Math. Struct. in Comp. Science (2002), *vol.* 12, *pp.* 701–737.
DOI: 10.1017/S0960129502003687 Printed in the United Kingdom

A theory of mixin modules: algebraic laws and reduction semantics[†]

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 10 November 2000; revised 4 September 2001

Mixins are modules that may contain deferred components, that is, components not defined in the module itself; moreover, in contrast to parameterised modules (like ML functors), they can be mutually dependent and allow their definitions to be overridden. In a preceding paper we defined a syntax and denotational semantics of a kernel language of mixin modules. Here, we take instead an axiomatic approach, giving a set of algebraic laws expressing the expected properties of a small set of primitive operators on mixins. Interpreting axioms as rewriting rules, we get a reduction semantics for the language and prove the existence of normal forms. Moreover, we show that the model defined in the earlier paper satisfies the given axiomatisation.

1. Introduction

The notion of a *mixin*, which was first introduced in the context of object-oriented programming (Bracha and Cook 1990), has recently become the subject of increasing interest in many respects and with many slight variations in the intended meaning (Bracha 1992; Bracha and Lindstrom 1992; Banavar and Lindstrom 1996; Duggan and Sourelis 1996; Van Limberghen and Mens 1996; Flatt *et al.* 1998; Duggan and Sourelis 1998; Findler and Flatt 1998).

This paper continues the work in Ancona and Zucca (1998b), where we provided a rigorous formulation of the mixin notion, covering and making precise the various ways in which the word is used in the literature; we will refer to this formulation in the following.

Mixins (or *mixin modules*) are a generalisation of the usual modules in programming languages, which are collections of definitions of heterogeneous components, for example, types, functions, procedures, exceptions and so on (typical examples are Modula-2 or Standard ML). The generalisation has two main features.

First, some of the components can be only declared in the module, without having an associated definition; we say that these components are *deferred* (the terminology comes

[†] Partially supported by Murst (*Tecniche formali per la specifica, l'analisi, la verifica, la sintesi e la trasformazione di sistemi software*) and CNR (*Formalismi per la specifica e la descrizione di sistemi ad oggetti*).

from object-oriented languages). A mixin with deferred components cannot be used in isolation; a binary *merge* operation allows us to combine two mixins, say M_1, M_2, in such a way that a deferred component in M_1 can be *concreted* by a definition provided in M_2, and conversely. This operation is commutative and associative, and no conflicting definitions are allowed, that is, a component defined in both arguments. In the resulting mixin $M_1 \oplus M_2$ there can still be deferred components (if some deferred component in one argument has not been matched in the other), or we can get a *concrete module*, that is, a module with no deferred components, which can be effectively used. An important remark is that the symmetry of the merge operator allows recursive definitions to span module boundaries, with a great benefit for modularity, as illustrated, for example, in Duggan and Sourelis (1996), Ancona and Zucca (1998b) and Crary *et al.* (1999).

The second extension with respect to the usual modules, which is again inspired by the object-oriented approach, is the possibility that when assembling modules together some definition in a module is replaced by a new definition provided in another module. This feature is called *overriding* and is typical of inheritance in object-oriented languages: anyway, the concept turns out to be completely orthogonal to the object-oriented nature of the language and can be formulated independently from the notions of object, class and subtyping hierarchies.

Formally, overriding can be seen as another binary operation such that $M_1 \Leftarrow M_2$ is the same as $M_1 \oplus M_2$, except that there can be conflicting definitions and in this case the definitions in M_2 takes precedence. Overriding can be seen as the composition of two different operations: a *restrict* operation whose effect is to 'cancel' some definitions in a module, and the merge operation (this view of overriding was originally due to Bracha (1992)).

Since definitions of components can refer to each other, redefining a component, say m, can actually change the behaviour of other components, for example, a component m' defined by $m' = \ldots\ m\ \ldots$. This is not always the case: some languages allow the user to specify explicitly whether, if m is redefined, m' should refer to the new or old version. We will say that m is *virtual* in the first case (*cf.* virtual and non virtual methods in C++) and *frozen* in the second (the term *frozen* was introduced in Bracha (1992)).

