textbf{type oper}\ \{\{?{<}{<}m\}\} \leftrightarrow \tau[\mathsf{mself}]\ \{\bar{L}_f\} \\ \quad \textbf{func}\ m\_\mathsf{c}\ ({*}\mathsf{mself}{<}{<}m) : \tau[\mathsf{mdef}]\ \{\bar{L}_c\} \\ \quad \textbf{val}\ m : \tau[m]\ \{\ \bar{L}_c\langle \mathsf{mself},\mathsf{mdef} \mapsto m,m\rangle\ \} \end{array} \tag{3.3}$$

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 may appear in $\bar{L}_c$. The third gives the type of the module object.

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

- Instantiate m to the name of the module and $\bar{\mathcal{J}}$ to its body. Extract the names of entities in $\bar{\mathcal{J}}$ and qualify free references to them to mself. Split the resulting body into current/future form, instantiating $\bar{J}_c$ and $\bar{L}_f$.
- Determine $\bar{L}_c$ from the annotations on $\bar{J}_c$. It is a type error if the annotations are insufficient.
- Show that the implementations are type correct: show that definition J_c^i (of $\bar{J}_c$) satisfies its corresponding declaration, L_c^i. Show this is an environment where mself is constrained by the future declarations and mdef by the current declarations (which may have mself free). The constraint mself:$\{\bar{L}_f\}$ is equivalent to mself<<m and ensures that implementations will continue to work in all descendant modules of m.
- Show that each current declaration L_c^i, of $\bar{L}_c$, implies the corresponding future declaration, L_f^i. Show this in the same environment discussed above.
- Conclude that the module definition yields entities satisfying the declarations given by Γ_c as defined by (3.3).

Justification: As in §3.1.2, this rule is justified by replacing the module definition in the conclusion with its encoding and showing that the premises lead to the (new) conclusion. In this case, the encoding is given by (3.2). The *let*-clause of the encoding corresponds to the first two premises of the rule and the *in*-clause yields three definitions corresponding to Γ_c as defined by (3.3). The first definition trivially implies its corresponding declaration (except for kind checks which are not discussed here). The third premise, ending with $\bar{J}_c{:}{\Longrightarrow}\bar{L}_c$, implies that the definition of m_c in the encoding has the declaration given by Γ_c. The fourth premise, ending with $\bar{L}_c \lhd\!: \bar{L}_f$, ensures that $m{<}{<}m$, which is needed for $\mu(m)m\_\mathsf{c}(m)$ to be well-typed. The type of this last expression is $\tau[\mathsf{mself}]\tau[\mathsf{mdef}]\{\bar{L}_c\}$, which is equivalent to $\tau[m]\{\bar{L}_c\langle \mathsf{mself},\mathsf{mdef} \mapsto m,m\rangle\}$ as appearing in the final declaration of Γ_c.

Terminology: "exact" and "general" types: Note that module m has two types: the one to the right of the colon in the last line of (3.3) and the one defined on the first line. P3 refers to these as the "exact" and "general" types of m, respectively. The exact type is based on the current declarations and applies only to this module, not its descendants. The general type is based on the future declarations and applies to this module and its descendants. The exact type is related to Bruce's exact types (those with @) [1, 3], while the general type is related to Bruce's hash types [2].

3.2.3 Encoding and typing module core

Let us consider the encoding and typing of the core module definition using the rules of this section. Technically neither applies since the future declarations of core reference mdef, but let us proceed anyway and point out where things fail.

Encoding: First consider applying encoding (3.2) to the core module definition (2.1). In this case, pattern variable m is instantiated to "core," $\bar{\mathcal{J}}$ to the body of core and $\bar{x}$ to the names defined by core as follows:

$$\bar{x}^{\mathsf{core}} \triangleq \mathsf{Event,Subject,Observer,Logger} \tag{3.4}$$

The body of core is rewritten to qualify all free sibling names with respect to mself and the equivalences of §2.2.3 are used to get it into current/future form, resulting in (2.8). Logic variables $\bar{J}_c$ and $\bar{L}_f$ are instantiated to the current and future aspects of core:

$$\bar{J}_c^{\mathsf{core}} \triangleq \textit{/* the part of (2.8) within } \textbf{current}\{\}\textit{ */} \tag{3.5}$$

$$\bar{L}_f^{\mathsf{core}} \triangleq \textit{/* the part of (2.8) within } \textbf{future}\{\}\textit{ */} \tag{3.6}$$

