Math. Struct. in Comp. Science (1998), *vol.* 8, *pp.* 401–446. Printed in the United Kingdom

A theory of mixin modules: basic and derived operators†

DAVIDE ANCONA and ELENA ZUCCA

Dipartimento di Informatica e Scienze dell'Informazione,
Via Dodecaneso, 35, 16146 Genova (Italy)
Email: {davide, zucca}@disi.unige.it

Received 26 November 1996; revised 18 December 1997

Mixins are modules in which some components are *deferred*, that is, their definition has to be provided by another module. Moreover, in contrast to parameterized modules (like ML functors), mixin modules can be mutually dependent and their composition supports the redefinition of components (*overriding*). In this paper, we present a formal model of mixins and their basic composition operators. These operators can be viewed as a kernel language with clean semantics in which one can express more complex operators of existing modular languages, including variants of inheritance in object-oriented programming. Our formal model is given in an 'institution independent' way, that is, it is parameterized by the semantic framework modelling the underlying core language.

Introduction

In object-oriented languages, the definition of an heir class H from a parent class P usually takes the form $H = extend\ P\ by\ M$, where M denotes a collection of definitions of components (typically methods) which are either new, or redefined with respect to the definition given for them in P (*overriding*). The definitions in M may refer to components defined in P.

A quite natural view of the above situation is to see M as an *abstract heir class*, that is, a class where some components are not defined (*deferred*), and which can be effectively used for instantiation (that is, become *concrete*) only when applied to some parent class, which supplies an implementation for the deferred components. An abstract subclass is sometimes called *mixin* (this name was firstly used in the LISP community, see Moon (1986) and Keene (1989)). At the semantic level, M can be seen as a function from deferred components, that is, components that must be provided from the outside, to components that are defined within the class. Note that, if we assume no overriding, deferred and defined components are disjoint sets.

Most existing object-oriented languages do not support explicit mixins, in the sense

† This work has been partially supported by Murst 40% (Modelli della computazione e dei linguaggi di programmazione) and CNR (Formalismi per la specifica e la descrizione di sistemi ad oggetti).

that it is not possible to define M separately and then to instantiate M on different parent classes, say P_1 and P_2, getting different heir classes $H_1 = M(P_1)$ and $H_2 = M(P_2)$. Allowing the possibility of naming mixins in the language leads to the so-called *mixin-based* inheritance, proposed in Bracha and Cook (1990). Going further, we can take mixins as basic language units, considering concrete classes as particular mixins with no deferred components (semantically, constant functions). That allows a clean and unifying view of different linguistic mechanisms, as explained later in this paper.

First, while mixins in the above sense (abstract heir classes) are not usually supported, some languages allow the definition of *abstract* (*parent*) *classes*, that is, classes with deferred components that can be concreted by heirs, for instance Eiffel (Meyer 1988), C++ (Stroustrup 1991) and Java (Arnold and Gosling 1996), where deferred methods are called *pure virtual* and *abstract*, respectively; in other languages deferred methods are implemented by extra-linguistic features (*cf.* the method `subclassResponsability` in Smalltalk-80 (Goldberg and Robson 1983)).

Actually, it is easy to see that the situation is completely symmetric; referring to the schema above, if P in turn is an abstract class, the combination of P and M can no longer be described as an application $M(P)$, but as the result of a binary *merge* operation. In the resulting class $P + M$, components that were deferred in P have been (possibly) concreted by definitions in M, and *vice versa.* This allows recursive definitions to span module boundaries (Duggan and Sourelis 1996), with a great benefit for modularity, as we illustrate in detail in Section 1.

Second, going back to our original schema of inheritance $H = \textit{extend } P \textit{ by } M$, with P a concrete class, we have ignored until now the fact that M may *redefine* components already defined in P. Since definitions of components can be mutually recursive, redefining some of them actually changes the whole class behaviour. The problem of giving a clean semantics to this mechanism has been independently solved by W. Cook (Cook 1989) and U. S. Reddy (Reddy 1988). The idea consists, roughly speaking, of interpreting P as a function, called by Cook a *generator*, mapping components into components. The current values of components in P are then obtained as the least fixed point of this function.

This approach can be combined in a very natural way with the view of an abstract class as a function from deferred into defined components. Putting the two things together, P can be seen as a function from *input* components into *output* components: the output components are the defined components; the input components are all the (either deferred or defined) components. A function of this kind is called in Cook (1989) an *inconsistent generator* (whose least fixed point cannot be evaluated).

In this paper, we introduce a further distinction, assuming that some defined components are *frozen*, that is, their redefinition cannot change other components (*cf.* non-virtual methods in C++). In this case, some output components are not input components.

A further remark is that, introducing mixins, overriding the definition of a component by a new definition can be seen as the composition of two different operations: first, the old definition is cancelled, obtaining a mixin in which the corresponding component is deferred; then, this mixin is merged with another supplying the new definition. This view of overriding was first introduced, to our knowledge, in Bracha (1992), where the operator that 'cancels' a definition is called *restrict*. Note that in this way it is possible to replace

an asymmetric (non-commutative) binary operator, that is, overriding, by *restrict* plus the commutative *merge* operator. This is the approach we adopt in this paper.

We have shown so far that introducing mixins allows a clean and unifying view of apparently different linguistic mechanisms. We come now to a second point, which is fundamental in this paper. In the discussion above, we have presented mixins as a generalization of classes in the sense of object-oriented programming, as it is from the historical point of view. Anyway, all the preceding considerations are, actually, completely independent of the object and class concepts: the mixin notion can be formulated in the much more general context of module composition. In other words, it is possible to extend the usual notion of module (a collection of definitions of components of different nature, *e.g.* types, functions, procedures, exceptions and so on) to the case where some definitions are deferred, obtaining what we call in this paper a *mixin module* (or simply mixin). Hence, an extension with mixins and corresponding composition operators is, in principle, applicable to any modular language (Bracha 1992; Banavar 1995), allowing the definitions of highly sophisticated module systems with a consequent enhancement of code reusability and extensibility; notice that the notion of mixin module can also be successfully introduced in object-oriented languages where, in many cases, the notion of class turns out to be inadequate as the unique modularity feature offered by the language (Szyperski 1992).

This generalization should be reflected at the semantic level: hence, in this paper, we aim to define a formal model of mixin modules, based on the ideas outlined above, and a corresponding interpretation of composition operators, independent of the semantic nature of a single concrete module, which should depend on the (semantics of) the underlying *core* language, that is, the language for defining module components, following the terminology introduced with Standard ML (Milner *et al.* 1990).

To this end, we generalize the model proposed above (mixins = functions over records of components) to functions over arbitrary semantic structures. This is achieved in a simple and natural way by taking the approach of *institutions* (Goguen and Burstall 1992), where syntactic interfaces (types) of modules are modelled by *signatures* of some institution I, and denotations of program modules are *models* of I; for instance, in a standard case, many-sorted algebras (collections of related functions together with the data domains they operate on). We do not deal explicitly in this paper, which is only concerned with programming modules, with the logical part of the institution concept (a language of sentences for expressing properties that models are required to satisfy); anyway, it should be clear to the reader that the possibility of integration with a specification language is an important motivation behind our approach.

Thus our work can be seen from the technical point of view in two symmetric ways: from one side, we generalize the model of inheritance in Cook (1989) from objects (modelled as records) to modules (modelled as arbitrary semantic structures). From the other side, we extend to the case of mixins the well-established algebraic treatment of module composition (see , for example, Bergstra *et al.* (1990), Diaconescu *et al.* (1993), Sannella and Tarlecki (1988), Ehrig and Mahr (1993), Ehrig *et al.* (1991), Ehrig and Löwe (1993) and Loeckx *et al.* (1996)) presenting a kernel module language with algebraic semantics. The main difference with respect to this literature is that we consider, together with classical operators (like export and renaming), new operators allowing module modification (like

restrict and overriding) that make no sense in the traditional model. Moreover, as already pointed out, we do not deal with specification modules (denoting classes of models), but with programming modules (denoting just one model).

The paper is organized as follows. In Section 1 we present informally the most relevant operators for composing mixins giving some examples written in an SML-like language. In Section 2 we present our formal model of mixins and a set of basic operators. In Section 3 we define a kernel language of mixin modules, with operators like merge, overriding, hiding and functional composition, which can all be expressed in terms of the basic operators introduced in the preceding section. In Section 4 we give a refined version of our model, which allows us to handle (possibly recursive) type definitions in modules. Finally, in Section 5 we make some comparisons with related work and outline our further research on this subject. Technical details and a more concrete model based on the notion of signature inclusion are given in the Appendix. A preliminary presentation of the ideas in this paper has been given in Ancona and Zucca (1996a); the work on mixins presented in the present and other papers (Ancona 1996; Ancona and Zucca 1997; Ancona and Zucca 1998) constitutes a major part of the first author's Ph. D. thesis (Ancona 1998), which we refer to for an organic and extensive presentation.

1. Mixins and mixin operators: an informal introduction

In this section, we introduce the notion of mixin module and the most relevant composition operators by means of some examples written in Standard ML (Milner *et al.* 1990).

We recall that SML supports the definition of module (*structure*), interface (*signature*) and parameterized (or generic) module (*functor*). We have adopted SML for its simplicity, but it should be clear to the reader that our intention here is not to extend a particular modular language with mixins, but to propose a basic set of operators.

1.1. *From concrete modules to mixins*

Assume that we want to implement in SML finite maps and finite sets of integers, with some of the usual operations (empty map, application, updating a map by a new association, getting the domain of a map, restricting the domain of a map, empty set, adding an element to a set and testing for membership). A structure `Map` implementing finite maps should match the following signature `MAP`:

```
signature MAP =
 sig
  type map;
  type set;
  val empty_map:map;
  val apply:map * int → int;
  val update:map * int * int → map;
  val def_dom:map → set;
  val restrict:map * set → map
 end;
```

https://doi.org/10.1017/S0960129598002576 Published online by Cambridge University Press

For a modular development, the implementation of the type `set` should be provided separately by a structure `Set` over the signature `SET`:

```
signature SET =
 sig
 type set;
  val empty_set:set;
  val add:int * set → set;
  val is_in:int * set → bool
 end;
```

This can be achieved in SML by defining `Map` as a functor from structures matching `SET` to structures matching `MAP` instead of as a structure. However, this solution is still inadequate since there is no way to implement the function `restrict` in terms of the functions in `SET`; intuitively, we miss the possibility of 'iterating' an action over all the elements of a set. Conversely, it is not possible to define `Set` as a functor taking `Map` as parameter, since there is no way to implement the function `def_dom` in terms of the functions in `Map`.

Of course we could handle the problem by adding new primitives, but that solution implies extra code and could lead to inefficient implementations of `restrict` and `def_dom`, respectively. On the other side, the `restrict` and `def_dom` functions could be efficiently defined inside a unique structure, but this would lose modularity.

A solution that keeps both modularity and efficiency would be to move `restrict` from `Map` to `Set`. However, in this way the two structures become mutually dependent, while functors in SML are only able to express one-way dependencies. This problem is overcome by the introduction of *mixin modules*. A mixin, like a functor, is a module that depends on other modules; the crucial difference is that mixins allow us to express mutual dependency. In a hypothetical extension of SML with mixins, the solution of our problem is the following:

```
mixin Map =
 mix
  datatype map = empty_map | update of map * int * int;
  type set;
  val empty_set:set;
  val add:int * set → set;
  val restrict:map * set → map;
  fun apply(update(f,x,y),z) = ...
  fun def_dom(empty_map) = empty_set |
      def_dom(update(f,x,_)) = add(x,def_dom(f))
 end;

mixin Set =
 mix
  datatype set = empty_set | add of int * set;
  type map;
```

https://doi.org/10.1017/S0960129598002576 Published online by Cambridge University Press

```
  val empty_map:map;
  val apply:map * int → int;
  val update:map * int * int → map;
  val def_dom:map → set;
  fun is_in(_,empty_set) = ...|...
  fun restrict(_,empty_set) = empty_map |
      restrict(f,add(x,s)) =
      if is_in(x,def_dom(f)) then update(restrict(f,s),x,apply(f,x))
      else restrict(f,s)
 end;
```

As shown by the example, a mixin differs from a standard module, which is a collection of definitions, since the implementation of some components may be *deferred*, as happens for the type `set` and the functions `empty_set`, `add` and `restrict` in `Map`. Correspondingly, SML signatures could be extended to mixin signatures by labelling components in such a way that each component is recognized as either defined or deferred†. For instance, the mixin `Set` matches the following mixin signature:

```
mixin signature SET =
 mix sig
  type set;
  val empty_set:set;
  val add:int * set → set;
  type map deferred;
  val empty_map:map deferred;
  val apply:map * int → int deferred;
  val update:map * int * int → map deferred;
  val def_dom:map → set deferred;
  val is_in:int * set → bool;
  val restrict:map * set → map
 end;
```

Note that, in general, it is not possible to split the signature of a mixin in two disjoint signatures corresponding to the defined and deferred components, respectively. For instance, in `SET` above, the deferred components (`map`, `empty_map`, `apply`, `update`, `def_dom`) do not form a signature, since the functionality of `def_dom` contains the defined type `set`.

Modules in the usual sense can be viewed as particular mixin modules in which all the components are defined. We call them *concrete* mixin modules. Of course a mixin module, for example `Map`, that is not concrete cannot be used in isolation, just as a parameterized module needs to be instantiated over an actual parameter. A way to combine mixins in order to obtain a concrete module eventually is provided by the *merge* operator (Bracha 1992): if M_1 and M_2 are two mixins, then $M_1 \oplus M_2$ is a mixin, where some definitions of M_1 are associated with corresponding declarations in M_2, and *vice*

† After introducing overriding, we will further distinguish defined components in either *frozen* or *virtual*, as illustrated in the following subsection.

https://doi.org/10.1017/S0960129598002576 Published online by Cambridge University Press

versa. This operator is commutative and is defined whenever no components are defined on both sides.

In the example above, we can define a mixin `SetAndMap = Set⊕Map` matching the signature `SET_AND_MAP = SET⊕MAP`; in `SetAndMap` every component has a definition, hence we have obtained a concrete module (a normal SML structure). This is not always the case: if some deferred component on one side has no corresponding definition on the other side, the module resulting from merging is still not concrete.

Finally, we stress the fact that mutually dependent modules arise in a very natural way at the design stage of software development, and, moreover, the lack of an ability to manage them causes code duplication and loss of software extensibility. For other examples of mutually dependent modules (including concrete cases) we refer to Ancona (1996), Banavar and Lindstrom (1996) and Duggan and Sourelis (1996).

1.2. *Overriding definitions*

In object-oriented languages, inheritance allows overriding of definitions with precedence of either the heir, as, for example, in Smalltalk (Goldberg and Robson 1983) or the parent class, as in Beta (Lehrmann *et al.* 1993), as has been analyzed in Bracha and Cook (1990).

By introducing the notion of mixin, overriding the definition of a component, say $f = expr$, in a module M, by a new definition $f = expr'$, can be seen as the composition of two operations: first, the old definition is cancelled, getting a module M' where f is deferred; then, M' is merged with $f = expr'$.

This view allows a more general version of overriding. Consider, for instance, the following mixin, which leaves deferred the implementation of lists of integers, and only defines two derived functions:

```
mixin AbsList =
 mix
  type int_list;
  val head:int_list → int;
  val tail:int_list → int_list;
  val is_empty:int_list → bool;
  fun length(l) = if is_empty(l) then 0 else length(tail(l))+1;
  fun eq(l1,l2) =
   if is_empty(l1) then is_empty(l2) else
   if is_empty(l2) then false else
   head(l1)=head(l2) andalso eq(tail(l1),tail(l2))
 end;
```

and assume that we want to combine it with another mixin where an implementation for the type `int_list` is provided, together with a more efficient version of `length` based on this implementation.

https://doi.org/10.1017/S0960129598002576 Published online by Cambridge University Press

```
mixin List =
 mix
  type int_list = int * int list;
  fun length(n,_) = n;
  ...
 end;
```

The sum `AbsList` $\oplus$ `List` is not correct, since the function `length` is defined in both the modules. We need a way of explicitly specifying which of the two definitions of `length` must take the precedence. This can be achieved by using a *restrict* operator (Bracha 1992), which allows us to cancel definitions inside a mixin.

This operator takes two arguments, a mixin M and a mixin signature specifying the defined components of M that have to be changed to deferred. The mixins `AbsList` and `List` can be combined as follows:

$$\texttt{mixin AbsListAndList} \;=\; (\texttt{restrict length in AbsList}) \oplus \texttt{List}.$$

In general the restrict operation allows us to specify, for each pair of conflicting definitions in two mixins M_1 and M_2, which of the two definitions must take the precedence. In this way we are able to define more combinations of M_1 and M_2 than by simply using an overriding operator that gives precedence to either M_1 or M_2 (Ancona and Zucca 1997). This is essential, for instance, in object-oriented languages supporting multiple inheritance where an heir class may inherit a method definition from several parent classes (Bracha 1992; Van Limbergehn and Mens 1996).

The inheritance operators of Smalltalk and Beta (without considering the pseudo-variables *super* and *inner*) can be defined by an appropriate combination of the merge and restrict operations:

$$H = (\texttt{restrict } \Sigma_P \texttt{ in } P) \oplus M \text{ (in Smalltalk)},$$
$$H = P \oplus (\texttt{restrict } \Sigma_M \texttt{ in } M) \text{ (in Beta)},$$

where H and P are the heir and the parent class, respectively, M is the mixin corresponding to the new definitions, Σ_P contains the components of P that are redefined in M, and analogously for Σ_M. A way to express the Smalltalk and Beta inheritance operators handling the pseudo-variables *super* and *inner* is described in Section 3, and in much more detail in Ancona and Zucca (1997).