In Ancona and Zucca (1998b) we proposed a formal model for mixin modules. The basic idea was to see a mixin as a function from input to output components, where output components are those defined in the module, while input components are those on which definitions in the module can depend (hence deferred and virtual components). Moreover, we have defined a kernel language of mixin modules, that is, a set of operators for composing mixins corresponding to a variety of constructs existing in programming languages (including merge, restrict, inheritance/overriding, hiding, functional composition). An important point is that we have not fixed an underlying *core* language (following the ML terminology), but provided a language of modules that can be instantiated on top of a variety of different languages – the module language is a small language of its own, which is, as far as possible, independent of the core language and has its own typing rules. This structure was a design goal of the Standard ML module system (Milner *et al.* 1990) and has been recently recognised as fundamental from both the type theoretic (Leroy 1994; Harper and Lillibridge 1994) and the software engineering (Bracha and Lindstrom

1992; Banavar and Lindstrom 1996; Van Limberghen and Mens 1996) points of view. The idea is also closely related to that of 'institution independent' operators for writing specifications, which were originally proposed in Goguen and Burstall (1992) and further developed, for example, in Sannella and Tarlecki (1986), Sannella and Tarlecki (1988), Sannella *et al.* (1992) and Sannella and Wallen (1992); indeed we achieve the independence from the core language at the semantic level by using an abstract semantic framework very close to that of institutions.

In this paper, we take a different approach to the formal definition of mixin modules, that is, we give an *axiomatic characterisation* of the operators, in the spirit of the seminal paper Bergstra *et al.* (1990). In other words, we state a number of algebraic laws specifying the expected properties of the operators. The axiomatisation is given in two steps: first, we characterise three primitive operations by means of a small set of axioms; then, we give for each operator of the language an axiom stating that it can be expressed in terms of these three primitive operators. Moreover, interpreting axioms as rewriting rules, we get a reduction semantics for the language and prove the existence of normal forms. Finally, we show that the the denotational semantics of the language previously defined actually satisfies the given axiomatisation.

The paper is organised as follows. In Section 2 we provide a summary of our preceding work in Ancona and Zucca (1998b): we introduce basic ideas on mixin modules through some examples in Section 2.1, give syntax and semantics of the kernel language in Section 2.2 and introduce the three primitive operators in Section 2.3. In Section 3 we provide the axiomatic characterisation of the operators (primitive in Section 3.1 and derived in Section 3.3). In Section 4 we prove the existence of normal forms and derive from the specification a confluent and strongly normalizing rewriting system, which provides a reduction semantics for the language. In Section 5 we show that the model defined in Ancona and Zucca (1998b), where mixins are interpreted as functions, actually satisfies the specification in Section 3: hence the 'mixins as functions' view satisfies all the expected properties. Finally, in Section 6 we summarise the contribution of the paper and outline further work.

This paper is meant to be a continuation of Ancona and Zucca (1998b); a preliminary version was presented in Ancona and Zucca (1998a).

2. A kernel language of mixin modules

In this section, in order to keep the paper self-contained, we provide a summary of our preceding work on mixin modules (Ancona and Zucca 1998b). In particular, Section 2.1 contains a brief informal introduction to mixin modules, in Section 2.2 we formally define the syntax and semantics of a kernel language, and in Section 2.3 we define three lower-level operators that can be used, as will be formally proved in Section 5, as primitive operators allowing us to define as derived forms all the operators of the kernel language. For an extended presentation, including more examples and discussions about the mixin notion and the various operators, we refer the reader to the preceding paper Ancona and Zucca (1998b).

https://doi.org/10.1017/S0960129502003687 Published online by Cambridge University Press