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

$$\begin{array}{l} \textbf{type oper}\ \{\{?{<}{<}\mathsf{core}\}\} = \tau[\mathsf{mself}]\ \{\ \bar{L}_f^{\mathsf{core}}\ \} \\ \textbf{func}\ \mathsf{core\_c}\ (\mathsf{mself}{<}{<}\mathsf{core}) = \mu(\mathsf{mdef})\ \{\ \bar{J}_c^{\mathsf{core}}\ \} \\ \textbf{val}\ \mathsf{core} = \mu(\mathsf{core})\ \mathsf{core\_c}(\mathsf{core}) \end{array} \tag{3.7}$$

The first defines {{?<<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 $\bar{L}_f^{\mathsf{core}}$ 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 $\bar{\mathcal{J}}$ to the body of core. The first two premises of the rule instantiate $\bar{x}$, $\bar{J}_c$ and $\bar{L}_f$ as defined by (3.4), (3.5) and (3.6). The list of current declarations, $\bar{L}_c$, is extracted from $\bar{J}_c$:

$$\begin{array}{l} \bar{L}_c^{\mathsf{core}} \triangleq \\ \quad \textbf{type}\ \mathsf{Event} \leftrightarrow \{\ \textbf{val}\ \mathsf{msg}\text{: String }\}; \\ \quad \textbf{class}\ \mathsf{Subject\ (String)}\ \textbf{selftype}\ \mathsf{mself.Subject}\ \{ \\ \quad\quad \textbf{val}\ \mathsf{id}\text{: String}; \end{array} \tag{3.8}$$

```
    val observers: Set[mself.Observer];
    func addObserver (mself.Observer): 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 Γ_v^{core} to capture the declarations of mself and mdef:

$$\begin{aligned}\Gamma_v^{\mathsf{core}} \triangleq\ & \textbf{val}\ \mathsf{mself}\text{: } \{\ \bar{L}_f^{\mathsf{core}}\ \}; \\ & \textbf{val}\ \mathsf{mdef}\text{: } \{\ \bar{L}_c^{\mathsf{core}}\ \}\end{aligned} \tag{3.9}$$

Note that these are mutually dependent declarations as both mself and mdef may appear in both $\bar{L}_f^{\text{core}}$ and $\bar{L}_c^{\text{core}}$.

The third premise, which ends $\bar{J}_c{:}\!\Longrightarrow\!\bar{L}_c$, ensures that field implementations (those in the **current** part of (2.8)) satisfy their declarations (as in (3.8)). 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:

  ```
  Γ; Γ_v^core; val self: mself.Subject ⊢
      func notifyObservers (ev: mself.Event) =
        for (observer <- self.observers)
          { observer.notify(self, ev) }
    :⟹ func notifyObservers (mself.Event): 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:$\{\bar{L}_f^{\mathsf{core}}\}$ implies mself.Subject<:mdef.Subject. Thus self.observers has type Set[mself.Observer] (by mdef:$\{\bar{L}_c^{\mathsf{core}}\}$), 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 because mself.Subject and mself.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 $\bar{L}_c^{\mathsf{core}}{\lhd}{:}\bar{L}_f^{\mathsf{core}}$, 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:

$$\begin{aligned}&(\textbf{type}\ \mathsf{Event} \leftrightarrow \{\ \textbf{val}\ \mathsf{msg}\text{: String}\ \})\ \lhd{:} \\ &\quad(\textbf{type}\ \mathsf{Event} <: \mathsf{mdef.Event})\end{aligned} \tag{3.10}$$

The first line is also the declaration of mdef.Event, so (3.10) comes down to showing that "{**val** 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 Γ_c^{core}:

$$\begin{aligned}&\Gamma_c^{\mathsf{core}} \triangleq \\ &\quad \textbf{type oper}\ \{\{?{<<}\mathsf{core}\}\} \leftrightarrow \tau[\mathsf{mself}]\ \{\ \bar{L}_f^{\mathsf{core}}\ \} \\ &\quad \textbf{func}\ \mathsf{core\_c}\ (\mathsf{mself}{<<}\mathsf{core}) : \tau[\mathsf{mdef}]\{\ \bar{L}_c^{\mathsf{core}}\ \} \\ &\quad \textbf{val}\ \mathsf{core} : \tau[\mathsf{core}]\ \{\ \bar{L}_c^{\mathsf{core}}\langle \mathsf{mself},\mathsf{mdef}{\mapsto}\mathsf{core},\mathsf{core}\rangle\ \}\end{aligned} \tag{3.11}$$