From the semantic point of view, the possibility of overriding definitions leads one to consider two different interpretations of a concrete module, which we call *closed* and *open semantics*, respectively (as mentioned in the introduction, this idea is originally due to W. Cook (Cook 1989) and U. S. Reddy (Reddy 1988), and is now a standard approach). Consider, for instance, the structure:

```
structure S =
 struct
  val i1=1;
  val i2=2*i1;
  val i3=i1+i2
 end;
```

https://doi.org/10.1017/S0960129598002576 Published online by Cambridge University Press

The closed semantics is a model over the signature Σ^{out}, that is, it associates a semantic value with each component of the module; in this case, $\Sigma^{out} = \{\mathtt{i1}, \mathtt{i2}, \mathtt{i3}\}$ and the closed semantics of `S` is the record $<\mathtt{i1}{:}1, \mathtt{i2}{:}2, \mathtt{i3}{:}3>$. Anyway, this semantics does not take into account the possibility of later redefinitions of components. For instance, redefining `i1` to 2 changes the values of `i2` and `i3` too. In the general case, definitions can be mutually recursive. In order to model this, it is necessary to see `S` as a function from models over Σ^{out} to models over Σ^{out} (open semantics); in this case, the function defined by

$$<\mathtt{i1}{:}x, \mathtt{i2}{:}y, \mathtt{i3}{:}z> \mapsto <\mathtt{i1}{:}1, \mathtt{i2}{:}2 * x, \mathtt{i3}{:}x + y>.$$

Note that in this concrete case the closed semantics is the unique fixed point of the open semantics. However, this fact is not relevant for our technical treatment in the following; we only assume a *freeze* operator at the semantic level that extracts one fixed model from a function from models into models (that is, gets closed semantics from open semantics).

As we said earlier, open semantics is needed for correctly modelling overriding. Consider, for instance, the following structure `S′`, which provides a different definition for `i2`:

```
structure S′ =
 struct
  val i2=3
 end;
```

The open semantics of the mixin expression `(restrict i2 in S)` $\oplus$ `S′` is the function

$$<\mathtt{i1}{:}x, \mathtt{i2}{:}y, \mathtt{i3}{:}z> \mapsto <\mathtt{i1}{:}1, \mathtt{i2}{:}3, \mathtt{i3}{:}x + y>,$$

which can be obtained from the open semantics of `S` by replacing the definition of `i2`, and gives as corresponding closed semantics the record $<\mathtt{i1}{:}1, \mathtt{i2}{:}3, \mathtt{i3}{:}4>$. Note that there is no way to obtain this record directly from the closed semantics of `S`.

As a final example, consider the following structure:

```
structure S″ =
 struct
  val i2=2
 end;
```

Then, again, the open semantics of the mixin expression `(restrict i2 in S)` $\oplus$ `S″` is different from that of `S`, even though in this case the closed semantics coincide.

The open semantics can be naturally extended to non-concrete mixins (where some components are deferred), seeing them as functions from models over a larger signature Σ^{in} to models over Σ^{out} (hence $\Sigma^{out} \subseteq \Sigma^{in}$). Of course, in this case it makes no sense to consider the closed semantics – corresponding to the fact that the module cannot be effectively 'used' as it stands.

1.3. *Freeze and hiding*

In the example above, a redefinition of `i2` in `S` changes the semantics of `i3`, too. In order to express this, we say this `i2` is a *virtual* component of `S`. In general, we introduce a further distinction of defined components into either *virtual* or *frozen* (*cf.*, for instance,

https://doi.org/10.1017/S0960129598002576 Published online by Cambridge University Press

virtual and *non virtual member functions* in C++). If a defined component is virtual, its redefinition may change the semantics of some other component of the mixin; by contrast, a redefinition of a frozen component cannot affect the semantics of any other component.

The freeze operation (Bracha 1992) allows us to freeze a virtual component f by eliminating all the dependences of the other components on f. As an example, assume we freeze `i2` in `S`. Then the structure `S1` we obtain is equivalent to the following[†]:

```
structure S1 =
 struct
  val i1=1;
  val i2=2*i1;
  local val i2=2*i1 in
  val i3=i1+i2 end
 end;
```

In other words, we get a structure whose open semantics is a function constant with respect to the component `i2`:

$$<\texttt{i1}:x, \texttt{i2}:y, \texttt{i3}:z> \mapsto <\texttt{i1}:1, \texttt{i2}:2 * x, \texttt{i3}:3 * x>.$$

Now a redefinition of `i2` cannot change the semantics of `i3`.

In order to keep this information explicit in the module interface, we can remove the assumption $\Sigma^{out} \subseteq \Sigma^{in}$ and say that the open semantics of `S1` is a function that no longer takes `i2` as argument. Correspondingly, we may add a new kind of label to function symbols within mixin signatures, specifying for each defined function whether it is virtual (default case) or frozen. For instance, `S1` matches the following mixin signature:

```
mix sig
 val i1 :int;
 val i2 :int frozen;
 val i3 :int
end;
```

The open semantics of a mixin matching this signature is a function taking `i1` and `i3` as arguments, and returning values for `i1`, `i2` and `i3`. For instance, the open semantics of `S1 = freeze i2 in S` is

$$<\texttt{i1}:x, \texttt{i3}:z> \mapsto <\texttt{i1}:1, \texttt{i2}:2 * x, \texttt{i3}:3 * x>.$$

For a mixin M where all the components are frozen, the open semantics is a constant function that returns the closed semantics of M. Note that it makes no sense to freeze a deferred component.

Now consider the possibility of hiding a defined component of a mixin. Hiding a defined function f intuitively corresponds to consider f as a locally declared function.

[†] To be precise, `i2` should be a local variable visible to all definitions in `S` – this cannot be expressed in SML, since the definition of `i2` depends on `i1`. However, in this case, `i2` can equivalently be local to the definition of `i3`, since `i1` does not depend on `i2`.

https://doi.org/10.1017/S0960129598002576 Published online by Cambridge University Press

For instance, assume we hide `i2` in the structure `S` shown above. We expect to obtain a structure equivalent to the following:

```
structure S2 =
 struct
  val i1=1;
  local val i2=2*i1 in
  val i3=i1+i2 end
 end;
```

Since the definition of `i2` is now local to `S2`, no operation on `S2` can modify its value; if we merge `S2` with a mixin that defines a component `i2`, the value of `i3` does not change. As a matter of fact, hiding `i2` corresponds to first freezing it and then deleting its definition by means of the restrict operation.

2. A parameterized framework for mixins

In this section we present our formal framework for mixin modules. More precisely, we define *mixin signatures*, *mixin models* and three basic operators over mixin models that will be used in the following section for expressing all the operators of our kernel language of mixin modules. The framework we introduce is parameterized by the semantic framework modelling the core level (*core framework*).

2.1. *Core frameworks*

Roughly speaking, a core framework is a specialization of the notion of *model part* of an institution, that is, a category of signatures and a model functor (Goguen and Burstall 1992). We assume an additional feature, *viz.* a family of operators (one for each signature Σ) taking functions from Σ-models to Σ-models and returning Σ-models. These operators intuitively correspond to extract closed semantics from open semantics.

Definition 2.1. A *model part* is a pair $<\mathbf{Sig}, Mod>$ where

— **Sig** is a category, whose objects are called *signatures*.
— Mod is a functor[†], $Mod : \mathbf{Sig}^{op} \rightarrow \mathbf{Set}$. For any signature Σ, objects in $Mod(\Sigma)$ are called *models* over Σ or Σ-*models*. For any signature morphism $\sigma : \Sigma \rightarrow \Sigma'$, $Mod(\sigma)$ is called the *reduct* via σ and denoted by $-_{|\sigma}$. We write $-_{|\Sigma_i}$ for $-_{|j_i}$, when $j_i : \Sigma_i \rightarrow \Sigma$, $i = 1, 2$, are the injections of a coproduct.

Notation. Let **Sig** be a category with all finite colimits. If Σ_1 and Σ_2 are two signatures in **Sig**, we use $\Sigma_1 + \Sigma_2$ and $\varnothing$ to denote the unique (up to isomorphism) coproduct of Σ_1 and Σ_2 and the initial object in **Sig**, respectively.

[†] There is no problem in allowing categories of models here, but model morphisms are not relevant for the subject of this paper, so we leave them out here.

https://doi.org/10.1017/S0960129598002576 Published online by Cambridge University Press

For any (small) category $\mathbf{C}$ the hom-functor $Hom_{\mathbf{C}}: \mathbf{C}^{op} \times \mathbf{C} \to \mathbf{Set}$ is defined as follows:

— $Hom_{\mathbf{C}}(<A, B>)$ is the set of $\mathbf{C}$-morphisms from A to B
— $Hom_{\mathbf{C}}(<h, k>) = k \circ \_ \circ h$.

If $\mathbf{C}$ is **Set** we also use the notation $Hom_{\mathbf{C}}(<A, B>) = B^A$.

Note that if $h: A \to A'$ in $\mathbf{C}^{op}$, then $h: A' \to A$ in $\mathbf{C}$, whence the compositions (all in $\mathbf{C}$) are well defined.

Definition 2.2. A model part is *regular* if **Sig** has all finite colimits and *Mod* preserves them.

Definition 2.3. A *core framework* is a triple $<$**Sig**, *Mod*, *freeze*$>$ where

— $<$**Sig**, *Mod*$>$ is a regular model part.
— *freeze* is a family of functions (that is, morphisms in **Set**)

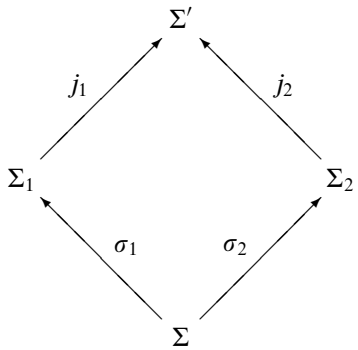
$$freeze_{\Sigma}: Mod(\Sigma)^{Mod(\Sigma)} \to Mod(\Sigma)$$

indexed over the signatures Σ in **Sig**.

We omit the subscript whenever it can be unambiguously determined from the context.

In concrete cases, each $freeze_{\Sigma}$ could correspond to some kind of 'fixed point' operator, as illustrated in the example below. However, no properties on *freeze* are required for defining mixin models and their basic operators.

The assumption that *Mod* preserves finite colimits ensures the existence of the *amalgamated sum* (see, for example, Ehrig and Mahr (1985)); indeed, *Mod* maps pushouts in **Sig** into pullbacks in **Set**, and this implies that, for every pushout diagram in **Sig**,



and for every pair of models $A_i \in Mod(\Sigma_i)$, $i = 1, 2$, such that $A_{1|\sigma_1} = A_{2|\sigma_2}$, there exists a unique model in $Mod(\Sigma')$, denoted by $A_1 + A_2$, such that $(A_1 + A_2)_{|j_i} = A_i$, $i = 1, 2$. In particular, when we consider coproducts instead of pushouts, we have that, for any $A_i \in Mod(\Sigma_i)$, there exists a unique model $A_1 + A_2$ in $Mod(\Sigma_1 + \Sigma_2)$ such that $(A_1 + A_2)_{|\Sigma_i} = A_i$, $i = 1, 2$ (recall that in all categories with initial objects a coproduct is a particular kind of pushout).

Example 2.4. As a standard concrete example of core framework, the reader can think of a language with a fixed set of predefined types, say T, where modules are collections of possibly recursive function definitions. In this case signatures are just $T^{suchthatar} \times T$-families of sets of (function) symbols. Assuming that each type $t \in T$ denotes a set of

https://doi.org/10.1017/S0960129598002576 Published online by Cambridge University Press

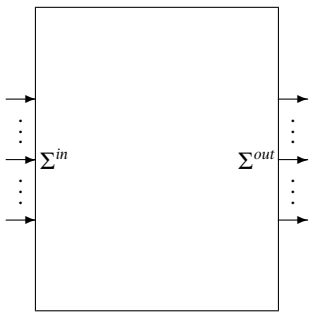


Fig. 1. A pictorial view of mixins

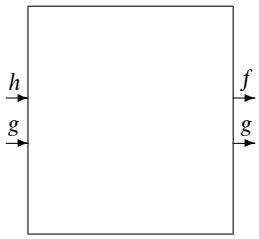


Fig. 2. A mixin

values† V_t, a model over a signature Σ associates with each function symbol in $\Sigma_{s_1 \ldots s_n, s}$ a partial function from $V_{s_1} \times \ldots \times V_{s_n}$ into V_s. In other words, a Σ-model is a Σ-family of partial functions. Then, a function $F : Mod(\Sigma) \to Mod(\Sigma)$ transforms families of partial functions into families of partial functions, and $freeze_\Sigma$ is expected to be the least fixed point of F in the usual sense whenever F is continuous.

2.2. *Mixin Signatures and Models*

We now give the formal definition of mixin signatures and models. In this subsection we assume a fixed core framework $<$**Sig**, Mod, $freeze>$.

Definition 2.5. A mixin signature is a pair $<\Sigma^{in}, \Sigma^{out}>$ of signatures in **Sig**. A mixin model over the mixin signature $<\Sigma^{in}, \Sigma^{out}>$ is a function in $Mod(\Sigma^{out})^{Mod(\Sigma^{in})}$.

Intuitively, the input signature gives all the components which definitions in the module may depend on, while the output signature gives all the components which are defined in the module.

A pictorial view is given in Figure 1, where a mixin model is represented as a black box having some inputs and outputs.

For instance, a mixin model F having two input components h, g and two output components f, g is depicted in Figure 2. This mixin model is a function from records

† We assume a fixed universe of values in order to avoid foundational problems.

https://doi.org/10.1017/S0960129598002576 Published online by Cambridge University Press

of the form $<h:\_, g:\_>$ into records of the form $<f:\_, g:\_>$; in other words, F associates with the output components f and g two values (definitions) that depend on the values of the input components h, g. In order to express this fact we represent F by the more suggestive notation

$$f \mapsto F_f(h, g)$$
$$g \mapsto F_g(h, g).$$

Referring to the core framework defined in Example 2.4 and the modules `S` and `S'` defined in Section 1, we have that the (open) semantics of `S` is a mixin model F_S over the mixin signature $<\Sigma_S^{in}, \Sigma_S^{out}>$ where

$$\Sigma_S^{in} = \Sigma_S^{out} = \{i1, i2, i3 : \rightarrow \texttt{int}\}$$

for any $A \in Mod(\Sigma_S^{in})$, $F_S(A) = B \in Mod(\Sigma_S^{out})$ such that

$$i1^B = 1, i2^B = 2 * i1^A, i3^B = i1^A + i2^A.$$

Using the above notation, F_S is represented by

$$i1 \mapsto 1$$
$$i2 \mapsto 2 * i1$$
$$i3 \mapsto i1 + i2.$$

Moreover, $freeze(F_S) = C \in Mod(\Sigma_S^{out})$ (the closed semantics of `S`), where $i1^C = 1$, $i2^C = 2$ and $i3^C = 3$.

Finally, the mixin expression `(restrict i2 in S)`$\oplus$ `S'` denotes a mixin model F'_S again over $<\Sigma_S^{in}, \Sigma_S^{out}>$ such that, for any $A \in Mod(\Sigma_S^{in})$, $F'_S(A) = B \in Mod(\Sigma_S^{out})$ such that $i1^B = 1$, $i2^B = 3$, $i3^B = i1^A + i2^A$.

The corresponding closed semantics is given by $freeze(F'_S) = D \in Mod(\Sigma_S^{out})$, where $i1^D = 1$, $i2^D = 3$ and $i3^D = 4$.

We now show that mixin signatures and models defined above actually constitute a model part in the sense of Definition 2.1.

Set **MixSig** = $\mathbf{Sig}^{op} \times \mathbf{Sig}$; we recall that a morphism in **MixSig** from $<\Sigma_1^{in}, \Sigma_1^{out}>$ to $<\Sigma_2^{in}, \Sigma_2^{out}>$ is a pair of morphisms $\sigma^{in} : \Sigma_2^{in} \rightarrow \Sigma_1^{in}$, $\sigma^{out} : \Sigma_1^{out} \rightarrow \Sigma_2^{out}$ in **Sig** and that $\mathbf{MixSig}^{op} = \mathbf{Sig} \times \mathbf{Sig}^{op}$.

Definition 2.6. The functor $MixMod : \mathbf{MixSig}^{op} \rightarrow \mathbf{Set}$ is defined by

$$MixMod = Hom_{\mathbf{Set}} \circ <Mod \circ \pi_1, Mod \circ \pi_2>,$$

where $\pi_1 : \mathbf{Sig} \times \mathbf{Sig}^{op} \rightarrow \mathbf{Sig}$ and $\pi_2 : \mathbf{Sig} \times \mathbf{Sig}^{op} \rightarrow \mathbf{Sig}^{op}$ are the projection functors. Thus

— for any object $<\Sigma^{in}, \Sigma^{out}>$ in $\mathbf{Sig} \times \mathbf{Sig}^{op}$,

$$MixMod(<\Sigma^{in}, \Sigma^{out}>) = Mod(\Sigma^{out})^{Mod(\Sigma^{in})};$$

— for any $<\sigma^{in}, \sigma^{out}> : <\Sigma_1^{in}, \Sigma_1^{out}> \rightarrow <\Sigma_2^{in}, \Sigma_2^{out}>$,

$$MixMod(<\sigma^{in}, \sigma^{out}>) = Mod(\sigma^{out}) \circ \_ \circ Mod(\sigma^{in}).$$

https://doi.org/10.1017/S0960129598002576 Published online by Cambridge University Press

Note that for any morphism $<\sigma^{in}, \sigma^{out}>: <\Sigma_1^{in}, \Sigma_1^{out}> \to <\Sigma_2^{in}, \Sigma_2^{out}>$ in $\mathbf{MixSig}^{op}$, $MixMod(<\sigma^{in}, \sigma^{out}>)$ denotes a function from $Mod(\Sigma_1^{out})^{Mod(\Sigma_1^{in})}$ to $Mod(\Sigma_2^{out})^{Mod(\Sigma_2^{in})}$ corresponding to the reduct via $<\sigma^{in}: \Sigma_1^{in} \to \Sigma_2^{in}, \sigma^{out}: \Sigma_2^{out} \to \Sigma_1^{out}>$ for mixin models. Using the lambda notation, we can express the definition of the reduct as follows:

$$MixMod(<\sigma^{in}, \sigma^{out}>) = \lambda F.\lambda A.(F(A_{|\sigma^{in}}))_{|\sigma^{out}}.$$

Henceforth we will use the notation ${}_{\sigma^{in}|}F_{|\sigma^{out}}$ for $MixMod(<\sigma^{in}, \sigma^{out}>)(F)$; we also omit the input (respectively, output) morphism when it is the identity and write ${}_{\Sigma^{in}|}F_{|\Sigma'^{out}}$ for ${}_{j^{in}|}F_{|j^{out}}$ when $j^{in}: \Sigma'^{in} \to \Sigma^{in}$ and $j^{out}: \Sigma'^{out} \to \Sigma^{out}$ are injections of coproducts.