2.1. *Mixin modules*

Consider the following example definition of a mixin `M`, defined on top of a very simple functional language supporting basic types only:

```
mixin M =
 deferred leq:int*int→bool
 frozen eq(i1,i2:int):bool=leq(i1,i2) and leq(i2,i1)
 frozen lth(i1,i2:int):bool=not leq(i2,i1)
 lleq(i1,i2,j1,j2:int):bool=lth(i1,j1) or (eq(i1,j1) and leq(i2,j2))
 frozen llth(i1,i2,j1,j2:int):bool=not lleq(j1,j2,i1,i2)
end
```

As shown by the example, the components of a mixin module (functions in this case) are of three kinds: *deferred*, *virtual* (default case with no label) and *frozen*. All these components are visible to the outside; in addition, there can be *local* components, which are only used inside the module (see later examples).

Deferred components have no associated definition, but are intended to be provided by some other module (indeed a mixin with deferred components cannot be used in isolation). For instance, the semantics of all the functions defined in `M` depends on the deferred component `leq`; `M` could be *merged* with another module defining the `leq` component, thus obtaining a *concrete* module (a mixin with no deferred components), which can be used effectively.

On the other hand, both virtual and frozen components are defined in the module; however, there is a difference in the way calls to these components are interpreted:

— A call to a virtual component (for example, to `lleq` in the body of `llth`) is meant to be bound to the (possibly changing) definition of the virtual component. Hence, if there is a redefinition (for example, replacing the definition of `lleq` by a new definition), the behaviour of all other components that refer to the redefined component (`llth` in the example) changes, as happens for methods in object-oriented programming. This change of definition can be achieved by composing `M` with another module `M'` providing an alternative definition for `lleq` via the *overriding* operator (see later). Hence, in this case inlining of `lleq` in the body of `llth` would not yield a mixin equivalent to `M`.

— On the other hand, call to a frozen component (for example, all those in the body of `lleq`), is meant to be bound *forever* to the *current* definition. Hence, redefining a frozen component has no effect on other components. In other words, a mixin can always be reduced to an equivalent mixin with no calls to frozen components by replacing each call either by the corresponding definition or (this is the only possibility for mutual recursion) by a call to a local function defined exactly in the same way, as shown below:

```
mixin M =
 deferred leq:int*int→bool
 frozen eq(i1,i2:int):bool=leq(i1,i2) and leq(i2,i1)
 frozen lth(i1,i2:int):bool=not leq(i2,i1)
```

https://doi.org/10.1017/S0960129502003687 Published online by Cambridge University Press

```
 local eq_local(i1,i2:int):bool=leq(i1,i2) and leq(i2,i1)
 local lth_local(i1,i2:int):bool=not leq(i2,i1)
 lleq(i1,i2,j1,j2:int):bool=
  lth_local(i1,j1) or (eq_local(i1,j1) and leq(i2,j2))
 frozen llth(i1,i2,j1,j2:int):bool=not lleq(j1,j2,i1,i2)
end
```

In the following, we will always write mixins in this standard form with no calls to frozen components.

We will describe as *input* components all those components on which definitions in the module may depend (that is, the deferred and virtual components are input components); we will describe as *output* components all those components that are defined in the module (that is, the virtual and frozen components are *output* components). Hence, virtual components are those that are both input and output.

Since we want to abstract with respect to the nature of components in a specific language, we take an approach analogous to that taken in the theory of *institutions* (Goguen and Burstall 1992) for abstracting from a particular specification language. More precisely, we model a collection of (names and types of) components by a *signature*, and a possible interpretation for them by a *model* over this signature. Hence, the syntactic interface of a mixin module will be modelled abstractly by a pair of signatures $<\Sigma^{in}, \Sigma^{out}>$, called the *input* and *output* signature, respectively, whereas a mixin module over $<\Sigma^{in}, \Sigma^{out}>$ will be modelled by a function F that, for any given model over Σ^{in} for the input components, returns a model over Σ^{out} for the output components. Signatures are required to form a category **Sig**, whose concrete definition will depend on the underlying core language. Moreover, we require that there is a notion of *inclusion* between signatures with related operations of union, intersection and difference, which are the generalisations of the corresponding operations on sets (formally, we assume that signatures form a *boolean signature category*, see Definition 2.2 below).