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 Γ_c^{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_c, whose general form is as follows:

$$\begin{aligned}&\textbf{func}\ m\_\mathsf{c}\ (\mathsf{mself}{<<}m) = \mu(\mathsf{mdef}) \\ &\quad \textbf{let val}\ \mathsf{msuper} = n\_\mathsf{c}(\mathsf{mself}) \\ &\quad \textbf{in}\ \mathsf{msuper}\ \textbf{with}\ \{\ \bar{J}_c\ \}\end{aligned} \tag{3.12}$$

The parent module, n, must have been encoded similarly and, so, has constructor function n_c. The operator **with** applied to two record values unions the fields of the values with those of the right value overriding those of the left. Similarly **with** applied to two record types unions the declarations with those of the right type overriding. $\bar{J}_c$ 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_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_c, is called to create n's part of the module, which is bound to the special variable msuper, which is used both to provide inheritance for any field not overridden and to satisfy references to it within the implementations, $\bar{J}_c$.

Constructor function m_c is used in two ways. First, it is used to construct module m:

$$\textbf{val}\ m = \mu(m)\ m\_\mathsf{c}(m) \tag{3.13}$$

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

Second, the constructor function m_c is used to provide inheritance to any child module. In particular, it is used in any child module's constructor, just as n_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 $m_{i>0}$ extends m_{i-1}. Then the following definition is equivalent to (3.13) and k versions of (3.12), where the nested **let**-expressions have been flattened into one:

```
val $m_k = \mu(m_k)$ let*                                        (3.14)
    val $s_0 = \mu(s_0)$ { $\bar{J}_c^0\langle$mself,mdef$\mapsto m_k, s_0\rangle$ }
    val $s_1 = \mu(s_i)$ $s_0$ with { $\bar{J}_c^1\langle mvars \mapsto m_k, s_1, s_0\rangle$ }
    ...
    val $s_k = \mu(s_k)$ $s_{i-1}$ with { $\bar{J}_c^k\langle mvars \mapsto m_k, s_k, s_{k-1}\rangle$ }
  in $s_k$
```

To save space, *mvars* stands for "mself,mdef,msuper." Each $\bar{J}_c^i$ denotes the list of module m_i's current definitions.

The **let***-clause defines a series of k "slice" objects, s_0 to s_k, and the **in**-clause yields the last one, s_k. Slice s_0 is as if created by m_0_c(m_k) and slices s_0 to s_i are as if created by m_i_c(m_k). 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 m's exact type.

Note that the exact type of a module can depend on its entire series of slices, because those slices may contain type definitions (either directly or via class definitions) and slices both inherit and can be referred to by msuper. For example, the type of mvc.Subject depends on the Subject class definitions from *both* core and mvc.

In general, the exact type of module m_k of (3.14) can be given by the following declaration:

```
val $m_k$: $\tau[m_k]$                                              (3.15)
    $\cup$[ $s_0$: {$\bar{L}_c^0\langle$mself,mdef$\mapsto m_k, s_0\rangle$} ]
    $\cup$[ $s_1$: {$\bar{x}_0 \leftrightarrow s_0.\bar{x}_0$} with {$\bar{L}_c^1\langle mvars \mapsto m_k, s_1, s_0\rangle$} ]
    ...
    $\cup$[ $s_k$: {$\bar{x}_{k-1} \leftrightarrow s_{k-1}.\bar{x}_{k-1}$}
          with {$\bar{L}_c^k\langle mvars \mapsto m_k, s_k, s_{k-1}\rangle$} ]
  {{?$\leftrightarrow s_k$}}
```

Again *mvars*="mself,mdef,msuper." The "$\cup$" 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 their own type and the types of later slices. Type {{?$\leftrightarrow s_k$}} is the type of things identical to s_k, which, in this case, means that module m_k *is* slice s_k.