Fact 2.7. *MixMod* is a functor such that $MixMod \circ Emb \cong Mod$, where

$$Emb: \mathbf{Sig}^{op} \to \mathbf{Sig} \times \mathbf{Sig}^{op}$$

is defined by

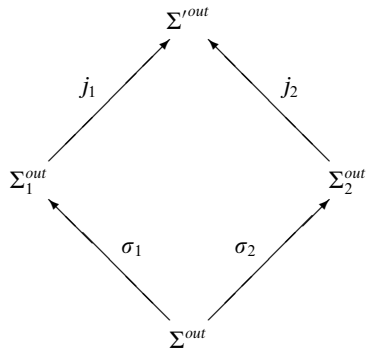
$$Emb(\Sigma) = <\emptyset, \Sigma>, Emb(\sigma) = <id_{\emptyset}, \sigma>.$$

Note that the natural isomorphism $\eta: Mod \to MixMod \circ Emb$ is such that, for any Σ in **Sig**, $\eta_{\Sigma}: Mod(\Sigma) \to Mod(\Sigma)^{Mod(\emptyset)}$ is the arrow isomorphic to the projection morphism

$$\pi_{\Sigma}: Mod(\Sigma) \times Mod(\emptyset) \to Mod(\Sigma) \text{ in } \mathbf{Set}.$$

We now define the equivalent of the amalgamated sum for mixin models. Since, in general, **MixSig** does not have all finite colimits, the amalgamation property does not hold (however, we can define a less general form of amalgamated sum by fixing the input signature Σ^{in}).

Proposition 2.8. Let the following diagram



be a pushout in **Sig**. Then, for any $F_i \in MixMod(<\Sigma^{in}, \Sigma_i^{out}>)$, $i = 1, 2$, such that $F_{1|\sigma_1} = F_{2|\sigma_2}$, there exists a unique $F \in MixMod(<\Sigma^{in}, \Sigma'^{out}>)$, denoted by $F_1 + F_2$, such that $F_{|j_i} = F_i$, $i = 1, 2$.

Proof. The proof comes directly from the fact that the hom-functor

$$Hom_{\mathbf{Set}}(<Mod(\Sigma^{in}), \_>): \mathbf{Set} \to \mathbf{Set}$$

and the model functor $Mod: \mathbf{Sig}^{op} \to \mathbf{Set}$ preserve finite limits. Thus $MixMod(<\Sigma^{in}, \_>)$ maps finite colimits (of a diagram D) of signatures to limits (of the diagram $Mod \circ D$) of model sets. Hence, $MixMod(<\Sigma^{in}, \Sigma'^{out}>)$ is a pullback object in **Set**. □

https://doi.org/10.1017/S0960129598002576 Published online by Cambridge University Press

One can easily verify that $F_1 + F_2$ is defined by

$$\text{for any } A \in Mod(\Sigma^{in}), (F_1 + F_2)(A) = F_1(A) + F_2(A).$$

As happens for the amalgamated sum of models, we obtain a particular case of sum between mixin models when we consider coproducts of output signatures instead of pushouts.

We conclude this subsection by defining an extension of the family of operators *freeze*.

Definition 2.9. Let:

— F be a mixin model in $MixMod(<\Sigma^{fr} + \Sigma^{in}, \Sigma^{out}>)$
— $\sigma : \Sigma^{fr} \to \Sigma^{out}$ be a signature morphism.

Then $freeze_\sigma$ denotes the function from

$$MixMod(<\Sigma^{fr} + \Sigma^{in}, \Sigma^{out}>)$$

to

$$MixMod(<\Sigma^{in}, \Sigma^{out}>)$$

defined by

$$freeze_\sigma(F)(A) = freeze_{\Sigma^{out}}(F \circ (A + \_) \circ Mod(\sigma))$$

for any F in $MixMod(<\Sigma^{fr} + \Sigma^{in}, \Sigma^{out}>)$ and A in $Mod(\Sigma^{in})$.

Here $(\_ + \_)$ denotes the function from $Mod(\Sigma^{in}) \times Mod(\Sigma^{fr})$ to $Mod(\Sigma^{in} + \Sigma^{fr})$, mapping each pair $<A, B>$ to $A + B$. Note that, more correctly, *freeze* should also be indexed over Σ^{in}, but we have omitted this here for the sake of simplicity. Using the lambda notation, $freeze_\sigma$ can be expressed as follows:

$$freeze_\sigma = \lambda F.\lambda A.freeze_{\Sigma^{out}}(\lambda X.F(A + X_{|\sigma})).$$

Intuitively, applying the $freeze_\sigma$ operator to a mixin model corresponds to (permanently) associating with some input components (Σ^{fr}) the definitions of some output components, in the way specified by σ, so that these components disappear from the input signature. In fact, in the rest of the paper we will use the operator $freeze_\sigma$ only when σ is the injection of a coproduct, but here we have considered any kind of morphism for generality.

2.3. *Basic operators*

In the previous subsection we actually defined three basic operators over mixin models that will be used in the next section for expressing a variety of higher level mixin combinators.

The graphical representation given in Section 2.2 suggests looking at mixin models as electronic devices and operations for mixin composition as rules for constructing from a set of mixins (devices) $M_1, \ldots, M_n$ a mixin M 'containing' $M_1, \ldots, M_n$ inside, by connecting inputs and outputs in an intuitively suitable way: more precisely, according to the four following rules:

— each input of $M_1, \ldots, M_n$ must be connected exactly to either one input of M or one output of some M_i, $i = 1..n$;

https://doi.org/10.1017/S0960129598002576 Published online by Cambridge University Press

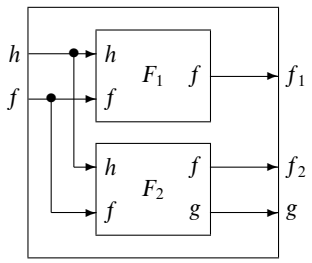


Fig. 3. Sum

— each output of $M_1, \ldots, M_n$ may either be connected to some outputs of M or inputs of $M_1, \ldots, M_n$;
— each input of M may be connected to some inputs of $M_1, \ldots, M_n$;
— each output of M must be connected to exactly one output of some M_i, $i = 1..n$.

Sum Given two mixin models:

— F_1 in $MixMod(<\Sigma^{in}, \Sigma_1^{out}>)$
— F_2 in $MixMod(<\Sigma^{in}, \Sigma_2^{out}>)$,

their sum $F_1 + F_2$ is a mixin model in $MixMod(<\Sigma^{in}, \Sigma_1^{out} + \Sigma_2^{out}>)$.

As an example, consider the following mixin models F_1 and F_2:

$$F_1 = f \mapsto (F_1)_f(h, f)$$

$$F_2 = f \mapsto (F_2)_f(h, f)$$
$$g \mapsto (F_2)_g(h, f)$$

Then, the sum $F_1 + F_2$ is given by

$$f_1 \mapsto (F_1)_f(h, f)$$
$$f_2 \mapsto (F_2)_f(h, f)$$
$$g \mapsto (F_2)_g(h, f).$$

The situation is sketched graphically in Figure 3. According to the intuitive composition rules given above, all the inputs are shared, while the outputs are kept distinct (as happens for the two components f). The sum operator intuitively represents the most primitive way of combining together two mixins and is the natural extension of the amalgamated sum over models. We will see in the next section how it is possible to define more complex binary operators on top of sum and the other two basic operators presented below.

We now show an alternative definition of sum, which will be used in Section 4. Given two mixin models F_1 in $MixMod(<\Sigma_1^{in}, \Sigma_1^{out}>)$ and F_2 in $MixMod(<\Sigma_2^{in}, \Sigma_2^{out}>)$, we define their combination $F_1 \uplus F_2$ as the mixin model in $MixMod(<\Sigma_1^{in} + \Sigma_2^{in}, \Sigma_1^{out} + \Sigma_2^{out}>)$ defined

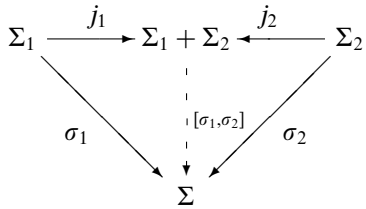
https://doi.org/10.1017/S0960129598002576 Published online by Cambridge University Press

by

$$\text{for any } A \in Mod(\Sigma_1^{in} + \Sigma_2^{in}), (F_1 \uplus F_2)(A) = F_1(A_{|\Sigma_1^{in}}) + F_2(A_{|\Sigma_2^{in}}).$$

Although the two operators $+$ and $\uplus$ have the same expressive power (see Fact.2.10 below), $\uplus$ does not correspond to any amalgamation property, therefore, whenever possible, we will use $+$ instead of $\uplus$.

Notation. If $\sigma_1 : \Sigma_1 \to \Sigma$ and $\sigma_2 : \Sigma_2 \to \Sigma$ are two signature morphisms, we denote by $[\sigma_1, \sigma_2]$ the unique morphism from $\Sigma_1 + \Sigma_2$ to Σ making the following diagram commute:



Fact 2.10. For any pair of mixin models

$$F_1 \text{ in } MixMod(<\Sigma^{in}, \Sigma_1^{out}>),$$
$$F_2 \text{ in } MixMod(<\Sigma^{in}, \Sigma_2^{out}>),$$

$$F_1 + F_2 = {}_{[id_{\Sigma^{in}}, id_{\Sigma^{in}}]|}(F_1 \uplus F_2).$$

For any pair of mixin models

$$F_1 \text{ in } MixMod(<\Sigma_1^{in}, \Sigma_1^{out}>),$$
$$F_2 \text{ in } MixMod(<\Sigma_2^{in}, \Sigma_2^{out}>),$$

$$F_1 \uplus F_2 = {}_{\Sigma_1^{in}+\Sigma_2^{in}|}F_1 + {}_{\Sigma_1^{in}+\Sigma_2^{in}|}F_2.$$

Reduct Given a mixin model F in $MixMod(<\Sigma^{in}, \Sigma^{out}>)$ and two signature morphisms $\sigma^{in} : \Sigma^{in} \to \Sigma'^{in}$ and $\sigma^{out} : \Sigma'^{out} \to \Sigma^{out}$, the reduct ${}_{\sigma^{in}|}F_{|\sigma^{out}}$ is a mixin model in

$$MixMod(<\Sigma'^{in}, \Sigma'^{out}>).$$

As an example, consider the following mixin model F:

$$f \mapsto F_f(f, g, h)$$
$$g \mapsto F_g(f, g, h).$$

Let $\sigma^{in} : \{f, g, h\} \to \{x, z\}$ and $\sigma^{out} : \{y\} \to \{f, g\}$ be the morphisms mapping f, g, h into x and y into f, respectively. Then, ${}_{\sigma^{in}|}F_{|\sigma^{out}}$ is given by

$$y \mapsto F_f(x, x, x).$$

The situation is sketched graphically in Figure 4. The figure shows that the informal composition rules are respected by the reduct operator. Note that through the reduct it

https://doi.org/10.1017/S0960129598002576 Published online by Cambridge University Press

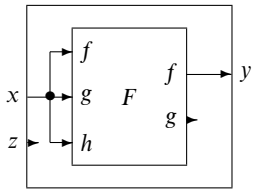


Fig. 4. Reduct

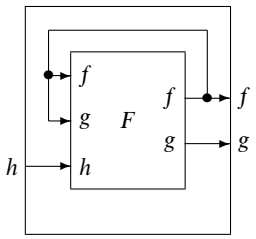


Fig. 5. Freeze

is possible to add dummy input components (like z) and to forget output components (like g). The reduct is a powerful renaming operator allowing us to rename input and output components in a separate way; again, this operator is the natural extension of the corresponding operator at the level of models.

Freeze Given a mixin model F in $MixMod(<\Sigma^{fr} + \Sigma^{in}, \Sigma^{out}>)$ and a signature morphism $\sigma: \Sigma^{fr} \to \Sigma^{out}$, $freeze_{\sigma}(F)$ is a mixin model in $MixMod(<\Sigma^{in}, \Sigma^{out}>)$.

As an example, let F be the previous mixin model and let σ be the signature morphism mapping f and g into f. Then, $freeze_{\sigma}(F)$ is given by:

$$f \mapsto F'_f(h)$$
$$g \mapsto F'_g(h),$$

where $F'(h) = freeze(\lambda<f, g>.F(f, f, h))$.

The situation is sketched graphically in Figure 5. The reader may also verify in this case that the informal composition rules are respected. Notice that the freeze operator is the only basic operator that allows one to build 'feed-backs' in mixins in order to truly eliminate dependences from some input components.

3. A kernel language of mixin modules

In this section we formally define a set of operators for combining mixin modules, which are intended to be a kernel language with clean semantics in which to express operators of existing modular languages. The semantics of the kernel language is given by means

https://doi.org/10.1017/S0960129598002576 Published online by Cambridge University Press

of the formal framework defined in the previous section. More precisely, any module expression M of the language has a type $<\Sigma^{def}, \Sigma^{vir}, \Sigma^{fro}>$, with $\Sigma^{def}, \Sigma^{vir}, \Sigma^{fro}$ signatures in **Sig**, and is interpreted as a mixin model over $<\Sigma^{def} + \Sigma^{vir}, \Sigma^{vir} + \Sigma^{fro}>$. Moreover, the interpretation of the operators of the language is given in terms of the three basic operators over mixin models defined in Section 2.

The three signatures Σ^{def}, Σ^{vir} and Σ^{fro} have the following meaning:

— the components in Σ^{def} are *deferred* components, that is, components that are not defined in the module;
— the components in Σ^{vir} are defined components whose redefinition can change other components (*virtual* components);
— the components in Σ^{fro} are defined components whose redefinition cannot change other components (*frozen* components).

The input signature consists of deferred and virtual components, that is, all the components on which definitions in the module may depend. The output signature consists of virtual and frozen components, that is, all the components that are defined in the module. In the particular case in which $\Sigma^{def} = \varnothing$ (hence $\Sigma^{in} = \Sigma^{vir}$) we get a *concrete* module, where all the components are defined.

Note that the approach taken here is rather abstract: the fact that input components can be decomposed into deferred and virtual components (and analogously output components) is modelled in a descriptive way by pattern matching, that is, by defining the input signature as a coproduct. All the typing rules in the rest of this paper follow an analogous approach.

A possible realization of the splitting of signatures consists of considering *boolean signature categories*, that is, signatures with *inclusions* and related operations of union, intersection and difference. This more concrete case is outlined in Appendix A.1 and extensively presented in Ancona (1998). In the definitions of this section we will use the abbreviations $\Sigma^{in} := \Sigma^{def} + \Sigma^{vir}$, $\Sigma^{out} := \Sigma^{vir} + \Sigma^{fro}$.

Merge The merge operator allows us to combine two mixin modules, say M_1 and M_2, obtaining a new module where some deferred components of M_1 (Σ_1^b) are made concrete by binding them to some defined components in M_2, and *vice versa*. These defined components can be either virtual (Σ_2^{bv}) or frozen (Σ_2^{bf}), and their status is preserved in $M_1 \oplus M_2$. Moreover, some deferred components of M_1 and M_2 can be shared in $M_1 \oplus M_2$ (Σ^s). On the other hand, defined components are kept distinct, as for sum. Hence, in $M_1 \oplus M_2$ the defined components are the (disjoint) union of the components defined in the two arguments; the deferred components are the components deferred in one (but not bound to a definition in the other) or both the arguments. The merge operator is a high-level version of sum, which allows us to combine together mixins with different input components and takes into account the difference between virtual and frozen components.

$$\text{(M-ty)}\quad \frac{\begin{array}{c} M_1 : <\Sigma_1^u + \Sigma^s + \Sigma_1^b, \Sigma_1^{vir}, \Sigma_1^{fro}> \\ M_2 : <\Sigma_2^u + \Sigma^s + \Sigma_2^b, \Sigma_2^{vir}, \Sigma_2^{fro}> \end{array}}{M_1 \oplus M_2 : <\Sigma_1^u + \Sigma^s + \Sigma_2^u, \Sigma_1^{vir} + \Sigma_2^{vir}, \Sigma_1^{fro} + \Sigma_2^{fro}>} \qquad \begin{array}{l} \text{for } i = 1, 2 \\ \Sigma_i^{vir} = \Sigma'^{vir}_i + \Sigma_i^{bv} \\ \Sigma_i^{fro} = \Sigma'^{fro}_i + \Sigma_i^{bf} \\ \Sigma_i^b = \Sigma_i^{bv} + \Sigma_i^{bf}. \end{array}$$

https://doi.org/10.1017/S0960129598002576 Published online by Cambridge University Press

The deferred components of each argument, say M_1, are distinguished in components to be shared (Σ^s), to be bound to an output component of the other mixin (Σ_1^b) and neither shared nor bound (Σ_1^u). The side conditions ensure that each deferred component to be bound (Σ_i^b) has a corresponding output component, either virtual ($\Sigma_{\bar{i}}^{bv}$) or frozen ($\Sigma_{\bar{i}}^{bf}$), in the other mixin (with the notation $\bar{1} = 2, \bar{2} = 1$).

$$\text{(M-sem)} \quad \frac{[\![M_i]\!] = F_i, i = 1,2}{[\![M_1 \oplus M_2]\!] = freeze_j(_{j_1|}F_1 + _{j_2|}F_2)}$$

where $j: \Sigma_1^{bf} + \Sigma_2^{bf} \to \Sigma_1^{out} + \Sigma_2^{out}$, $j_i: \Sigma_i^{in} \to (\Sigma_1^{in} +_{\Sigma_s} \Sigma_2^{in}) + \Sigma_1^{bf} + \Sigma_2^{bf}$, $i = 1,2$, are the obvious injections, with the abbreviations $\Sigma_i^{in} := \Sigma_i^u + \Sigma_s + \Sigma_i^b + \Sigma_i^{vir}$, $i = 1,2$, $\Sigma_1^{in} +_{\Sigma_s} \Sigma_2^{in} := (\Sigma_1^u + \Sigma_s + \Sigma_2^u) + (\Sigma_1^{vir} + \Sigma_2^{vir})$.