In the example language, signatures are just sets of function symbols with their types; the input signature Σ^{in} is $\{\texttt{leq}\colon \texttt{int} * \texttt{int} \to \texttt{bool}, \texttt{lleq}\colon \texttt{int} * \texttt{int} * \texttt{int} * \texttt{int} \to \texttt{bool}\}$, whereas the output signature Σ^{out} is

$$\{\texttt{eq}\colon \texttt{int} * \texttt{int} \to \texttt{bool}, \texttt{lth}\colon \texttt{int} * \texttt{int} \to \texttt{bool},\ \texttt{lleq}\colon \texttt{int} * \texttt{int} * \texttt{int} * \texttt{int} \to \texttt{bool}, \texttt{llth}\colon \texttt{int} * \texttt{int} * \texttt{int} * \texttt{int} \to \texttt{bool}\}.$$

Hence, omitting types for simplicity, $\Sigma^{in} \setminus \Sigma^{out} = \{\texttt{leq}\}$ (deferred components), $\Sigma^{in} \cap \Sigma^{out} = \{\texttt{lleq}\}$ (virtual component) and $\Sigma^{out} \setminus \Sigma^{in} = \{\texttt{eq}, \texttt{lth}, \texttt{llth}\}$ (frozen components).

In this case, models over a signature are families of partial functions indexed over the function symbols of the signature; for instance, an example of model A over Σ^{in} is given by two partial functions $\texttt{leq}^A\colon \mathscr{Z} \times \mathscr{Z} \to \mathscr{B}$ and $\texttt{lleq}^A\colon \mathscr{Z} \times \mathscr{Z} \times \mathscr{Z} \times \mathscr{Z} \to \mathscr{B}$, where $\mathscr{Z}$ and $\mathscr{B}$ denote the sets of integer and boolean values, respectively.

Hence, a mixin module over $<\Sigma^{in}, \Sigma^{out}>$ is modelled by a functional F (that is, a function over families of functions) that, for any model over the input signature Σ^{in} (that is, a family of functions), returns a model over the output signature Σ^{out} (that is, another family of functions). In the example, for each model A^{in} over Σ^{in}, we obtain a model

https://doi.org/10.1017/S0960129502003687 Published online by Cambridge University Press

$A^{out} = F(A^{in})$ defined by:

$$\begin{aligned}
\mathtt{eq}^{A^{out}}(i_1, i_2) &= \mathtt{leq}^{A^{in}}(i_1, i_2) \wedge \mathtt{leq}^{A^{in}}(i_2, i_1) \\
\mathtt{lth}^{A^{out}}(i_1, i_2) &= \neg\mathtt{leq}^{A^{in}}(i_2, i_1) \\
\mathtt{lleq}^{A^{out}}(i_1, i_2, j_1, j_2) &= \mathtt{lth}^{A^{in}}(i_1, j_1) \vee (\mathtt{eq}^{A^{in}}(i_1, j_1) \wedge \mathtt{leq}^{A^{in}}(i_2, j_2)) \\
\mathtt{llth}^{A^{out}}(i_1, i_2, j_1, j_2) &= \neg\mathtt{lleq}^{A^{in}}(j_1, j_2, i_1, i_2)
\end{aligned}$$

Note that, as a matter of fact, the functional F keeps track of the dependency from the input (hence, both deferred and virtual) components. Our proposed model for mixin modules follows the same idea for modelling inheritance that was first adopted by W. Cook (Cook 1989) and U. S. Reddy (Reddy 1988).

2.2. *Typing and semantic rules*

We now define a set of operators for composing mixin modules, that is, a kernel language of mixins.