Declarations $\bar{L}_c^i$ are those capturing the result of the level-i current definitions, $\bar{J}_c^i$. The part before **with** is a type expressing that all fields inherit from the previous slice (which are overridden by corresponding fields after **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 structure

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 *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:

```
type oper {{?<<$m_k$}} $\leftrightarrow$ $\tau$[mself]                               (3.16)
    $\cup$[ $s_0$: {$\bar{L}_c^0\langle$mdef$\mapsto s_0\rangle$} ]
    $\cup$[ $s_1$: {$\bar{x}_0 \leftrightarrow s_0.\bar{x}_0$} with {$\bar{L}_c^1\langle$mdef,msuper$\mapsto s_1, s_0\rangle$} ]
    ...
    $\cup$[ $s_k$: {$\bar{x}_{k-1} \leftrightarrow s_{k-1}.\bar{x}_{k-1}$}
          with {$\bar{L}_c^k\langle$mdef,msuper$\mapsto s_k, s_{k-1}\rangle$} ]
  ( {$\bar{L}_f^0\langle$mdef$\mapsto s_0\rangle$} with ... with {$\bar{L}_f^k\langle$mdef$\mapsto s_k\rangle$} )
```

Basically this says that for a module, f, to extend m_k, (1) it must have at least k slices, s_0 to s_k, satisfying the corresponding part of (3.15) except with mself bound to f; and (2) each of f's members must satisfy $\bar{L}_f^k\langle$mself,mdef$\mapsto f, s_k\rangle$ or the latest $\bar{L}_f^i\langle$mself,mdef$\mapsto f, s_i\rangle$ if the future type is inherited.

3.3.5 Encoding open module definitions in general

The encoding of this section defines 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:

```
open module $m$ extends $n$ { $\bar{\mathcal{J}}$ } $\longmapsto$                        (3.17)
  let
    $\bar{x}$ = namesDefinedBy( $\bar{J}$ )
    $\bar{\mathcal{J}}\langle \bar{x} \mapsto$mself.$\bar{x}\rangle \approx$ current $\bar{J}_c$; future $\bar{L}_f$
    $\bar{L}_c$ = declarationsOf($\bar{J}_c$)
  in
    /* type definitions (3.18) and (3.19) */
    func $m$_c (mself<<$m$): $m$_eTc[mself] = $\mu$(mdef)
      let val msuper: $n$_eTc[mself] = $n$_c(mself)
      in msuper with { val _super = msuper; $\bar{J}_c$ }
    val $m$: $m$_eT = $\mu(m)$ $m$_c$(m)$
```

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 two types, to be given by definitions (3.18) and (3.19) below. The remainder is the same as (3.12) and (3.13), except type annotations are added and a field _super is defined.

This additional field, "_super," is defined to refer to the slice of its parent and so _super._super refers to its grandparent and so on. This use of _super will avoid the need for some of the unions appearing in (3.15) and (3.16).

The first part of the encoding contains types related to the exact type of the module:

```
type $m$_eTccc [mself, mdef, msuper] =                             (3.18)
   { val $\bar{x}^n$ $\leftrightarrow$ msuper.$\bar{x}^n$ } with { $\bar{L}_c$ }
type $m$_eTcc [mself, mdef] =
   { val _super: $n$_eTcc[mself,mdef._super] }
   with $m$_eTccc[mself,mdef,mdef._super]
type $m$_eTc [mself] = $\tau$[mdef] $m$_eTcc[mself,mdef]
type $m$_eT = $\tau$[mself] $m$_eTc[mself]
```

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 "**type** $X[\bar{Y}]$=T" creates a binding from variable X to the type function $\lambda[\bar{Y}]T$. 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[mself]" in (3.17)), only its type-group projection matters.

The types of (3.17) are such that m_i_eTccc[m_k,s_i,s_{i-1}] is the type of slice s_i in (3.15) and m_i_eTcc[m_k,s_i] and m_i_eTc[m_k] are both types of slice s_i with the additional _super field (which gives access to s_{i-1} to s_0). Type m_k_eT is the exact type of module object m_k. Note that, because $s_k \leftrightarrow m_k$ and $s_{i-1} \leftrightarrow s_i$._super, there is no need for the explicit unions of (3.15).