Whence

$$_{j_1|}F_1 + _{j_2|}F_2 : Mod((\Sigma_1^{in} +_{\Sigma_s} \Sigma_2^{in}) + (\Sigma_1^{bf} + \Sigma_2^{bf})) \to Mod(\Sigma_1^{out} + \Sigma_2^{out})$$

and

$$freeze_j(_{j_1|}F_1 + _{j_2|}F_2) : Mod(\Sigma_1^{in} +_{\Sigma_s} \Sigma_2^{in}) \to Mod(\Sigma_1^{out} + \Sigma_2^{out}).$$

The semantics of the merge operator is rather simple: first the input signatures of both M_1 and M_2 have to be extended via renaming to the same input signature; then, it is possible to perform the sum; finally, the deferred components bound to frozen components (Σ_1^{bf} and Σ_2^{bf}) have to be frozen in order to preserve their status (see the example below).

With the general approach taken here, when merging two mixins one has to specify explicitly the shared input components (Σ^s), the input components to be bound (Σ_i^b, $i = 1,2$) and the output components they are bound to ($\Sigma_i^{bv}, \Sigma_i^{bf}$, $i = 1,2$). However, in concrete instances of the model one can assume an implicit choice. For instance, if **Sig** is a boolean signature category (see Appendix A.1) and, correspondingly, the type of a mixin is uniquely determined by the input and output signatures ($\Sigma^{def} = \Sigma^{in} \setminus \Sigma^{out}$, $\Sigma^{vir} = \Sigma^{in} \cap \Sigma^{out}$, $\Sigma^{fro} = \Sigma^{out} \setminus \Sigma^{in}$), we can set

$$\begin{aligned} \Sigma^s &= (\Sigma_1^{def} \setminus \Sigma_2^{out}) \cap (\Sigma_2^{def} \setminus \Sigma_1^{out}), \\ \Sigma_i^b &= \Sigma_i^{def} \cap \Sigma_{\bar{i}}^{out}, \quad i = 1,2 \\ \Sigma_{\bar{i}}^{bv} &= \Sigma_i^{def} \cap \Sigma_{\bar{i}}^{vir}, \quad i = 1,2. \end{aligned}$$

with the proviso that $\Sigma_1^{out} \cap \Sigma_2^{out} = \varnothing$ (see typing rule (M-ty) in Figure 11 in the Appendix). We make this assumption in the example below.

Consider a mixin M_1 with two deferred components h and f and one frozen component g, and a mixin M_2 with two deferred components h and g and one virtual component f. If M_1, M_2 denote the mixin models F_1 and F_2, respectively,

$$\begin{aligned} F_1 &= g \mapsto (F_1)_g(h, f), \\ F_2 &= = f \mapsto (F_2)_f(h, g, f), \end{aligned}$$

https://doi.org/10.1017/S0960129598002576 Published online by Cambridge University Press

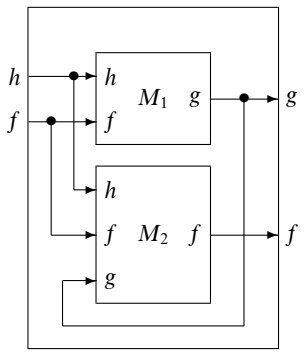


Fig. 6. Merge

then the semantics of $M_1 \oplus M_2$ is

$$g \mapsto F_g(h, f)$$
$$f \mapsto F_f(h, f)$$

where $F(h, f) = freeze(\lambda{<}g, f'{>}.{<}(F_1)_g(h, f), (F_2)_f(h, g, f){>})$.

Correspondingly, we obtain the picture in Figure 6. The example formally matches the typing rule (M-ty) as follows:

$$\frac{M_1{:}{<}\varnothing + \{h\} + \{f\}, \varnothing, \{g\}{>} = {<}\{h, f\}, \varnothing, \{g\}{>} \quad M_2{:}{<}\varnothing + \{h\} + \{g\}, \{f\}, \varnothing{>} = {<}\{h, g\}, \{f\}, \varnothing{>}}{M_1 \oplus M_2{:}{<}\varnothing + \{h\} + \varnothing, \{f\}, \{g\}{>} = {<}\{h\}, \{f\}, \{g\}{>}}$$

Note that the deferred components f of M_1 and g of M_2 are both bound to the corresponding defined components in the other mixin. However, the way in which this binding is achieved is different, since f is bound to a virtual component, whereas g is bound to a frozen component. In the first case, the binding is obtained by sharing the input components f of both the mixins (no need for freeze), while in the second case the input component g of M_2 is bound to the corresponding output component in M_1 by means of the freeze operator. This asymmetry is also captured by the lambda expression denoting $M_1 \oplus M_2$: the function that has to be frozen depends on the first parameter (the g component) and is constant with respect to the second (the f component).

Freeze The freeze operation allows us to make a module independent from the redefinition of some components, say Σ^{fr}; hence these components, which were virtual, become frozen. In this way definitions can be encapsulated to prevent unwanted changes due to some component redefinitions. Note that the converse operation that we have omitted here, simply corresponds to a sort of type coercion in which one loses the type information that some components are frozen, and, hence, that their redefinition does not change the semantics of the other components.

https://doi.org/10.1017/S0960129598002576 Published online by Cambridge University Press

The high-level freeze operator is directly expressed by the low-level *freeze*.

$$\text{(F-ty)}\ \frac{M{:}<\Sigma^{def}, \Sigma^{fr} + \Sigma^{vir}, \Sigma^{fro}>}{\textbf{freeze}\ \Sigma^{fr}\ \textbf{in}\ M{:}<\Sigma^{def}, \Sigma^{vir}, \Sigma^{fr} + \Sigma^{fro}>}$$

$$\text{(F-sem)}\ \frac{[\![M]\!] = F}{[\![\textbf{freeze}\ \Sigma^{fr}\ \textbf{in}\ M]\!] = freeze_j(F)}$$

where $j{:}\Sigma^{fr} \to \Sigma^{fr} + \Sigma^{out}$ is the obvious injection.

Note that the only effect of the freeze operation is to switch the status of defined components from virtual to frozen; deferred and frozen components are not modified.

Restrict The restrict operation allows us to 'cancel' some definitions in a module, making the corresponding components deferred. Hence it makes sense only for virtual components. Note that restrict is different from the hiding described later, since a component whose definition is deferred remains in the interface of the module and can be redefined, while a hidden component is no longer visible from the outside.

$$\text{(RS-ty)}\ \frac{M{:}<\Sigma^{def}, \Sigma^{rs} + \Sigma^{vir}, \Sigma^{fro}>}{\textbf{restrict}\ \Sigma^{rs}\ \textbf{in}\ M{:}<\Sigma^{def} + \Sigma^{rs}, \Sigma^{vir}, \Sigma^{fro}>}$$

$$\text{(RS-sem)}\ \frac{[\![M]\!] = F}{[\![\textbf{restrict}\ \Sigma^{rs}\ \textbf{in}\ M]\!] = F_{|\Sigma^{out}}}$$

Intuitively, the definitions in Σ^{rs} are 'forgotten', and hence the corresponding components are no longer in the output signature (this is formally expressed by the reduct functor). However, they are still in the interface of the module as deferred components (hence the input signature remains the same as before).

Hiding The hiding operation allows us to hide some defined components from the outside. Hiding deferred components makes no sense, since definitions of other components could depend on them and in this way it would be impossible to obtain a concrete module eventually.

$$\text{(H-ty)}\ \frac{M{:}<\Sigma^{def}, \Sigma_v^{hd} + \Sigma^{vir}, \Sigma_f^{hd} + \Sigma^{fro}>}{\textbf{hide}\ <\Sigma_v^{hd}, \Sigma_f^{hd}>\ \textbf{in}\ M{:}<\Sigma^{def}, \Sigma^{vir}, \Sigma^{fro}>}$$

$$\text{(H-sem)}\ \frac{[\![M]\!] = F}{[\![\textbf{hide}\ <\Sigma_v^{hd}, \Sigma_f^{hd}>\ \textbf{in}\ M]\!] = (freeze_j(F))_{|\Sigma^{out}}}$$

where $j{:}\Sigma_v^{hd} \to \Sigma^{out} + \Sigma_v^{hd} + \Sigma_f^{hd}$ is the obvious injection.

Hiding virtual components requires first freezing them in such a way that all the other definitions will refer from now on to their current definitions. Frozen components can be simply thrown away by restricting the output signature.

https://doi.org/10.1017/S0960129598002576 Published online by Cambridge University Press

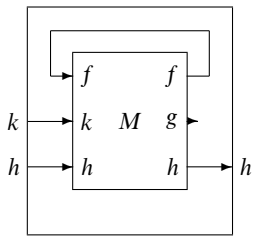


Fig. 7. Hiding

	local	*frozen*	*deferred*	*virtual*
local	-	no	no	no
frozen	hiding	-	no	no
deferred	no	merge	-	merge
virtual	hiding	freeze	restrict	-

Table 1. *Status transformations*

As an example, consider a mixin M with one deferred component k, two virtual components f and h and one frozen component g. If M denotes the mixin model F:

$$\begin{aligned} F = f &\mapsto F_f(f,k,h) \\ g &\mapsto F_g(f,k,h) \\ h &\mapsto F_h(f,k,h). \end{aligned}$$

then the semantics of **hide** $\{f,g\}$ **in** M is

$$h \mapsto F'_h(k,h)$$

where $F'(k,h) = \mathit{freeze}(\lambda{<}f,k',h'{>}.F(f,k',h'))$.

Correspondingly, we obtain the picture in Figure 7.

Up to this point we have presented a set of operators for mixin combination. As already stated, each mixin component has a status; if we also consider locally defined components (components not present in the mixin signature), the possible statuses are given by *local, frozen, deferred* and *virtual*. Each mixin operator has a different effect upon the component status, as illustrated in Table 1. The first column and row contain the initial and final status of a component, respectively. Note that not all status changes are allowed.

Since a local component is not visible, there is no way to change its status. A frozen component can only become local by means of hiding. A deferred component can be transformed into a frozen (respectively, virtual) component by merging it with a frozen (respectively, virtual) component. A virtual component can become local, frozen or deferred by means of hiding, freeze or restrict, respectively.

https://doi.org/10.1017/S0960129598002576 Published online by Cambridge University Press

We will now consider some more sophisticated operators, corresponding to constructs effectively used in programming languages.

Overriding The overriding operator is a non-commutative variant of the merge operator allowing us to ignore some output component (either virtual or frozen) of one of the two arguments. This is useful in practice when two mixins have some output components in common (conflicting definitions) and we want the definitions of one mixin to take precedence over the definitions of the other.

$$\text{(O-ty)}\ \frac{\begin{array}{l} M_1{:}<\Sigma_1^u+\Sigma^s+\Sigma_1^b,\Sigma_1^{vir}+\Sigma_v^{rs},\Sigma_1^{fro}+\Sigma_f^{rs}> \\ M_2{:}<\Sigma_2^u+\Sigma^s+\Sigma_2^b,\Sigma_2^{vir},\Sigma_2^{fro}> \end{array}}{M_1{\leftarrow}M_2{:}<\Sigma_1^u+\Sigma^s+\Sigma_2^u,\Sigma_1^{vir}+\Sigma_2^{vir},\Sigma_1^{fro}+\Sigma_2^{fro}>}\ \begin{array}{l} \Sigma_i^{vir}=\Sigma'^{vir}_i+\Sigma_i^{bv}, i=1,2 \\ \Sigma_i^{fro}=\Sigma'^{fro}_i+\Sigma_i^{bf}, i=1,2 \\ \Sigma_1^b+\Sigma_v^{rs}=\Sigma_2^{bv}+\Sigma_2^{bf} \\ \Sigma_2^b=\Sigma_1^{bv}+\Sigma_1^{bf} \end{array}$$

$$\text{(O-sem)}\ \frac{[\![M_i]\!]=F_i, i=1,2}{[\![M_1{\leftarrow}M_2]\!]=\mathit{freeze}_j({}_{j_1|}F_{1|\Sigma_1^{out}}+{}_{j_2|}F_2)}$$

where

$$\begin{array}{l} j{:}\Sigma_1^{bf}+\Sigma_2^{bf}\rightarrow\Sigma_1^{out}+\Sigma_2^{out}, \\ j_1{:}\Sigma_1^{in}+\Sigma_v^{rs}\rightarrow(\Sigma_1^{in}+_{\Sigma_s}\Sigma_2^{in})+(\Sigma_1^{bf}+\Sigma_2^{bf}), \\ j_2{:}\Sigma_2^{in}\rightarrow(\Sigma_1^{in}+_{\Sigma_s}\Sigma_2^{in})+(\Sigma_1^{bf}+\Sigma_2^{bf}), \end{array}$$

are the obvious injections, with the abbreviations

$$\begin{array}{rl} \Sigma_i^{in} & :=\Sigma_i^u+\Sigma^s+\Sigma_i^b+\Sigma_i^{vir}, i=1,2, \\ \Sigma_1^{in}+_{\Sigma_s}\Sigma_2^{in} & :=(\Sigma_1^u+\Sigma_s+\Sigma_2^u)+(\Sigma_1^{vir}+\Sigma_2^{vir}). \end{array}$$

Whence

$${}_{j_1|}F_1+{}_{j_2|}F_2{:}\mathit{Mod}((\Sigma_1^{in}+_{\Sigma_s}\Sigma_2^{in})+(\Sigma_1^{bf}+\Sigma_2^{bf}))\rightarrow\mathit{Mod}(\Sigma_1^{out}+\Sigma_2^{out})$$

and

$$\mathit{freeze}_j({}_{j_1|}F_{1|\Sigma_1^{out}}+{}_{j_2|}F_2){:}\mathit{Mod}(\Sigma_1^{in}+_{\Sigma_s}\Sigma_2^{in})\rightarrow\mathit{Mod}(\Sigma_1^{out}+\Sigma_2^{out}).$$

The typing rule for the overriding operator is similar to that of merge; indeed, $M_1{\leftarrow}M_2$ is obtained by first restricting the output components of M_1 (cancelling the virtual and the frozen components in Σ_v^{rs} and Σ_f^{rs}, respectively) and, then, by merging the resulting mixin with M_2. Indeed, the following fact holds.

Fact 3.1. For any pair of mixin expressions M_1 and M_2 verifying the typing rule (O-ty),

$$[\![M_1{\leftarrow}M_2]\!] = [\![M_1]\!]_{|\Sigma_1^{out}} \oplus [\![M_2]\!].$$

https://doi.org/10.1017/S0960129598002576 Published online by Cambridge University Press

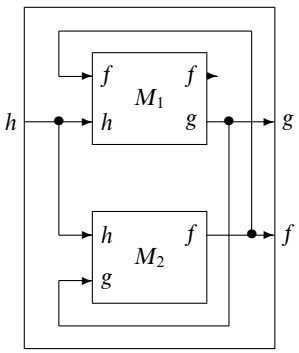


Fig. 8. Overriding

Again, in the case of boolean signature categories, we can set (see typing rule (O-ty) in Figure 11 in Appendix)

$$\begin{aligned}
\Sigma^s &= (\Sigma_1^{def} \setminus \Sigma_2^{out}) \cap (\Sigma_2^{def} \setminus \Sigma_1^{out}) \\
\Sigma_i^b &= \Sigma_i^{def} \cap \Sigma_{\bar{i}}^{out}, \quad i = 1, 2 \\
\Sigma_1^{bv} &= \Sigma_2^{def} \cap (\Sigma_1^{vir} \setminus \Sigma_2^{out}), \\
\Sigma_2^{bv} &= (\Sigma_1^{def} \cup \Sigma_v^{rs}) \cap \Sigma_2^{vir} \\
\Sigma_v^{rs} &= \Sigma_1^{vir} \cap \Sigma_2^{out}, \\
\Sigma_f^{rs} &= \Sigma_1^{fro} \cap \Sigma_2^{out}.
\end{aligned}$$

As an example, consider a mixin M_1 with one deferred component h, one virtual component f and one frozen component g, and a mixin M_2 with two deferred components h and g, and one frozen component f. If M_1, M_2 denote the mixin models F_1 and F_2, respectively:

$$\begin{aligned}
F_1 &= f \mapsto (F_1)_f(f, h), \\
&\quad\ g \mapsto (F_1)_g(f, h)
\end{aligned}$$

$$F_2 = f \mapsto (F_2)_f(h, g),$$

then the semantics of $M_1 \leftarrow M_2$ is

$$\begin{aligned}
f &\mapsto F_f(h) \\
g &\mapsto F_g(h)
\end{aligned}$$

where

$$F(h) = freeze(\lambda <f', g, f>.<(F_1)_f(f, h), (F_1)_g(f, h), (F_2)_f(h, g)>).$$

Correspondingly, we obtain the picture in Figure 8. The example formally matches the

https://doi.org/10.1017/S0960129598002576 Published online by Cambridge University Press

typing rule (O-ty) as follows:

$$\frac{M_1{:}<\varnothing+\{h\}+\varnothing,\varnothing+\{f\},\{g\}+\varnothing>=<\{h\},\{f\},\{g\}> \quad M_2{:}<\varnothing+\{h\}+\{g\},\varnothing,\{f\}>=<\{h,g\},\varnothing,\{f\}>}{M_1{\leftarrow}M_2{:}<\varnothing+\{h\}+\varnothing,\varnothing,\{f,g\}>=<\{h\},\varnothing,\{f,g\}>}$$

Functional Composition Since mixins are modelled as functions, a natural way for combining two mixins M_1 and M_2 is functional composition $M_2{\circ}M_1$ (provided that types are compatible). From the graphical point of view, this operation gives sequential composition of mixins, in opposition to parallel composition given by the merge operation.