As we have already pointed out, this language is parametric, in the sense that its syntax and semantics depend on some ingredients that should be provided by the core language. These ingredients are formalised by the notion of *core framework*, given below.

Definition 2.1. A *signature category* is a category **Sig** with all finite colimits whose objects are called *signatures*. If Σ_1 and Σ_2 are two signatures, then we use $\Sigma_1 {}^{j_1}\!+^{j_2} \Sigma_2$ to denote the coproduct of Σ_1 and Σ_2 with injections j_1, j_2. We will omit the injections when they are clear from the context.

Definition 2.2. A *boolean signature category* is a pair $<$**Sig**, $\mathscr{I}>$ where **Sig** is a signature category and $\mathscr{I}$ is a subcategory of **Sig** with $|\mathscr{I}| = |$**Sig**$|$ and such that:

— $\mathscr{I}$ is a distributive lattice with bottom element (denoted $\varnothing$); we call the morphisms in $\mathscr{I}$ *inclusions* and use the notation $\Sigma_1 \subseteq \Sigma_2$ if there is an inclusion from Σ_1 into Σ_2, and denote this (unique) inclusion i_{Σ_1,Σ_2}. We call *union* (denoted $\Sigma_1 \cup \Sigma_2$) and *intersection* (denoted $\Sigma_1 \cap \Sigma_2$), respectively, the join and the meet of Σ_1 and Σ_2 in $\mathscr{I}$. For any morphism $\sigma : \Sigma_1 \to \Sigma_2$ and signature $\Sigma_1' \subseteq \Sigma_1$, we write $\sigma_{|\Sigma_1'}$ for the composition $\sigma \circ i_{\Sigma_1',\Sigma_1}$;

— for any $\Sigma_1, \Sigma_2 \in |$**Sig**$|$ there exists a signature Σ such that $\Sigma \cup \Sigma_1 = \Sigma_2 \cup \Sigma_1$ and $\Sigma \cap \Sigma_1 = \varnothing$. It is easy to show that such a signature (denoted $\Sigma_2 \setminus \Sigma_1$) is unique;

— $\varnothing$ is initial in **Sig** and for any $\Sigma_1, \Sigma_2 \in |$**Sig**$|$, we have $\Sigma_1 \hookrightarrow \Sigma_1 \cup \Sigma_2 \hookleftarrow \Sigma_2$ is a pushout for $\Sigma_1 \hookleftarrow \Sigma_1 \cap \Sigma_2 \hookrightarrow \Sigma_2$.

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

— **Sig** is a boolean signature category;

— *Mod* is a functor, *Mod* : **Sig**$^{op} \to$ **Set**, preserving finite colimits; for any signature Σ, objects in $Mod(\Sigma)$ are called *models* over Σ or Σ-*models*; for any signature morphism $\sigma : \Sigma \to \Sigma'$, $Mod(\sigma)$ is called the *reduct* via σ and denoted $-_{|\sigma}$;

— *fix* is a family of functions (that is, morphisms in **Set**)

$$fix_\Sigma : (Mod(\Sigma) \to Mod(\Sigma)) \to Mod(\Sigma)$$

https://doi.org/10.1017/S0960129502003687 Published online by Cambridge University Press

satisfying the following properties:

1 (Fix-point) for any $F : Mod(\Sigma) \to Mod(\Sigma)$,

$$F(fix_\Sigma(F)) = fix_\Sigma(F);$$

2 (Uniformity) for any $F : Mod(\Sigma) \to Mod(\Sigma)$, $F' : Mod(\Sigma') \to Mod(\Sigma')$ and for any $\sigma : \Sigma' \to \Sigma$, if $(F(A))_{|\sigma} = F'(A_{|\sigma})$ for any $A \in Mod(\Sigma)$, then

$$(fix_\Sigma(F))_{|\sigma} = fix_{\Sigma'}(F');$$

3 (Currying) for any $F : Mod(\Sigma) \times Mod(\Sigma) \to Mod(\Sigma)$,

$$fix_\Sigma(fix_\Sigma \circ \overline{F}) = fix_\Sigma(F \circ <id, id>),$$

where $\overline{F}$ and $<id, id>$ are the functions defined by

$$\overline{F}(A)(B) = F(A, B), <id, id>(A) = <A, A>,$$

for any $A, B \in Mod(\Sigma)$.