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

$$\begin{array}{l} \textbf{type}\ m\_\mathsf{gTcc[mself,\ mdef]} = \\ \quad n\_\mathsf{gTcc[mself,mdef.\_super]}\ \textbf{with}\ \{\ \bar{L}_f\ \} \\ \textbf{type oper}\ \{\{?{<}{<}m\}\} = \tau\mathsf{[mself]} \\ \quad \cup\mathsf{[mdef:}\ m\_\mathsf{eTcc[mself,mdef]]}\ m\_\mathsf{gTcc[mself,mdef]} \end{array} \tag{3.19}$$

The g is for "general" (see §3.3.4). These two definitions, together with m_eTcc from (3.18), represent (3.16) where s_k is referred to by mdef and s_{i-1} by s_i._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_c over mself and a module object m, just as discussed in §3.3.1. The constructor function m_c is crafted so that m's implementations, $\bar{J}_c$, 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_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_c, m 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_eTccc and m_eTcc and m_gTcc are as follows:

$$\begin{array}{l} \textbf{type}\ m\_\mathsf{eTcc\ [mself,\ mdef]} = \{\ \bar{L}_c\ \} \\ \textbf{type}\ m\_\mathsf{gTcc\ [mself,\ mdef]} = \{\ \bar{L}_f\ \} \end{array}$$

Also the constructor has no let clause nor use of _super:

$$\textbf{func}\ m\_\mathsf{c}\ (\mathsf{mself}{<}{<}m) : m\_\mathsf{eTc\ [mself]} = \mu(\mathsf{mdef})\ \{\ \bar{J}_c\ \}$$

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 formalism than does typing closed or top-level open modules. It can be based on encoding (3.17):

MOD-OPEN:

$$\frac{\begin{array}{c} \bar{x} \triangleq \textit{namesDefinedBy}(\ \bar{\mathcal{J}}\ ) \\ \bar{\mathcal{J}}\langle \bar{x} \mapsto \mathsf{mself}.\bar{x} \rangle \approx \textbf{current}\ \bar{J}_c;\ \textbf{future}\ \bar{L}_f \\ \Gamma_t \triangleq \textit{/* declarations of (3.18) and (3.19) */} \\ \textit{/* definitions of } \Gamma_v \textit{ from (3.20) and } \Gamma_c \textit{ from (3.21) */} \\ \Gamma;\ \Gamma_t;\ \Gamma_v \vdash \bar{J}_c :\Longrightarrow \bar{L}_c \qquad \Gamma;\ \Gamma_t;\ \Gamma_v \vdash \bar{L}_c \lhd:\ \bar{L}_f \\ \Gamma;\ \Gamma_t;\ \Gamma_v \vdash \mathsf{mself}:\ n\_\mathsf{gTcc[mself,msuper]} \end{array}}{\Gamma \vdash \textbf{open module}\ m\ \textbf{extends}\ n\ \{\ \bar{\mathcal{J}}\ \} :\Longrightarrow \Gamma_t;\ \Gamma_c}$$

The pattern variables of (3.17) are logic variables of this rule, with the addition of $\bar{L}_c$, Γ_t, Γ_v and Γ_c. $\bar{L}_c$ ranges over declaration lists and is determined from annotations appearing in $\bar{J}_c$ or by inference (not involving mself nor mdef). Logic variable Γ_t is instantiated to the list of declarations formed by substituting "$\leftrightarrow$" for "=" in (3.18) and (3.19). Lists $\bar{J}_c$, $\bar{L}_c$ 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 Γ_v is defined to have the declarations for mself, mdef and msuper used in the main typing clauses:

$$\begin{array}{l} \Gamma_v \triangleq \\ \quad \textbf{val}\ \mathsf{mself}:\ m\_\mathsf{gTcc[mself,mdef]}; \\ \quad \textbf{val}\ \mathsf{mdef}:\ m\_\mathsf{eTccc[mself,mdef,msuper]}; \\ \quad \textbf{val}\ \mathsf{msuper}:\ n\_\mathsf{eTcc[mself,msuper]} \end{array} \tag{3.20}$$

Note that these declarations are mutually dependent.