Whereas the merge operator allows us to combine together mixins that are mutually dependent (that is, where some deferred components of one mixin are bound to some output components of the other, and *vice versa*), in functional composition the dependence is only one-way; in this case the mixins can be considered as generic modules (for example, ML functors) and the functional composition operator corresponds to (a generalization of) generic instantiation (for example, functor application in ML).

The generalization is given by the fact that the parameter can be in turn parameterized (that is, have deferred components), as would happen if ML allowed functor composition.

$$\text{(FC-ty)}\ \frac{M_i{:}<\Sigma_i^{def},\Sigma_i^{vir},\Sigma_i^{fro}>, i=1,2}{M_2{\circ}M_1{:}<\Sigma_1^{def},\Sigma_2^{vir},\Sigma_2^{fro}>}\ \Sigma_2^{def}=\Sigma_1^{out}$$

$$\text{(FC-sem)}\ \frac{[\![\, M_i \,]\!]=F_i, i=1,2}{[\![\, M_2{\circ}M_1 \,]\!]=(freeze_j(_{j_1|}F_1+_{j_2|}F_2))_{|\Sigma_2^{out}}}$$

where $j{:}\Sigma_1^{out}\to\Sigma_1^{out}+\Sigma_2^{out}$, $j_i{:}\Sigma_i^{in}\to\Sigma_1^{def}+\Sigma_1^{out}+\Sigma_2^{vir}$, $i=1,2$, are the obvious injections.

The side condition in the typing rule requires that the deferred components of M_2 are exactly the output components of M_1. We could relax this condition by simply requiring that each deferred component of M_2 is bound to some output component of M_1, but the definition would become more complex, without any gain of expressive power.

The deferred components of $M_2{\circ}M_1$ are the deferred components of M_1; the defined components are those of M_2, with the same status; the defined components of M_1 (deferred components of M_2) are not present in $M_2{\circ}M_1$; in particular, the virtual components of M_1 must be frozen in order to eliminate them from the resulting input signature.

Note that $M_2{\circ}M_1=$ **hide** $<\Sigma_1^{vir},\Sigma_1^{fro}>$ **in** $(M_1\oplus M_2)$ where the merge $M_1\oplus M_2$ corresponds to the particular case where (*cf.* rule M-ty):

— $\Sigma^s=\varnothing$ (no sharing of deferred components);
— $\Sigma_1^{bv}=\Sigma_1^{vir},\Sigma_1^{bf}=\Sigma_1^{fro}$, therefore $\Sigma_2^b=\Sigma_1^{vir}+\Sigma_1^{fro}=\Sigma_2^{def}$ (all the deferred components of F_2 are bound);
— $\Sigma_2^{bv},\Sigma_2^{bf}=\varnothing$, therefore $\Sigma_1^b=\varnothing$ (all the deferred components of F_1 are unbound).

Note, finally, that, as a consequence, $\Sigma_1^u=\Sigma_1^{def}$ and $\Sigma_2^u=\varnothing$.

As an example, consider a mixin M_1 with just one virtual component g, and a mixin M_2 with one deferred component g and one virtual component f. If M_1, M_2 denote the

https://doi.org/10.1017/S0960129598002576 Published online by Cambridge University Press

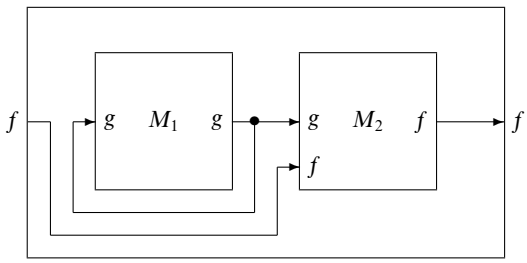


Fig. 9. Functional Composition

mixin models F_1 and F_2, respectively:

$$F_1 = g \mapsto (F_1)_g(g),$$
$$F_2 = f \mapsto (F_2)_f(f, g),$$

then their functional composition is

$$f \mapsto F_f(f),$$

where $F(f) = \mathit{freeze}(\lambda{<}g, f'{>}.{<}(F_1)_g(g), (F_2)_f(f, g){>})$.

Correspondingly, we obtain the picture in Figure 9.

The example formally matches the typing rule (FC-ty) as follows:

$$\frac{M_1{:}{<}\varnothing, \{g\}, \varnothing{>} \quad M_2{:}{<}\{g\}, \{f\}, \varnothing{>}}{M_1 \circ M_2{:}{<}\varnothing, \{f\}, \varnothing{>}}$$

Referring to Overridden Components We now show how to express by means of our basic operators over mixin modules an inheritance mechanism like that of Smalltalk or Beta. This topic is analyzed in detail in Ancona and Zucca (1997); here we just give an outline.

In Smalltalk, subclasses can override methods of parent classes (that is, subclasses take precedence over parent classes), and subclass methods can invoke original superclass methods via *super*.

By contrast, in Beta, subclasses (called *subpatterns*) cannot override methods of parent classes, called *prefix patterns*, (that is, parent classes take precedence over subclasses), and superclass methods can invoke subclass methods via *inner*. As pointed out in Bracha and Cook (1990), these two mechanisms are two different uses of a single underlying construct; the only difference is the direction of class hierarchy growth.

This underlying construct can be modelled in our context by an asymmetric binary operator, which we call *weak* overriding and denote by $\lhd\!\!-$, such that, in $M_1 \lhd\!\!- M_2$, definitions in M_2 take precedence over definitions in M_1, as with the overriding operator, but overridden definitions in M_1 remain significant, since definitions in M_2 may refer to them via *other* (we use this keyword to point out that the mechanism is the same whichever inheritance relation is used between M_1 and M_2).

https://doi.org/10.1017/S0960129598002576 Published online by Cambridge University Press

Formally, we assume that the type of a mixin module is a 4-tuple

$$<\Sigma^{oth}, \Sigma^{def}, \Sigma^{vir}, \Sigma^{fro}>;$$

the corresponding mixin signature will be

$$<\Sigma^{oth} + \Sigma^{def} + \Sigma^{vir}, \Sigma^{vir} + \Sigma^{fro}>.$$

This models the fact that a module component, say f, appears on the right-hand side of definitions either in the usual way (written f or *self* f, that is, the symbols in $\Sigma^{def} + \Sigma^{vir}$) or under the form *other* f (that is, the symbols in Σ^{oth}, which we will call the *other-components*). Whenever f is defined, the first form is a (possibly recursive) reference to its current definition, while the second form refers to a (deferred) alternative definition of f, which will be provided when composing the module with another by means of the $\lhd\!\leftarrow$ operator (hence, *other* f is always a deferred component). In the case in which f is not defined, both the forms refer to a definition to be provided by the outside, though they are still different with respect to their behaviour after that definition has actually been provided. Indeed, *other* f is permanently associated with this definition, while f could be redefined later.

$$\text{(WO-ty)}\quad \frac{\begin{array}{c} M_1 :< \Sigma_1^{oth} + \Sigma_s^{oth}, \Sigma_1^u + \Sigma^s + \Sigma_1^b, \Sigma_1^{vir} + \Sigma_v^{rs}, \Sigma_1^{fro} + \Sigma_f^{rs} > \\ M_2 :< \Sigma_2^{oth} + \Sigma_s^{oth} + \Sigma_b^{oth}, \Sigma_2^u + \Sigma^s + \Sigma_2^b, \Sigma_2^{vir}, \Sigma_2^{fro} > \end{array}}{M_1 \lhd\!\leftarrow M_2 :< \Sigma_1^{oth} + \Sigma_s^{oth} + \Sigma_2^{oth}, \Sigma_1^u + \Sigma^s + \Sigma_2^u, \Sigma_1^{vir} + \Sigma_2^{vir}, \Sigma_1^{fro} + \Sigma_2^{fro} >}$$

where

$$\begin{aligned} \Sigma_i^{vir} &= \Sigma_i'^{vir} + \Sigma_i^{bv}, \quad i = 1,2 \\ \Sigma_i^{fro} &= \Sigma_i'^{fro} + \Sigma_i^{bf}, \quad i = 1,2 \\ \Sigma_1^b + \Sigma_v^{rs} &= \Sigma_2^{bv} + \Sigma_2^{bf} \\ \Sigma_2^b &= \Sigma_1^{bv} + \Sigma_1^{bf} \\ \Sigma_b^{oth} &= \Sigma_v^{rs} + \Sigma_f^{rs} \end{aligned}$$

$$\text{(WO-sem)}\quad \frac{[\![M_i]\!] = F_i, i = 1,2}{[\![M_1 \lhd\!\leftarrow M_2]\!] = (freeze_j(_{j_1|}F_1 +_{j_2|} F_2))_{|\Sigma_1^{out} + \Sigma_2^{out}}}$$

where

$$\begin{aligned} &j : \Sigma_b^{oth} + \Sigma_1^{bf} \to +\Sigma_2^{bf} \Sigma_1^{out} + \Sigma_2^{out} + \Sigma_v^{rs} + \Sigma_f^{rs}, \\ &j_1 : \Sigma_1^{oth} + \Sigma_s^{oth} + \Sigma_i^{in} + \Sigma_v^{rs} \to \\ &\qquad (\Sigma_1^{oth} +_{\Sigma_s^{oth}} \Sigma_2^{oth}) + \Sigma_b^{oth} + (\Sigma_1^{in} +_{\Sigma_s} \Sigma_2^{in}) + (\Sigma_1^{bf} + \Sigma_2^{bf}), \\ &\mathrm{J}_2 : \Sigma_2^{oth} + \Sigma_s^{oth} + \Sigma_b^{oth} + \Sigma_i^{in} \to \\ &\qquad (\Sigma_1^{oth} +_{\Sigma_s^{oth}} \Sigma_2^{oth}) + \Sigma_b^{oth} + (\Sigma_1^{in} +_{\Sigma_s} \Sigma_2^{in}) + (\Sigma_1^{bf} + \Sigma_2^{bf}), \end{aligned}$$

are the obvious injections, with the abbreviations

$$\begin{aligned} \Sigma_i^{in} &:= \Sigma_i^u + \Sigma_s + \Sigma_i^b + \Sigma_i^{vir}, \quad i = 1,2, \\ \Sigma_1^{in} +_{\Sigma_s} \Sigma_2^{in} &:= (\Sigma_1^u + \Sigma_s + \Sigma_2^u) + (\Sigma_1^{vir} + \Sigma_2^{vir}), \\ \Sigma_1^{oth} +_{\Sigma_s^{oth}} \Sigma_2^{oth} &:= \Sigma_1^{oth} + \Sigma_s^{oth} + \Sigma_2^{oth}. \end{aligned}$$

https://doi.org/10.1017/S0960129598002576 Published online by Cambridge University Press

Whence

$$_{j_1|}F_1 + _{j_2|}F_2 : Mod((\Sigma_1^{oth} +_{\Sigma_s^{oth}} \Sigma_2^{oth}) + \Sigma_b^{oth} + (\Sigma_1^{in} +_{\Sigma_s} \Sigma_2^{in}) + (\Sigma_1^{bf} + \Sigma_2^{bf})) \rightarrow Mod(\Sigma_1^{out} + \Sigma_v^{rs} + \Sigma_f^{rs} + \Sigma_2^{out})$$

and

$$freeze_j(_{j_1|}F_{1|\Sigma_1^{out}} + _{j_2|}F_2) : Mod((\Sigma_1^{oth} +_{\Sigma_s^{oth}} \Sigma_2^{oth}) + (\Sigma_1^{in} +_{\Sigma_s} \Sigma_2^{in})) \rightarrow Mod(\Sigma_1^{out} + \Sigma_v^{rs} + \Sigma_f^{rs} + \Sigma_2^{out})$$

and

$$freeze_j(_{j_1|}F_{1|\Sigma_1^{out}} + _{j_2|}F_2)_{|\Sigma_1^{out}+\Sigma_2^{out}} : Mod((\Sigma_1^{oth} +_{\Sigma_s^{oth}} \Sigma_2^{oth}) + (\Sigma_1^{in} +_{\Sigma_s} \Sigma_2^{in})) \rightarrow Mod(\Sigma_1^{out} + \Sigma_2^{out}).$$

The same considerations made for overriding (rules (O-ty) and (O-sem)) also apply in this case. Moreover, one has to deal with the other-components. The signature Σ_s^{oth} represents the other-components to be shared (consider, for example, two Smalltalk classes, parent and heir, referring via *super* to a component defined in a common ancestor). The other-components in Σ_b^{oth} are those to be bound – note that, by definition, only the other-components of M_2 can be bound, and that they have to be bound to the overridden components of M_1 (represented by $\Sigma_v^{rs} + \Sigma_f^{rs}$). The remaining other-components are neither shared nor bound.

As an example, consider a mixin M_1 with one other-component $o.f$, one virtual component g and one frozen component f, and a mixin M_2 with two other-components $o.f$ and $o.g$, and one frozen component g. If M_1, M_2 denote the mixin models F_1 and F_2, respectively:

$$\begin{aligned} F_1 &= g \mapsto (F_1)_g(g, o.f), \\ &\quad\ f \mapsto (F_1)_f(g, o.f) \\ F_2 &= g \mapsto (F_2)_g(o.g, o.f), \end{aligned}$$

then the semantics of $M_1 \lhd\!\!\leftarrow M_2$ is

$$\begin{aligned} f &\mapsto F_f(o.f) \\ g &\mapsto F_g(o.f) \end{aligned}$$

where $F(o.f) = freeze(\lambda <g', f, g>.<(F_1)_g(g, o.f), (F_1)_f(g, o.f), (F_2)_g(g', o.f)>)$.

Correspondingly, we obtain the picture in Figure 10.

The example formally matches the typing rule (WO-ty) as follows:

$$\frac{\begin{array}{c} M_1 : <\emptyset + \{o.f\}, \emptyset + \emptyset + \emptyset, \emptyset + \{g\}, \{f\} + \emptyset> = <\{o.f\}, \emptyset, \{g\}, \{f\}> \\ M_2 : <\emptyset + \{o.f\} + \{o.g\}, \emptyset + \emptyset + \emptyset, \emptyset, \{g\}> = <\{o.f, o.g\}, \emptyset, \emptyset, \{g\}> \end{array}}{M_1 \lhd\!\!\leftarrow M_2 : <\emptyset + \{o.f\} + \emptyset, \emptyset + \emptyset + \emptyset, \emptyset + \emptyset, \{f\} + \{g\}> = <\{o.f\}, \emptyset, \emptyset, \{f, g\}>}$$

As mentioned above, the interested reader can find in Ancona and Zucca (1997) a presentation devoted to the treatment of different overriding operators in the mixin framework described in this paper.

https://doi.org/10.1017/S0960129598002576 Published online by Cambridge University Press

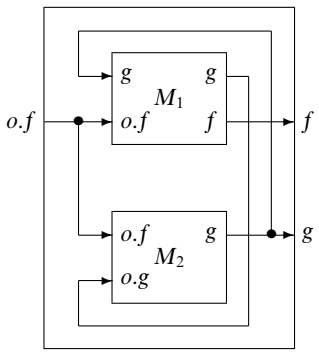


Fig. 10. Inheritance

4. Dealing with types

In this section we show that the formal framework we have presented up to this point needs to be refined in order to handle type definitions in modules properly.

First, we consider mixin signatures. We recall that a mixin signature has been formalized by a pair of signatures $<\Sigma^{in}, \Sigma^{out}>$ where $\Sigma^{in} = \Sigma^{def} + \Sigma^{vir}$, $\Sigma^{out} = \Sigma^{vir} + \Sigma^{fro}$, with $\Sigma^{def}, \Sigma^{vir}, \Sigma^{fro}$ signatures corresponding to the deferred, frozen and virtual components, respectively. This expresses in an abstract way the fact that the only relation between the input and the output signatures is given by the virtual components, which are shared – in particular, if there are no virtual components, Σ^{in} and Σ^{out} can be considered as two independent signatures.

Let us now return to the example of sets and maps of Section 1.1. Considering the mixin signature `SET`, it is easy to see that, as already pointed out earlier, this signature cannot be divided into two independent subsignatures corresponding to deferred and defined components, respectively. This is because the functionality of deferred components may contain defined types, and *vice versa*. Hence, we have to consider signatures in which there is an explicit notion of *sorts* modelling type components. In this specialized framework a mixin signature can be defined as a 4-tuple $<S^{in}, \Sigma^{in}, S^{out}, \Sigma^{out}>$ where S^{in} and S^{out} are two disjoint sets of symbols corresponding to deferred and defined type components, respectively, and Σ^{in}, Σ^{out} are two signatures over $S^{in} \cup S^{out}$, which have the same meaning as before.