Logic variable Γ_c is defined to have the declarations for the module constructor function, m_c, and the module itself, m:

$$\begin{array}{l} \Gamma_c \triangleq \\ \quad \textbf{func}\ m\_\mathsf{c}\ ({*}\mathsf{mself}{<}{<}m):\ m\_\mathsf{eTc[mself]}; \\ \quad \textbf{val}\ m:\ m\_\mathsf{eT} \end{array} \tag{3.21}$$

Explanation: This rule says that to check that a 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 $\bar{\mathcal{J}}$ to its body. Also extract the names of entities in $\bar{\mathcal{J}}$, qualify sibling references to mself and split the resulting body into current/future form, instantiating $\bar{J}_c$ and $\bar{L}_f$. Determine $\bar{L}_c$ from the annotations on $\bar{J}_c$, as before.
- Show that the implementations are well typed: that $\bar{J}_c$ satisfy their corresponding declarations, $\bar{L}_c$. Show this is an environment containing Γ_t (which contains the declarations of (3.18) and (3.19)) and Γ_v (defined by (3.20)). Γ_v gives the types of mself, mdef and msuper using the types declared by Γ_t.
- Show that the current declarations imply the future ones: $\bar{L}_c \lhd: \bar{L}_f$. Show this in the same environment discussed above.
- Show mself:n_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 Γ_t (from (3.18) and (3.19)) and Γ_c as defined by (3.3).

Note that the implementations, $\bar{J}_c$, 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 $\bar{L}_c$. The conclusion of the rule asserts that the module definition results in Γ_t;Γ_c. The declarations of Γ_t 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 Γ_c follow directly from the annotations on m_c and m.

What remains is to show that the last three premises of MOD-OPEN imply that the implementations of m_c and m satisfy their annotations. Note that Γ_v (as defined by (3.20)) gives the types of mself, mdef and msuper in the body of m_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_eTc[mself] since n was encoded by this same encoding.

The premise ending with $\bar{J}_c{:}{\Longrightarrow}\bar{L}_c$ 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

m_eTcc[mself,mdef], which means that the recursion over mdef has type m_eTc[mself], which is the result type of m_c.

The premise ending $\bar{L}_c \lhd: \bar{L}_f$ maintains the following encoding invariant:

$\forall[\Gamma_v]$ mdef : m_gTcc[mself, mdef]

Its proof uses msuper:n_gTcc[mself,msuper], which follows from the invariant of the parent module, n, and from the definition of m_gTcc. This ensures m<<m, which is needed for $\mu(m)m$_c(m) to be well-typed.

3.3.7 Encoding and typing module mvc

Now let us consider encoding and typing the mvc module definition (2.2) using the encoding and type rules of this section. The restriction that future declarations may not reference mdef is lifted.

Revised encoding of* core*: Because encoding (3.17) requires that its parent module be encoded by it, let us first reconsider the encoding of core as defined by (2.1). As in §3.2.3, pattern variable m is instantiated to "core" and $\bar{\mathcal{J}}$ to the body of core. The instantiations of $\bar{x}$, $\bar{J}_c$, $\bar{L}_f$ and $\bar{L}_c$ are as defined by (3.4), (3.5), (3.6) and (3.8), respectively. Thus the following is the revised encoding of core:

```
type core_eTcc [mself, mdef] = { $L_c^{core}$ }                    (3.22)
type core_eTc [mself] = $\tau$[mdef] core_eTcc[mself,mdef]
type core_eT = $\tau$[mself] core_eTc[mself]
type core_gTcc [mself, mdef] = { $L_f^{core}$ }
type oper {{?<<core}} = $\tau$[mself]
  $\cup$[mdef: core_eTcc[mself,mdef]] core_gTcc[mself,mdef]
func core_c (*mself<<core): core_eTc[mself]
  = $\mu$(mdef) { $J_c^{core}$ }
val core: core_eT = $\mu$(core) core_c(core)
```

Recall that J_c^{core} is the part of (2.8) within **current{}**, L_f^{core} is the part of (2.8) within **future{}** and L_c^{core} are the declarations extracted from J_c^{core} and given by (3.8).

Note that the definitions of core_c and core are the same as in the previous encoding, (3.7), except with annotations. The definition of {{?<<core}} comes down to the following:

type oper {{?<<core}} = τ[mself] $\cup$[mdef: $\{L_c^{\text{core}}\}$] $\{L_f^{\text{core}}\}$

This would be equivalent to the definition given in (3.7) if L_f^{core} did not refer to mdef, but this definition properly handles such references (at the cost of having a union).

Encoding of* mvc*: Now consider applying encoding (3.17) to the mvc module definition (2.2). In this case, pattern variable m is instantiated to "mvc," n to "core," $\bar{\mathcal{J}}$ to the body of mvc and $\bar{x}$ to the names defined by mvc as follows:

$$\bar{x}^{\text{mvc}} \triangleq \text{Event,Subject,Observer,Logger} \tag{3.23}$$

The body of mvc is rewritten to qualify $\bar{x}^{\text{mvc}}$ to mself.$\bar{x}^{\text{mvc}}$ and rewritten into current/future form, resulting in (2.9). Thus logic variables $\bar{J}_c^{\text{mvc}}$ and $\bar{L}_f^{\text{mvc}}$ are instantiated to the current and future aspects of mvc, respectively:

$$\bar{J}_c^{\text{mvc}} \triangleq \text{/* \textit{the part of (2.9) within} \textbf{current}\{\} */} \tag{3.24}$$
$$\bar{L}_f^{\text{mvc}} \triangleq \text{/* \textit{the part of (2.9) within} \textbf{future}\{\} */} \tag{3.25}$$

Pattern variable $\bar{L}_c^{\text{mvc}}$ is defined to be *declarationsOf($\bar{J}_c^{mvc}$)*:

```
$\bar{L}_c^{mvc}$ ≜                                                      (3.26)
  class Subject (String) extends msuper.Subject
    selftype mself.Subject { func modelSize(): Int; ... }
  partial class Observer extends msuper.Observer
    selftype mself.Observer { ... }
  class Logger (Stream) extends msuper.Logger
    with mdef.Observer selftype mself.Logger {}
```

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

```
type mvc_eTccc [mself, mdef, msuper] =                            (3.27)
  { val $\bar{x}^{core}$ ↔ msuper.$\bar{x}^{core}$ } with { $\bar{L}_c^{mvc}$ }
type mvc_eTcc [mself, mdef] =
  { val _super: core_eTcc[mself,mdef._super] }
  with mvc_eTccc[mself,mdef,mdef._super]
type mvc_eTc [mself] = $\tau$[mdef] mvc_eTcc[mself,mdef]
type mvc_eT = $\tau$[mself] mvc_eTc[mself]
type mvc_gTcc [mself, mdef] =
  core_gTcc[mself,mdef._super] with { $\bar{L}_f^{mvc}$ }
type oper {{?<<mvc}} = $\tau$[mself]
  $\cup$[mdef: mvc_eTcc[mself,mdef]] mvc_gTcc[mself,mdef]
func mvc_c (mself<<mvc): mvc_eTc[mself] = $\mu$(mdef)
  let val msuper: core_eTc[mself] = core_c(mself)
  in msuper with { val _super = msuper; $\bar{J}_c^{mvc}$ }
val mvc: mvc_eT = $\mu$(mvc) mvc_c(mvc)
```

Typing: Finally, let us consider the typing of the example modules given 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 logic variable m to "mvc," n to "core" and $\bar{\mathcal{J}}$ to the body of mvc. The first two premises of the rule instantiate $\bar{x}$, $\bar{J}_c$ and $\bar{L}_f$ as defined by (3.23), (3.24) and (3.25). The list of current declarations, $\bar{L}_c$, is extracted from $\bar{J}_c$, yielding (3.26). Logic variables Γ_t and Γ_c are instantiated as follows:

```
$\Gamma_t^{mvc}$ ≜ /* declarations of the type fields of (3.27) */       (3.28)
$\Gamma_c^{mvc}$ ≜                                                       (3.29)
  func mvc_c (*mself<<mvc): mvc_eTc[mself]
  val mvc: mvc_eT
```

The instantiation of Γ_v comes down to the following:

```
$\Gamma_v^{mvc}$ ≜                                                       (3.30)
  val mself: {
    type Event <: msuper.Event;
    type Subject <: mdef.Subject;
    type Observer <: mdef.Observer;
    type Logger <: mdef.Logger $\cap$ mself.Observer
  }
  val mdef: {
    type Event ↔ msuper.Event;
    $\bar{L}_c^{mvc}$ /* see (3.26) */
  }
  val msuper: { $L_c^{core}$⟨mdef↦msuper⟩ } /* see (3.8) */
```

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 $\bar{J}_c :\Longrightarrow \bar{L}_c$, 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 $\bar{L}_c \lhd: \bar{L}_f$ 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_f^{mvc} (see (3.26)) implies its declaration in $L_f^{\mathrm{core}}\langle\mathrm{mdef}\mapsto\mathrm{msuper}\rangle$ (see (3.8)). For the Subject field, this comes down to

$$\Gamma;\ \Gamma_t;\ \Gamma_v \vdash (\textbf{type}\ \mathsf{Subject} <:\ \mathsf{mdef.Subject}) \lhd: \\ (\textbf{type}\ \mathsf{Subject} <:\ \mathsf{msuper.Subject})$$

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 Γ_t^{mvc} and Γ_c^{mvc} as defined by (3.28) and (3.29).