The assumption that S^{in} and S^{out} are disjoint sets models the fact that type components cannot be virtual. Indeed, referring again to the example of Section 1.1, considering, for example, the mixin `Set`, we can see that it makes sense to replace the definition of either the `is_in` or the `restrict` function by a new definition, but not the definition of the type `set` – indeed the well-formedness of the definitions of `is_in` and `restrict` relies on the given implementation of this type. Hence, type components in signatures must be distinguished explicitly from other components for a second reason also – that they are

https://doi.org/10.1017/S0960129598002576 Published online by Cambridge University Press

components that cannot be redefined, since it is not true that other definitions make sense for (formally, are parameterized by) any possible definition of them†.

Correspondingly, at the semantic level, we need to refine our notion of a mixin model as a pair $<H, F>$ where:

— H is a function giving the semantic counterpart of type definitions in a module, in terms of deferred types. For instance, in the mixin `Set`, this function gives an interpretation of the type `set` for each given interpretation of the type `map`. In this example H is a constant function, since the definition of the type `set` does not depend on `map`, but in general type definitions in a mixin could depend on deferred types in a mutually recursive way. Think, for example, of two mixin modules M_1 and M_2 whose type components are as follows:

M_1
```
type forest deferred
datatype tree = pair of int * forest
```

M_2
```
datatype forest = empty | add of tree * forest
type tree deferred
```

We assume that the denotation of a type component is the set of its possible values, and hence H is more precisely a function:

$$H : SSet(S^{in}) \rightarrow SSet(S^{out})$$

where $SSet(S)$ denotes the set of S-sorted sets (see Definition 4.3 below). Moreover, H must be continuous (in the usual sense, see A.2 in the Appendix); indeed, referring to the above example, when we merge M_1 and M_2, we get a mixin $M_1 + M_2$ with no deferred type components ($S^{in} = \emptyset$) and two defined type components

$M_1 + M_2$
```
datatype forest = empty | add of tree * forest
datatype tree = pair of int * forest
```

whose interpretation is the least fixed point of the above recursive definition.

— F is a family of (total) functions indexed over possible interpretations Y of deferred types (formally, $Y \in SSet(S^{in})$), such that, for any Y, F_Y gives the semantic counterpart of other (non-type) definitions in a module, in terms of input components, for a fixed interpretation of deferred (Y) and defined ($H(Y)$) types.
Formally, denoting by $Mod_X(\Sigma)$ the class of the Σ-models with carrier X (that is, where the interpretation of sorts is fixed to be as in X),

$$F_Y : Mod_{Y+H(Y)}(\Sigma^{in}) \rightarrow Mod_{Y+H(Y)}(\Sigma^{out})$$

For instance, in the mixin `Set`, any Y gives an interpretation (set of values) for the type `map`, and, correspondingly, F_Y gives an interpretation of the functions `is_in` and `restrict` for each given interpretation of the functions `empty_map`, `apply`, `update` and `def_dom`; the domains and codomains of all these functions are fixed for each index Y.

† A less drastic solution, which we leave for further work, would be to allow the simultaneous redefinition of an abstract type together with all its primitives.

https://doi.org/10.1017/S0960129598002576 Published online by Cambridge University Press

The formal definitions follow.

4.1. *Sorted core frameworks*

We define a sorted signature category as a particular category of signatures where a signature has a set of *sorts*, and, conversely, a set of sorts can be seen as a particular (empty) signature.

Definition 4.1. A *sorted* signature category is a triple $<\mathbf{Sig}, Sorts, \varnothing_{\_}>$, where **Sig** is a signature category and $Sorts : \mathbf{Sig} \to \mathbf{Set}$, $\varnothing_{\_} : \mathbf{Set} \to \mathbf{Sig}$ are functors such that

— *Sorts* is a right adjoint of $\varnothing_{\_}$ with the identity as unit of the adjunction; we use φ to denote the isomorphism $\varphi : Hom_{\mathbf{Set}}(<S, Sorts(\Sigma)>) \cong Hom_{\mathbf{Sig}}(<\varnothing_S, \Sigma>)$;
— for each pair of signatures Σ_1 and Σ_2 with $Sorts(\Sigma_i) = S_i$, $i = 1, 2$, there exists a pushout Σ (denoted by $\Sigma_1 \oplus \Sigma_2$) of $\Sigma_1 \xleftarrow{f_1} \varnothing_{S_1 \cap S_2} \xrightarrow{f_2} \Sigma_2$ such that $Sorts(\Sigma) = S_1 \cup S_2$, where $f_i = \varphi(\epsilon_i)$, with ϵ_i the inclusion from $S_1 \cap S_2$ into S_i, $i = 1, 2$.

If $\Sigma \in \mathbf{Sig}$ with $Sorts(\Sigma) = S$, we say that Σ is *over* the set of *sorts* S; $\varnothing_S$ is called the *empty* signature over S.

Example 4.2. The category of algebraic many-sorted signatures is a sorted signature category. The functor *Sorts* returns for any signature its set of sorts, and for any morphism its component over sorts, whereas for any set of sorts S, $\varnothing_S$ is the signature over S having no operation symbols, and for any function $f : S_1 \to S_2$, $\varnothing_f$ is the unique signature morphism from $\varnothing_{S_1}$ to $\varnothing_{S_2}$ having f as component over sorts.

Corresponding to the fact that signatures have a set of sorts, we consider model parts where models are (sorted) sets enriched by some structure, such as operations in the case of standard algebras. We call them *concrete* model parts, since the category of Σ-models, for any signature Σ, is a *concrete model category* (Bidoit and Tarlecki 1996), that is, a particular case of a *concrete category* (Adamek *et al.* 1990); the same notion has been called a *static framework* in a context where the aim was to enrich such a framework by dynamic features (Ancona and Zucca 1996; Zucca 1996).

Definition 4.3. The functor $SSet : \mathbf{Set}^{op} \to \mathbf{Set}$ is defined as follows:

— for any set S, the elements of $SSet(S)$ are the S-sorted sets;
— for any map $\sigma : S_1 \to S_2$ and any S_2-sorted set A, $(A_{|\sigma})_s = A_{\sigma(s)}, \forall\, s \in S_1$.

Note that *SSet* is, in fact, the algebraic model functor to the algebraic signature $<S, \varnothing>$, which is known to preserve colimits. Therefore, for any X_1 in $SSet(S_1)$ and X_2 in $SSet(S_2)$, there exists the amalgamated sum $X_1 + X_2$ in $SSet(S_1 + S_2)$.

Definition 4.4.
A *concrete model part* is a 4-tuple *suchthatatFrameTuple* where

— $<\mathbf{Sig}, Sorts>$ is a sorted signature category;
— $<\mathbf{Sig}, Mod>$ is a model part;
— $|-|$ is a natural transformation, $|-| : Mod \stackrel{\cdot}{\to} SSet \circ Sorts^{op}$, such that, for any $S \in \mathbf{Set}$, $|-|_{\varnothing_S}$ is an embedding. For any Σ-algebra A, $|A|_\Sigma$ is called the *carrier* of A, and denoted by $|A|$, or just A when there is no ambiguity.

https://doi.org/10.1017/S0960129598002576 Published online by Cambridge University Press

The assumption that $|-|_{\varnothing_S}$ is an embedding ensures that models over an empty signature are essentially sorted sets.

Definition 4.5. A concrete model part is *regular* iff $<\mathbf{Sig}, Mod>$ is a regular model part.

Definition 4.6. A *sorted core framework* is a tuple $<\mathbf{Sig}, Sorts, Mod, |-|, freeze>$ where

— $<\mathbf{Sig}, Sorts, Mod, |-|>$ is a regular concrete model part;
— *freeze* is a family of functions

$$freeze_{X,\Sigma} : (Mod_X(\Sigma) \to Mod_X(\Sigma)) \to Mod_X(\Sigma)$$

indexed over pairs $<X, \Sigma>$ with Σ signature over S, $X \in SSet(S)$, where $Mod_X(\Sigma)$ denotes the class of Σ-models A such that $|A| = X$.

Example 4.7. Many sorted partial algebras form a sorted core framework. Indeed, partial algebras are pairs consisting of a sorted set together with the interpretation of the operation symbols, and hence the carrier is simply obtained by taking the first component. Then, a function $F : Mod_X(\Sigma) \to Mod_X(\Sigma)$ transforms tuples of partial functions into tuples of partial functions, and $freeze_{X,\Sigma}$ is the least fixed point of F, whenever F is continuous. Since in this case mixin models correspond with continuous functions F, we can disregard the behaviour of *freeze* over non-continuous functions.

4.2. *Sorted mixin signatures and models*

We now give the refined formal definition of mixin signatures and models. In this subsection, we assume a fixed sorted core framework $<\mathbf{Sig}, Sorts, Mod, |-|, freeze>$.

Definition 4.8. A *sorted mixin signature* is a 4-tuple $<S^{in}, \Sigma^{in}, S^{out}, \Sigma^{out}>$, where $S^{in} \cap S^{out} = \varnothing$ and Σ^{in}, Σ^{out} are two signatures over $S^{in} \cup S^{out}$. We call Σ^{in} and Σ^{out} the *input* and the *output* signatures, respectively, and S^{in} and S^{out} the set of *input* and *output* sorts, respectively.

Intuitively, a sorted mixin signature models the syntactic interface of a mixin module with type components. The input and output sorts model deferred and defined type components, respectively. Since defined components may involve deferred types, and conversely, Σ^{in} and Σ^{out} are signatures over the full set of sorts.

Definition 4.9. A *sorted mixin model* over a mixin signature $<S^{in}, \Sigma^{in}, S^{out}, \Sigma^{out}>$ is a pair $<H, F>$, where

— $H : SSet(S^{in}) \to SSet(S^{out})$ is a continuous function (see A.2 in the Appendix);
— F is a family of functions,

$$F_Y : Mod_{Y+H(Y)}(\Sigma^{in}) \to Mod_{Y+H(Y)}(\Sigma^{out}),$$

indexed over $Y \in SSet(S^{in})$.

Intuitively, the first component is the semantic counterpart of the type definitions in a module: these definitions give a semantic value to defined types (formally, a sorted set over S^{out}), once a semantic value has been provided for deferred types (formally, a sorted set over S^{in}). The second component is the semantic counterpart of definitions of other components (for example, functions): these definitions give a semantic value to the output

https://doi.org/10.1017/S0960129598002576 Published online by Cambridge University Press

components (formally, a model over Σ^{out}), once a semantic value has been provided for input components (formally, a model over Σ^{in}). The semantic value of type components is the same (the amalgamated sum $Y + H(Y)$) both in the argument and in the result model.

We show now that sorted mixin signatures and models constitute a model part in the sense of Definition 2.1.

Definition 4.10. A morphism from $<S^{in}, \Sigma^{in}, S^{out}, \Sigma^{out}>$ to $<S'^{in}, \Sigma'^{in}, S'^{out}, \Sigma'^{out}>$ is a pair

$$<\sigma^{in} : \Sigma^{in} \to \Sigma'^{in}, \sigma^{out} : \Sigma'^{out} \to \Sigma^{out}>,$$

where σ^{in}, σ^{out} are signature morphisms such that $Sorts(\sigma^{in})_{|S^{in}}$ and $Sorts(\sigma^{out})_{|S'^{out}}$ turns out to be functions from S^{in} to S'^{in} and from S'^{out} to S^{out}, respectively (henceforth denoted by f^{in} and f^{out}, respectively), verifying the following properties:

$$(id_{S'^{in}} + f^{out}) \circ Sorts(\sigma^{in}) = f^{in} + id_{S^{out}}$$
$$(f^{in} + id_{S^{out}}) \circ Sorts(\sigma^{out}) = id_{S'^{in}} + f^{out}$$

It is not hard to prove that if we define morphism composition component-wise, then we obtain a category, which we denote by **SortMixSig**.

Note that the conditions

$$(id_{S'^{in}} + f^{out}) \circ Sorts(\sigma^{in}) = f^{in} + id_{S^{out}}$$
$$(f^{in} + id_{S^{out}}) \circ Sorts(\sigma^{out}) = id_{S'^{in}} + f^{out}$$

on sorted signature morphisms ensure that f^{in} and f^{out} are surjective and $Sorts(\sigma^{in})_{|S^{out}}$ and $Sorts(\sigma^{out})_{|S'^{in}}$ are injective. This is needed to make *SortMixMod* a functor (see Definition 4.12 below).

Lemma 4.11. Let $\sigma^{in} : \Sigma^{in} \to \Sigma'^{in}$, $\sigma^{out} : \Sigma'^{out} \to \Sigma^{out}$ be signature morphisms. Then, we can correctly restrict the domain and codomain of the operator ${}_{\sigma^{in}|-|\sigma^{out}}$ defined in Section 2.2 to

$${}_{\sigma^{in}|-|\sigma^{out}} : Mod_{X^{out}}(\Sigma^{out})^{Mod_{X^{in}}(\Sigma^{in})} \to Mod_{Y^{out}}(\Sigma'^{out})^{Mod_{Y^{in}}(\Sigma'^{in})}$$

whenever

— $X^{out}{}_{|Sorts(\sigma^{out})} = Y^{out}$;
— $Y^{in}{}_{|Sorts(\sigma^{in})} = X^{in}$.

Proof. The result is a direct consequence of the fact that $|-| : Mod \dot{\to} SSet \circ Sorts^{op}$ is a natural transformation. □

Definition 4.12. Let $SortMixMod : \mathbf{SortMixSig}^{op} \to \mathbf{Set}$ be defined as follows:

— for any sorted mixin signature $<S^{in}, \Sigma^{in}, S^{out}, \Sigma^{out}>$,

$$SortMixMod(<S^{in}, \Sigma^{in}, S^{out}, \Sigma^{out}>)$$

is the set of the sorted mixin models over $<S^{in}, \Sigma^{in}, S^{out}, \Sigma^{out}>$;
— for any morphism

$$<\sigma^{in}, \sigma^{out}> : <S'^{in}, \Sigma'^{in}, S'^{out}, \Sigma'^{out}> \to <S^{in}, \Sigma^{in}, S^{out}, \Sigma^{out}>$$

https://doi.org/10.1017/S0960129598002576 Published online by Cambridge University Press

and for any sorted mixin model $<H, F>$ over $<S^{in}, \Sigma^{in}, S^{out}, \Sigma^{out}>$,

$$SortMixMod(<\sigma^{in}, \sigma^{out}>)(<H, F>) = <H', F'>$$

where

$$H' = {}_{f^{in}|}H_{|f^{out}} (= \lambda Y.H(Y_{|f^{in}})_{|f^{out}}),$$
$$F'_Y = {}_{\sigma^{in}|}(F_Z)_{|\sigma^{out}}, \text{with } Z = Y_{|f^{in}}, \text{ for any } Y \in SSet(S'^{in}).$$

We denote $<H', F'>$ by ${}_{\sigma^{in}|}<H, F>_{|\sigma^{out}}$.

Fact 4.13. The functor *SortMixMod* is well-defined.

Proof. The function H' is continuous by virtue of Fact A.6 in the Appendix. The functoriality of *SortMixMod* is obvious. It remains to show that ${}_{\sigma^{in}|}(F_Z)_{|\sigma^{out}}$ is well-defined.

$$\begin{aligned}
(H(Z) + Z)_{|Sorts(\sigma^{out})} &= \quad \text{(by the amalgamation property)} \\
(H(Z) + Y)_{|(id_{S^{out}} + f^{in}) \circ Sorts(\sigma^{out})} &= \quad \text{(by the definition of } \textbf{SortMixSig}) \\
(H(Z) + Y)_{|f^{out} + id_{S'^{in}}} &= \quad \text{(by the amalgamation property)} \\
H(Z)_{|f^{out}} + Y &= H'(Y) + Y \\
(H'(Y) + Y)_{|Sorts(\sigma^{in})} &= \\
(H(Z)_{|f^{out}} + Y)_{|Sorts(\sigma^{in})} &= \quad \text{(by the amalgamation property)} \\
(H(Z)_{|f^{out}} + Y)_{|(f^{out} + id_{S'^{in}}) \circ Sorts(\sigma^{in})} &= \quad \text{(by the definition of } \textbf{SortMixSig}) \\
(H(Z)_{|f^{out}} + Y)_{|id_{S^{out}} + f^{in}} &= \quad \text{(by the amalgamation property)} \\
H(Z) + Z &
\end{aligned}$$

Hence we conclude by Lemma 4.11. □

Definition 4.14. Let $F_i : Mod_X(\Sigma^{in}) \to Mod_{Y_i}(\Sigma_i^{out})$, $i = 1, 2$, be two functions, with $Y_{1|S} = Y_{2|S}$, $S = Sorts(\Sigma_1^{out}) \cap Sorts(\Sigma_2^{out})$. Then

$$F_1 \oplus F_2 : Mod_X(\Sigma^{in}) \to Mod_{Y_1 \oplus Y_2}(\Sigma_1^{out} \oplus \Sigma_2^{out})$$

is the function defined by

$$(F_1 \oplus F_2)(A) = F_1(A) \oplus F_2(A), \text{for any } A \in Mod_X(\Sigma^{in}).$$

Note that $F_1(A) \oplus F_2(A)$ is well-defined since $|F_1(A)|_{|S} = Y_{1|S} = Y_{2|S} = |F_2(A)|_{|S}$, therefore, by Definition 4.4, $(F_1(A))_{|\varnothing_S} = (F_2(A))_{|\varnothing_S}$. Moreover, by the naturality of $|-|$ and the amalgamation property, $|F_1(A) \oplus F_2(A)|_{|Sorts(\Sigma_i^{out})} = |(F_1(A) \oplus F_2(A))_{|\Sigma_i^{out}}| = |F_i(A)|$, $i = 1, 2$, therefore, $|F_1(A) \oplus F_2(A)| = |F_1(A)| \oplus |F_2(A)| = Y_1 \oplus Y_2$, and $F_1(A) \oplus F_2(A) \in Mod_{Y_1 \oplus Y_2}(\Sigma_1^{out} \oplus \Sigma_2^{out})$.

https://doi.org/10.1017/S0960129598002576 Published online by Cambridge University Press

Definition 4.15. Let $<H_i, F_i>$ be models over $<S_i^{in}, \Sigma_i^{in}, S_i^{out}, \Sigma_i^{out}>$), $i = 1, 2$, with $S_1^{out} \cap S_2^{out} = \varnothing$. Then $<H, F> = <H_1, F_1> \uplus <H_2, F_2>$ is the sorted mixin model over

$$<(S_1^{in} \cup S_2^{in}) \setminus (S_1^{out} \cup S_2^{out}), \Sigma_1^{in} \oplus \Sigma_2^{in}, S_1^{out} \cup S_2^{out}, \Sigma_1^{out} \oplus \Sigma_2^{out}>$$

defined by:
for any $Y \in SSet((S_1^{in} \cup S_2^{in}) \setminus (S_1^{out} \cup S_2^{out}))$

$$H(Y) = fix(\lambda Z.H_1(Y_{|S_1^{in} \setminus S_2^{out}} + Z_{|S_1^{in} \cap S_2^{out}}) + H_2(Y_{|S_2^{in} \setminus S_1^{out}} + Z_{|S_2^{in} \cap S_1^{out}}))$$
$$F_Y = {}_{\Sigma_1^{in} \oplus \Sigma_2^{in}|}F_{1\,Y_1} \oplus {}_{\Sigma_1^{in} \oplus \Sigma_2^{in}|}F_{2\,Y_2}, \text{with} \quad Y_i = (Y + H(Y))_{|S_i^{in}}, \quad i = 1, 2.$$

Fact 4.16. The sum $<H_1, F_1> \uplus <H_2, F_2>$ is well-defined.