Typing* notify*'s implementation: As discussed above, the premise of MOD-OPEN ending $\bar{J}_c{:}\!\Longrightarrow\!\bar{L}_c$ requires that the overridden implementation of class Logger (in $\bar{J}_c^{\mathrm{mvc}}$ which is the **current** aspect of (2.9)) must satisfy its declaration (in $\bar{L}_c^{\mathrm{mvc}}$ as defined by (3.26)). This comes down to showing that the overridden implementation of method notify in mvc's Logger 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):

```
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)):

```
func notify (subj: mself.Subject,                       (3.32)
             ev: mself.Event): Void
```

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

$$\begin{array}{ll} \Gamma;\ \Gamma_t^{\mathsf{mvc}};\ \Gamma_v^{\mathsf{mvc}}; & (3.33)\\ \quad \textbf{val}\ \mathsf{self{:}\ mself.Logger;} & \\ \quad \textbf{val}\ \mathsf{subj{:}\ mself.Subject;} & \\ \quad \textbf{val}\ \mathsf{ev{:}\ mself.Event} & \\ \quad \vdash \mathsf{self.log.printf("event\ from\ \%s{:}\ \%s\ (size{=}\%d)\backslash n",} & \\ \qquad \mathsf{subj.id,\ ev.msg,\ subj.modelSize())\ {:}\ Void} & \end{array}$$

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 is a subtype of msuper.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 equals msuper.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 $\bar{J}_c{:}\!\Longrightarrow\!\bar{L}_c$ is established.

With respect to the discussion of §2.1.2, the declaration of mself given by (2.5) follows from the declarations of Γ_v^{mvc} (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's 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(mself).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 mself ranges over future modules, which must be such that mself<<mvc. Within object mvc, mself and mdef 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-kinded.

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 **impl** 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 $\bar{L}_c \lhd\!: \bar{L}_f$ 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:

```
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:

```
final val outer = mself
```

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

```
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:

future type Logger $<:$ mdef.Logger $\cap$ 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:

```
future hazy class Logger (Stream)                     (4.3)
  extends mdef.Logger with mself.Observer
```

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

```
future operator new Logger (Stream): mself.Logger    (4.4)
```

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:

Show: mdef.Logger $<:$ mdef.Logger $\cap$ mself.Observer
  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$\leftrightarrow m$ (or maybe mself$\leftrightarrow$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 J_c^{core} (see the **current** part of (2.8)), there is a corresponding **type**, **class**, **val** or **func** declaration in L_c^{core} (see (3.8)). The "yields" operator, $:\Longrightarrow$, provides the logical relation between each pair. However, this paper does not present a form of declaration for **module**, but instead has $:\Longrightarrow$ 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

[1] Kim B. Bruce. Some challenging typing issues in object-oriented languages. *Electronic Notes in Computer Science*, 82(8), 2003.

[2] Kim B. Bruce, Adrian Fiech, and Leaf Petersen. Subtyping is not a good "match" for object-oriented languages. In *ECOOP '97 Proceedings, LNCS 1241*, pages 104–127. Springer-Verlag, 1997. Extended abstract.

[3] Kim B. Bruce and J. Nathan Foster. Looj: Weaving loom into java. To appear in *Proceedings of ECOOP 2004*, 2004.

[4] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Prentice Hall, 3rd edition, 2005.

[5] Martin Odersky and al. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[6] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[7] Peter Vanderbilt. Author's repository of research technical notes. At URL `petervanderbilt.com/research/files/technotes/`. See `index.pdf` for more information.

[8] Peter Vanderbilt. Index of P3 technical notes. In [7] as index.pdf⇑, version 3.2.3b, Feb 2014.