Proof. The continuity of H comes directly from Facts A.6 and A.7 in the Appendix and the fact that *fix* is continuous.
By the amalgamation and the fixed point property, we can derive the following equalities:

$$\begin{aligned}
(Y + H(Y))_{|Sorts(\sigma_1^{in})} &= (Y + H(Y))_{|S_1^{out} \cup S_1^{in}} \\
&= (Y + H(Y))_{|S_1^{out}} + (Y + H(Y))_{|S_1^{in}} \\
&= H(Y)_{|S_1^{out}} + (Y + H(Y))_{|S_1^{in}} \\
&= H_1(Y_{|S_1^{in} \setminus S_2^{out}} + H(Y)_{|S_1^{in} \cap S_2^{out}}) + (Y + H(Y))_{|S_1^{in}} \\
&= H_1((Y + H(Y))_{|S_1^{in}}) + (Y + H(Y))_{|S_1^{in}} \\
&= H_1(Y_1) + Y_1.
\end{aligned}$$

Analogously, $(Y + H(Y))_{|Sorts(\sigma_2^{in})} = H_2(Y_2) + Y_2$, hence, by Lemma 4.11, ${}_{\Sigma_1^{in} \oplus \Sigma_2^{in}|}F_{i|Y_i}$ is well-defined, for $i = 1, 2$.
Now set $S = (S_1^{in} \cup S_1^{out}) \cap (S_2^{in} \cup S_2^{out})$. Then

$$\begin{aligned}
(H_1(Y_1) + Y_1)_{|S} &= (Y + H(Y))_{|S_1^{in} \cup S_1^{out}|S} \\
&= (Y + H(Y))_{|S} \\
&= (Y + H(Y))_{|S_2^{in} \cup S_2^{out}|S} \\
&= (H_2(Y_2) + Y_2)_{|S},
\end{aligned}$$

therefore ${}_{\Sigma_1^{in} \oplus \Sigma_2^{in}|}F_{1|Y_1} \oplus {}_{\Sigma_1^{in} \oplus \Sigma_2^{in}|}F_{2|Y_2}$ is well-defined.
Moreover,

$$\begin{aligned}
(H_1(Y_1) + Y_1) \oplus (H_2(Y_2) + Y_2) &= (Y + H(Y))_{|S_1^{in} \cup S_1^{out}} \oplus (Y + H(Y))_{|S_2^{in} \cup S_2^{out}} \\
&= Y + H(Y),
\end{aligned}$$

hence

$${}_{\Sigma_1^{in} \oplus \Sigma_2^{in}|}F_{1|Y_1} \oplus {}_{\Sigma_1^{in} \oplus \Sigma_2^{in}|}F_{2|Y_2} \in Mod_{Y + H(Y)}(\Sigma_1^{out} \oplus \Sigma_2^{out}).$$ □

Definition 4.17. Let $<H, F>$ be a model over $<S^{in}, \Sigma^{in} \oplus \Sigma^{fr}, S^{out}, \Sigma^{out}>$, and let $\sigma: \Sigma^{fr} \to \Sigma^{out}$ be a morphism such that $Sorts(\sigma) = id_{S^{in} \cup S^{out}}$. Then

$$<H, F'> = freeze_{\sigma}(<H, F>)$$

https://doi.org/10.1017/S0960129598002576 Published online by Cambridge University Press

is the model over $<S^{in}, \Sigma^{in}, S^{out}, \Sigma^{out}>$ defined by

$$\text{for any } Y \in SSet(S^{in}), F'_Y = \lambda A.freeze_{Y+H(Y),\Sigma^{out}}(\lambda B.F_Y(A \oplus B_{|\sigma})).$$

Note that $|A| = Y + H(Y) = |B| = |B|_{|Sorts(\sigma)} = |B_{|\sigma}|$, therefore $A_{|\varnothing_{S^{in} \cup S^{out}}} = B_{|\sigma|\varnothing_{S^{in} \cup S^{out}}}$, $A \oplus B_{|\sigma}$ is well-defined and $|A \oplus B_{|\sigma}| = Y + H(Y)$.

4.3. *Basic operators*

We can also in the case of sorted core frameworks define the analogue of the three basic operators defined in Section 2.3.

Sum If $<H_i, F_i>$ are sorted mixin models over $<S_i^{in}, \Sigma_i^{in}, S_i^{out}, \Sigma_i^{out}>$), $i = 1, 2$, with $S_1^{out} \cap S_2^{out} = \varnothing$, then $<H, F> = <H_1, F_1> \uplus <H_2, F_2>$ denotes a sorted mixin model over

$$<(S_1^{in} \cup S_2^{in}) \setminus (S_1^{out} \cup S_2^{out}), \Sigma_1^{in} \oplus \Sigma_2^{in}, S_1^{out} \cup S_2^{out}, \Sigma_1^{out} \oplus \Sigma_2^{out}>.$$

We have extended the basic operator $\uplus$ rather than $+$ as defined in Section 2.3 since, in the case of sorts, the requirement that the two input signatures must coincide is too restrictive, since $S_1^{in} = S_2^{in}$, $\Sigma_1^{in} = \Sigma_2^{in}$ and $S_1^{out} \cap S_2^{out} = \varnothing$ would imply $S_1^{out} = S_2^{out} = \varnothing$.

However, this sum operator allows the sharing of the input sorts (S_1^{in} and S_2^{in} are not required to be disjoint) and of the sorts in the resulting input and output signatures ($\Sigma_1^{in} \oplus \Sigma_2^{in}$ and $\Sigma_1^{out} \oplus \Sigma_2^{out}$, respectively). Indeed, the condition $S^{in} \cap S^{out} = \varnothing$ on sorted mixin signatures implies that mixins cannot have virtual type components and that, as a result, types cannot be frozen. Therefore, when combining two mixins we must assume that the common sorts in the two mixins just represent the same sorts (in other words, there is no duplication of equal sorts). This allows us to correctly bind deferred types of one mixin to the defined ones of the other mixin, and conversely; as a consequence, the resulting set of input sorts is given by $(S_1^{in} \cup S_2^{in}) \setminus (S_1^{out} \cup S_2^{out})$, so that the sorts in $S_1^{in} \cap S_2^{out}$ and $S_2^{in} \cap S_1^{out}$ are no longer deferred. Notice that $((S_1^{in} \cup S_2^{in}) \setminus (S_1^{out} \cup S_2^{out})) \cap (S_1^{out} \cup S_2^{out}) = \varnothing$ (no virtual type components), and that $((S_1^{in} \cup S_2^{in}) \setminus (S_1^{out} \cup S_2^{out})) \cup (S_1^{out} \cup S_2^{out}) = S_1^{in} \cup S_2^{in} \cup S_1^{out} \cup S_2^{out}$ (no new type is added, and no pre-existent type is deleted).

Reduct For any morphism

$$<\sigma^{in}, \sigma^{out}>: <S'^{in}, \Sigma'^{in}, S'^{out}, \Sigma'^{out}> \to <S^{in}, \Sigma^{in}, S^{out}, \Sigma^{out}>$$

and for any sorted mixin model

$$<H, F> \text{ over } <S^{in}, \Sigma^{in}, S^{out}, \Sigma^{out}>,$$

${}_{\sigma^{in}|}<H, F>_{|\sigma^{out}}$ denotes a sorted mixin model over $<S'^{in}, \Sigma'^{in}, S'^{out}, \Sigma'^{out}>$.

Freeze If $<H, F>$ is a model over $<S^{in}, \Sigma^{in} \oplus \Sigma^{fr}, S^{out}, \Sigma^{out}>$) and $\sigma: \Sigma^{fr} \to \Sigma^{out}$ is a morphism such that $Sorts(\sigma) = id_{S^{in} \cup S^{out}}$, then $freeze_\sigma(<H, F>)$ denotes a sorted mixin model over $<S^{in}, \Sigma^{in}, S^{out}, \Sigma^{out}>$.

Notice that the type components are not frozen, since they are not virtual; indeed the set S^{in} does not change and the morphism $\sigma: \Sigma^{fr} \to \Sigma^{out}$ is the identity over the sorts.

https://doi.org/10.1017/S0960129598002576 Published online by Cambridge University Press

However, the components to be frozen (Σ^{fr}) have sorts ranging over $S^{in} \cup S^{out}$, therefore the two signatures Σ^{fr} and Σ^{in} are not completely disjoint, but they share the sorts.

5. Related research and further work

As mentioned in the introduction, the name *mixin* was first used in the LISP community (Moon 1986; Keene 1989). However, the first clear formulation of the concept was given in Bracha and Cook (1990), where the possibility of explicitly naming mixins is proposed as a useful linguistic feature (*mixin-based inheritance*) and it is shown that the inheritance mechanism of Smalltalk and Beta can both be seen as the same mixin combinator, as proved in a formal context in this paper.

Later, G. Bracha (Bracha 1992; Bracha and Lindstrom 1992) proposed a set of module operators (called Jigsaw), which provides a framework for modularity independent of a particular computational paradigm. In other words, Jigsaw defines a language of mixin modules parameterized by the programming language used for defining module components (the core language). For instance, in Bracha (1992), Jigsaw is instantiated over Modula-3 obtaining an extension of this language supporting the new operators for module combination.

In Banavar (1995) and Banavar and Lindstrom (1996), G. Banavar extended Bracha's work by realizing an object-oriented *application framework* (called Etyma) and by introducing a composition operation for hierarchical nesting. More importantly, Banavar has shown that the notion of mixin module, intended as a collection of self-referencing components, can be successfully applied within a wide range of systems by building four different tools as completions of Etyma: an interpreter for a module extension to the functional programming language Scheme; a programmable linker; a compiler front-end for a compositional interface definition language; and a compositional document processing system.

In Van Limbergehn and Mens (1996), a combination of just two operators on mixin modules (a variant of hiding and inheritance with a *super* mechanism) is proposed as a solution to multiple inheritance problems such as name collisions. The aim in Van Limbergehn and Mens (1996) is to limit the flexibility of basic operations for mixin composition by defining on top of them more restrictive operators encouraging orthogonalization of concepts and reinforcing software reliability. For instance, in contrast to our *other* mechanism, the *super* mechanism proposed in Van Limbergehn and Mens (1996) allows one to refer only to the method that is overridden by the definition where *super* is invoked.

Another paper involving mixin modules is Duggan and Sourelis (1996), where they are proposed as a new construct for SML. In Duggan and Sourelis (1996) mixins can be combined by means of a binary operator $\otimes$ that allows one to mix together not only definitions of different components (like our merge), but also two definitions for the same component (data-type or function). This is possible, since in SML data-types and functions are defined by cases, thus it makes sense to consider the 'union' of two definitions. The $\otimes$ operator is non-commutative, since, in $M_1 \otimes M_2$, M_1 can refer to further extensions of function definitions provided by M_2 via an *inner* mechanism, while *inner*

https://doi.org/10.1017/S0960129598002576 Published online by Cambridge University Press

calls in M_2 remain open, as in our weak overriding operator. However, no possibility of overriding is considered (no virtual components): the main aim is to introduce in SML modules the possibility that recursive definitions span module boundaries. The proposal turns out to be very specific and tied to the features of SML.

Finally, Bono *et al.* (1996) presents a lambda calculus of *incomplete objects*, that is, objects (records of methods) that may be typed even though they contain references to methods that are yet to be added; it is easy to recognize that these incomplete objects are mixins.

The work of Bracha has been by far the most important source of inspiration of our own work. We are indebted to Jigsaw both for the overall idea of defining a language of mixin modules parameterized by the underlying core language and for some operators, such as *restrict*. However, the idea of defining a module system as a small language of its own constructed on top of the base language, on which as few as possible assumptions are made, is now becoming a standard approach, see, for example, some recent work on type-theoretic foundations of SML-like modules (Harper and Lillibridge 1994; Leroy 1994; Leroy 1996; Jones 1996; Courant 1997; Courant 1997a). A paper that can be taken as representative of this point of view, summarized as a slogan in the title ('A modular module system'), is Leroy (1996), whose aim is to give a constructive proof of the validity of the idea. To this end, it presents (the implementation by an SML functor of) a transformation that takes a core language and its associated type-checking functions, and returns an SML-style modular language with its type-checker. The input interface of the functor gives sufficient conditions for an existing or future language to support SML-style modules.

The first main contribution of our paper is to give a rigorous counterpart at the semantic level of this two-level view of a modular language. In our opinion, the work we have done is very much in the spirit of Leroy (1996) cited above. Indeed, we also describe a transformation that takes as parameter a core language and gives a modular language (in particular, a language of mixin modules): the difference is that in our case we deal with the semantic descriptions of the two levels. In that respect our work is new with respect to Bracha (1992), where, though supported as a methodological principle, the parameterization by the core language is not explicitly formalized, and the independence of the proposed framework from the object-oriented nature of the language is not always clearly stated.

The second main contribution of our paper is to provide an analysis of the formal basis of the relationship between different operators over mixins, identifying three basic operators that allow us to express as derived constructs a variety of other operators closer to concrete programming languages. In Ancona and Zucca (1998), we develop this aspect further by providing a set of algebraic laws holding between the operators (for example, commutativity and associativity of merge), and proving a normal form theorem in the spirit of Bergstra *et al.* (1990).

Finally, an important byproduct of our research is to have recognized the need for handling type definitions in mixins differently from other components. This aspect should be analyzed at a deeper level as, more generally, should the relation of our approach to the type-theoretic analysis of module languages (see below).

Some continuation of the work done in this paper is in the already cited Ancona and Zucca (1998), and in Ancona (1996) and Ancona and Zucca (1997), which are both concerned with the instantiation of our framework in concrete cases. In Ancona (1996), a concrete mixin language is obtained by fixing a particular core framework. In Ancona and Zucca (1997) we give the translation in terms of our operators of various overriding mechanisms present in programming languages, and we are able, in this way, to formalize the relation between these different versions (this paper largely extends the work outlined in Section 3 here). Finally, a comprehensive survey of our work on mixins is given in Ancona (1998).

The most interesting aspect left for future research is, as mentioned above, the relation of our approach to the work on type aspects of module languages. We pointed out in Section 4 that our present choice of forbidding type redefinitions could be relaxed to allow *abstract data type definitions* in the sense of SML. The introduction in our framework of the notion of *manifest types* (types whose definitions are visible in the module signature (Leroy 1994)) would allow mixins to share type definitions. Finally, we plan to develop (within a common project) an integration of our mixin language with the functional module calculus defined in Courant (1997a), which enjoys the important property of subject reduction and guaranteeing true separate compilation.

Another topic of particular relevance is the relation between mixins and parameterized modules, like SML functors. In this paper, we have already given an interesting result showing that functional application can be expressed in terms of our three basic operators. Since overriding can be defined as a derived operator as well, we have actually shown that both inheritance and genericity (often considered two opposite approaches to increasing software modularity, and the subject of great debate in the object-oriented community (Meyer 1986)) can be seen as two higher level mechanisms expressible by the same set of primitives.

Acknowledgments.

We warmly thank the anonymous referee for the accurate revision and the helpful comments on a preceding version of this paper.

Appendix A.

A.1. *Boolean signature categories*

Notation. If **C** is a category, then $|\mathbf{C}|$ denotes the class of its objects.

Definition A.1. An *inclusive signature category* is a pair $<\mathbf{Sig}, \mathscr{I}>$ where **Sig** is a category whose objects are called *signatures* and $\mathscr{I}$ is a subcategory of **Sig** with $|\mathscr{I}| = |\mathbf{Sig}|$ a distributive lattice – we call the morphisms in $\mathscr{I}$ *inclusions*, use the notation $\Sigma_1 \subseteq \Sigma_2$ if there is an inclusion from Σ_1 into Σ_2, and denote this (unique) inclusion by i_{Σ_1,Σ_2}. We call *union* (denoted by $\Sigma_1 \cup \Sigma_2$) and *intersection* (denoted by $\Sigma_1 \cap \Sigma_2$), respectively, the join and the meet of Σ_1 and Σ_2 in $\mathscr{I}$.

The pair $<\mathbf{Sig}, \mathscr{I}>$ is denoted simply by **Sig** when there is no ambiguity.

In the original definition of inclusive categories (Ancona and Zucca 1996; Diaconescu *et al.* 1993), which is at the basis of the above definition, a property of *unique factorization* of any morphism as an epimorphism composed with an inclusion is required, which corresponds to the intuition that any morphism defines an 'image' object that is included in the codomain. However, this property is unnecessary for the following technical treatment.

Definition A.2. A *boolean signature category* is an inclusive signature category where:

— there exists the bottom element $\varnothing$;
— let us say that two signatures Σ_1 and Σ_2 are *disjoint*, and write $\Sigma_1 \mid \Sigma_2$, if $\Sigma_1 \cap \Sigma_2 = \varnothing$; then, for any $\Sigma_1, \Sigma_2 \in |\mathbf{Sig}|$, with $\Sigma_1 \subseteq \Sigma_2$, there exists a signature, denoted by $\Sigma_2 \setminus \Sigma_1$, such that $(\Sigma_2 \setminus \Sigma_1) \cup \Sigma_1 = \Sigma_2$ and $(\Sigma_2 \setminus \Sigma_1) \mid \Sigma_1$.

We extend the definition of $\Sigma_2 \setminus \Sigma_1$ to arbitrary pairs of signatures, by letting $\Sigma_2 \setminus \Sigma_1 = \Sigma_2 \setminus (\Sigma_2 \cap \Sigma_1)$.

Fact A.3. For any $\Sigma_1, \Sigma_2 \in |\mathbf{Sig}|$, $\Sigma_2 \setminus \Sigma_1$ is the unique signature such that $(\Sigma_2 \setminus \Sigma_1) \mid \Sigma_1$ and $(\Sigma_2 \setminus \Sigma_1) \cup \Sigma_1 = \Sigma_2 \cup \Sigma_1$.

Proof. By definition $(\Sigma_2 \setminus \Sigma_1) \cup (\Sigma_1 \cap \Sigma_2) = \Sigma_2$, hence $(\Sigma_2 \setminus \Sigma_1) \cup \Sigma_1 = \Sigma_2 \cup \Sigma_1$. Since $\Sigma_2 \setminus \Sigma_1 \subseteq \Sigma_2$, we have $(\Sigma_2 \setminus \Sigma_1) \cap \Sigma_1 = (\Sigma_2 \setminus \Sigma_1) \cap \Sigma_1 \cap \Sigma_2$, and then we conclude by definition.
For the uniqueness, assume that Σ' and Σ'' are such that $\Sigma' \mid \Sigma_1$, $\Sigma'' \mid \Sigma_1$ and $\Sigma' \cup \Sigma_1 = \Sigma_2 \cup \Sigma_1$, $\Sigma'' \cup \Sigma_1 = \Sigma_2 \cup \Sigma_1$. Then

$$\begin{aligned}\Sigma' &= \Sigma' \cup (\Sigma'' \cap \Sigma_1)\\ &= (\Sigma' \cup \Sigma'') \cap (\Sigma' \cup \Sigma_1)\\ &= (\Sigma' \cup \Sigma'') \cap (\Sigma_2 \cup \Sigma_1)\\ &= (\Sigma' \cup \Sigma'') \cap \Sigma_2\\ &= \Sigma' \cup \Sigma''\end{aligned}$$

Analogously, we can show that $\Sigma'' = \Sigma' \cup \Sigma''$. □

In the context of boolean signatures, it is enough to associate with each mixin expression a type information of the form $\Sigma^{in} \to \Sigma^{out}$; indeed, the type information consisting of the three components $<\Sigma^{def}, \Sigma^{vir}, \Sigma^{fro}>$ associated with a mixin expression in Section 3 is in this case $<\Sigma^{in} \setminus \Sigma^{out}, \Sigma^{in} \cap \Sigma^{out}, \Sigma^{out} \setminus \Sigma^{in}>$.

Correspondingly, the typing rules for the operators of the kernel language defined in Section 3 can be rewritten as shown in Figure 11. We have omitted the semantic rules, since they can be simply obtained by specializing the rules in Section 3 according to the new typing rules.

A.2. *The domain of S-sorted sets*

For the definitions concerning domain theory we refer to Tennent (1991).

Definition A.4. For any set S, let $\sqsubseteq_S$ denote the partial order over $SSet(S)$ defined as follows:

$$X \sqsubseteq_S Y \text{ iff for any } s \in S \; X_s \subseteq Y_s$$

https://doi.org/10.1017/S0960129598002576 Published online by Cambridge University Press

$$\text{(M-ty)}\ \frac{M_i : \Sigma_i^{in} \to \Sigma_i^{out}, i = 1,2}{M_1 \oplus M_2 : (\Sigma_1^{in} \cup \Sigma_2^{in}) \setminus (\Sigma_1^{fr} \cup \Sigma_2^{fr}) \to \Sigma_1^{out} \cup \Sigma_2^{out}} \quad \begin{array}{l} \Sigma_1^{out} \cap \Sigma_2^{out} = \varnothing \\ \Sigma_i^{fr} = \Sigma_i^{out} \setminus \Sigma_i^{in}, i = 1,2 \end{array}$$

$$\text{(F-ty)}\ \frac{M : \Sigma^{in} \to \Sigma^{out}}{\textbf{freeze}\ \Sigma^{fr}\ \textbf{in}\ M : \Sigma^{in} \setminus \Sigma^{fr} \to \Sigma^{out}} \quad \Sigma^{fr} \subseteq \Sigma^{out}$$

$$\text{(R-ty)}\ \frac{M : \Sigma^{in} \to \Sigma^{out}}{\textbf{restrict}\ \Sigma^{rs}\ \textbf{in}\ M : \Sigma^{in} \to \Sigma^{out} \setminus \Sigma^{rs}} \quad \Sigma^{rs} \subseteq \Sigma^{out}$$

$$\text{(H-ty)}\ \frac{M : \Sigma^{in} \to \Sigma^{out}}{\textbf{hide}\ \Sigma^{hd}\ \textbf{in}\ M : \Sigma^{in} \setminus \Sigma^{hd} \to \Sigma^{out} \setminus \Sigma^{hd}} \quad \Sigma^{hd} \subseteq \Sigma^{out}$$

$$\text{(O-ty)}\ \frac{M_i : \Sigma_i^{in} \to \Sigma_i^{out} \qquad i = 1,2}{M_1 \Leftarrow M_2 : (\Sigma_1^{in} \cup \Sigma_2^{in}) \setminus (\Sigma_1^{fr} \cup \Sigma_2^{fr}) \to \Sigma_1^{out} \cup \Sigma_2^{out}} \quad \begin{array}{l} \Sigma_1^{fr} = (\Sigma_1^{out} \setminus \Sigma_2^{out}) \setminus \Sigma_1^{in} \\ \Sigma_2^{fr} = \Sigma_2^{out} \setminus \Sigma_2^{in} \end{array}$$

$$\text{(FC-ty)}\ \frac{M_i : \Sigma_i^{in} \to \Sigma_i^{out} \qquad i = 1,2}{M_2 \circ M_1 : (\Sigma_1^{in} \setminus \Sigma_1^{out}) \cup (\Sigma_2^{in} \cap \Sigma_2^{out}) \to \Sigma_2^{out}} \quad \begin{array}{l} \Sigma_2^{in} \setminus \Sigma_2^{out} = \Sigma_1^{out} \\ (\Sigma_1^{in} \setminus \Sigma_1^{out}) \cap \Sigma_2^{out} = \varnothing \end{array}$$

Fig. 11. Typing rules for boolean signatures

Note that it is straightforward to verify that $\sqsubseteq_S$ is a partial order.

Fact A.5. For any set S, $<SSet(S), \sqsubseteq_S>$ is a domain with bottom element.

Proof. It is immediate to show that

— for any increasing chain $(X_i)_{i\in\omega}$, its limit $\sqcup(X_i)_{i\in\omega}$ is such that for any $s \in S$ $(\sqcup(X_i)_{i\in\omega})_s = \cup_{i\in\omega}(X_i)_s$;
— the bottom element is the S-sorted set $\perp$ such that for each $s \in S$ $\perp_s = \varnothing$. □

Fact A.6. For any function $f : S' \to S$, $-_{|f} : SSet(S) \to SSet(S')$ is continuous.

Proof. For any $s \in S'$ we have

$$((\sqcup(X_i)_{i\in\omega})_{|f})_s = (\sqcup(X_i)_{i\in\omega})_{f(s)} = \cup_{i\in\omega}(X_i)_{f(s)} = \cup_{i\in\omega}((X_i)_{|f})_s = (\sqcup((X_i)_{|f})_{i\in\omega})_s$$

□

Fact A.7. For any pair of sets S_1 and S_2 such that $S_1 \cap S_2 = \varnothing$, the function g

$$\_ + \_ : SSet(S_1) \times SSet(S_2) \to SSet(S_1 \cup S_2)$$

is continuous.

https://doi.org/10.1017/S0960129598002576 Published online by Cambridge University Press

Proof. For any $s \in S_1 \cup S_2$ we have

$$(\sqcup(X_i)_{i\in\omega} + \sqcup(Y_i)_{i\in\omega})_s = \begin{cases} (\sqcup(X_i)_{i\in\omega})_s = \cup_{i\in\omega}(X_i)_s & \text{if } s \in S_1 \\ (\sqcup(Y_i)_{i\in\omega})_s = \cup_{i\in\omega}(Y_i)_s & \text{if } s \in S_2 \end{cases}$$

$$(\sqcup(X_i + Y_i)_{i\in\omega})_s = \cup_{i\in\omega}(X_i + Y_i)_s = \begin{cases} \cup_{i\in\omega}(X_i)_s & \text{if } s \in S_1 \\ \cup_{i\in\omega}(Y_i)_s & \text{if } s \in S_2 \end{cases}$$

□

References

Adámek, J., Herrlich, H. and Strecker, G. (1990) *Abstract and Concrete Categories*, Pure and Applied Mathematics, Wiley Interscience, New York.

Ancona, D. (1996) MIX(FL): a kernel language of mixin modules. Technical Report DISI-TR-96-23, DISI, University of Genova (submitted for journal publication).

Ancona, D. (1998) *Modular Formal Frameworks for Module Systems.* Ph. D. thesis, Dip. Informatica, Univ. Pisa (to appear).

Ancona, D. and Zucca, E. (1998) An algebra of mixin modules. 12th Workshop on Algebraic Development Techniques – Selected Papers. *Springer-Verlag Lecture Notes in Computer Science* (to appear).

Ancona, D. and Zucca, E. (1996) An algebraic approach to mixins and modularity. In: Hanus, M. and Rodríguez Artalejo, M. (eds.) ALP '96 – 5th Intl. Conf. on Algebraic and Logic Programming. *Springer-Verlag Lecture Notes in Computer Science* **1139** 179-193.

Ancona, D. and Zucca, E. (1996) A formal framework for modules with state. In: Wirsing, M. and Nivat, M. (eds.) AMAST '96 (Algebraic Methodology and Software Technology 1996). *Springer-Verlag Lecture Notes in Computer Science* **1101** 148–162.

Ancona, D. and Zucca, E. (1997) Overriding operators in a mixin-based framework. In: Glaser, H., Hartel, P. and Kuchen, H. (eds.) Proc. PLILP '97 (9th International Symposium on Programming Languages, Implementations, Logics, and Programs). *Springer-Verlag Lecture Notes in Computer Science* **1292** 47–61.

Arnold, K. and Gosling, J. (1996) *The Java™ programming language*, Addison-Wesley.

Banavar, G. (1995) *An Application Framework for Compositional Modularity*, Ph. D. thesis, Department of Comp. Sci., Univ. of Utah.

Banavar, G. and Lindstrom, G. (1996) An application framework for module composition tools. In: Proc. of European Conference on Object-Oriented Programming. *Springer-Verlag Lecture Notes in Computer Science* **1098** 91–113.

Bergstra, J. A., Heering, J. and Klint, P. (1990) Module algebra. *Journal of the Association for Computing Machinery* **37** (2) 335-372.

Bidoit, M. and Tarlecki, A. (1996) Behavioural satisfaction and equivalence in concrete model categories. In: Kirchner, H. (ed.) CAAP '96 (20th Coll. on Trees in Algebra and Computing). *Springer-Verlag Lecture Notes in Computer Science* **1059** 241–256.

Bono, V., Bugliesi, M. and Liquori, L. (1996) A lambda calculus of incomplete objects. In: Penczek, W. and Szałas, A. (eds.) Mathematical Foundations of Computer Science 1996. *Springer-Verlag Lecture Notes in Computer Science* **1113** 218–229.

Bracha, G. (1992) *The Programming Language JIGSAW: Mixins, Modularity and Multiple Inheritance*, Ph. D. thesis, Department of Comp. Sci., Univ. of Utah.

Bracha, G. and Cook, W. (1990) Mixin-based inheritance. In: *Proc. of the Joint ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications and the European Conference on Object-Oriented Programming*.

Bracha, G. and Lindstrom, G. (1992) Modularity meets inheritance. In: *Proc. International Conference on Computer Languages*, San Francisco, IEEE Computer Society 282–290.

Cook, W. R. (1989) *A Denotational Semantics of Inheritance*, Ph. D. thesis, Dept. of Computer Science, Brown University.

Courant, J. (1997) An applicative module calculus. In: Bidoit, M. and Dauchet, M. (eds.) Proc. TAPSOFT '97 (Theory and Practice of Software Development). *Springer-Verlag Lecture Notes in Computer Science* **1217** 622–636.

Courant, J. (1997) A module calculus for pure type systems. In: TLCA'97 (3rd Intl. Conf. on Typed Lambda Calculi and Applications). *Springer-Verlag Lecture Notes in Computer Science* **1210**.

Diaconescu, R., Goguen, J. and Stefaneas, P. (1991) Logical support for modularisation. In: Huet, G. and Plotkin, G. (eds.) *Logical Environments* (Proceedings of a Workshop held in Edinburgh), Cambridge University Press 83–130.

Duggan, D. and Sourelis, C. (1996) Mixin modules. In: *Intl. Conf. on Functional Programming*, Philadelphia, ACM Press.

Ehrig, H., Baldamus, M., Cornelius, F. and Orejas, F. (1991) Theory of algebraic module specification including behavioural semantics, constraints and aspects of generalized morphisms. In: Nivat, M., Rattray, C., Rus, T. and Scollo, G. (eds.) *Proc. 2nd Int. Conf. on Algebraic Methodology and Software Technology, Iowa City, USA, May 91*, Workshops in Computing, Springer-Verlag 145–172.

Ehrig, H. and Löwe, M. (1993) Categorical principles, techniques and results for high-level replacement systems in computer science. *Appl. Cat. Struct.* **1** 21–50.

Ehrig, H. and Mahr, B. (1985) Fundamentals of Algebraic Specification 1. Equations and Initial Semantics. *EATCS Monographs in Computer Science* **6**, Springer-Verlag,

Ehrig, H. and Mahr, B. (1990) Fundamentals of Algebraic Specification 2. Module Specifications and Constraints. *EATCS Monographs in Computer Science* **21**, Springer-Verlag.

Goguen, J. A. and Burstall, R. (1992) Institutions: abstract model theory for specification and programming. *Journ. ACM* **39** (1) 95–146.

Goldberg, A. and Robson, D. (1983) *Smalltalk-80: the Language and Its Implementation*, Addison-Wesley.

Harper, R. and Lillibridge, M. (1994) A type theoretic approach to higher-order modules with sharing. In: *Proc. 21st ACM Symp. on Principles of Programming Languages*, ACM Press 127–137.

Jones, M. P. (1996) Using parameterized signatures to express modular structure. In: *Proc. 23rd ACM Symp. on Principles of Programming Languages*, St. Petersburg Beach, Florida, ACM Press 68–78.

Keene, S. E. (1989) *Object-Oriented Programming in Common Lisp*, Addison Wesley.

Lehrmann, O., Moller, B., Pedersen, and Nygaard, K. (1993) *Object-Oriented Programming in the BETA Programming Language*, ACM Press.

Leroy, X. (1994) Manifest types, modules and separate compilation. In: *Proc. 21st ACM Symp. on Principles of Programming Languages*, ACM Press 109–122.

Leroy, X. (1996) A modular module system. Technical Report 2866, INRIA.

van Limberghen, M. and Mens, T. (1996) Encapsulation and composition as orthogonal operators on mixins: a solution to multiple inheritance problems. *Object-oriented Systems* **3** 1–30.

Loeckx, J., Ehrich, H. D. and Wolf, M. (1996) *Specifications of Abstract Data Types*, Wiley-Teubner Computing.

Meyer, B. (1986) Genericity versus inheritance. In: ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications. *SIGPLAN Notices* **21** (11), ACM Press 391–405.

Meyer, B. (1988) *Object-oriented Software Construction*, Computer Science series, Prentice Hall.

Milner, R., Tofte, M. and Harper, R. (1990) *The Definition of Standard ML*, The MIT Press.

Moon, D. A. (1986) Object-oriented programming with Flavors. In: ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications. *SIGPLAN Notices* **21** (11), ACM Press 1–8.

Reddy, U. S. (1988) Objects as closures: Abstract semantics of object-oriented languages. In *Proc. ACM Conf. on Lisp and Functional Programming* 289–297.

Sannella, D. and Tarlecki, A. (1988) Specification in an arbitrary institution. *Information and Computation* **76** 165–210.

Stroustrup, B. (1991) *The C++ Programming Language*, 2nd edition, Addison-Wesley.

Szyperski, C. (1992) Import is not inheritance. Why we need both: Modules and classes. In: Lehrmann Madsen, O. (ed.) Proc. of ECOOP '92 European Conference on Object-Oriented Programming. *Springer-Verlag Lecture Notes in Computer Science* **615** 19–32.

Tennent, R. D. (1991) *Semantics of Programming Languages*, Computer Science series, Prentice Hall,

Zucca, E. (1996) From static to dynamic abstract data-types. In: Penczek, W. and Szałas, A. (eds.) Mathematical Foundations of Computer Science 1996. *Springer-Verlag Lecture Notes in Computer Science* **1113** 579–590.