We will omit the subscript from fix_Σ whenever it can be determined unambigously from the context.

As we have already said, signatures provide an abstract notion of (names and types of) components (for example, sets of typed function names) and, correspondingly, models over a signature Σ provide an abstract notion of possible interpretations for components in Σ (for example, a partial function for each function name). These two components, as the reader can see, correspond, with some additional requirements, to the first two components of an institution (Goguen and Burstall 1992), that is, a category of signatures and a model functor. The third component provides an abstract notion of fixed point operator. As we have already mentioned, in our example language elements of $Mod(\Sigma)$ are families of partial functions; mixin modules are modelled by functions $F : Mod(\Sigma) \to Mod(\Sigma)$ that are expected to be continuous, whereas fix_Σ denotes the usual least fixed point operator. Property 1 reflects the intuition that each fix_Σ is actually a fixed point operator, while the two other properties express global coherency conditions over the family of *fix* operators.

From now on, we will assume a fixed core framework $<\mathbf{Sig}, Mod, fix>$.

For each $<j_1, j_2>$ coproduct of Σ_1, Σ_2 in **Sig**, A_1, A_2 models in $Mod(\Sigma_1)$, $Mod(\Sigma_2)$, respectively, we use $A_1 {}^{j_1}\!+\!{}^{j_2} A_2$ to denote the *amalgamated sum* of A_1 and A_2, that is, the unique model A over $\Sigma_1 + \Sigma_2$ such that $A_{|j_i} = A_i$, for $i = 1, 2$. The existence and unicity of A follows from the assumption that *Mod* preserves finite colimits (see Ancona (1998) for the proof). We omit the superscript from ${}^{j_1}\!+\!{}^{j_2}$ whenever it can be unambigously determined from the context.

We will now introduce the kernel language of mixin modules. Any expression M of the language has a type, modelling the interface of the module, which is a pair of signatures: we write $M : \Sigma^{in} \to \Sigma^{out}$ and the intended meaning is that $\Sigma^{in} \setminus \Sigma^{out}$, $\Sigma^{in} \cap \Sigma^{out}$ and $\Sigma^{out} \setminus \Sigma^{in}$ are the deferred, virtual and frozen components, respectively. The semantics of an expression M of type $M : \Sigma^{in} \to \Sigma^{out}$ will be a function from $Mod(\Sigma^{in})$ into $Mod(\Sigma^{out})$.

https://doi.org/10.1017/S0960129502003687 Published online by Cambridge University Press

For each operator, we give a typing rule specifying compatibility conditions between the types of the arguments, and the resulting type of the result, and a semantic rule formally expressing the meaning of the operator.

Merge This 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 are concreted by the definitions given in M_2, and *vice versa.* Two mixin modules can be merged together only if no components are defined in both (first side condition). The defined (output) components of $M_1 \oplus M_2$ are the (disjoint) union of those of M_1 and M_2. The input components are the union of those of M_1 and M_2, where components that were frozen in either M_1 or M_2 are cancelled: that eliminates components that were deferred in one argument and have been bound to a frozen component in the other. The semantics of $M_1 \oplus M_2$ corresponds to taking the union (with possible sharing) of the deferred components and the (disjoint) union of the definitions of M_1 and M_2:

$$\text{(M-ty)}\ \frac{M_i : \Sigma_i^{in} \to \Sigma_i^{out}, i = 1,2}{M_1 \oplus M_2 : \Sigma^{in} \setminus \Sigma^{fr} \to \Sigma_1^{out} \cup \Sigma_2^{out}} \quad \begin{array}{l} \Sigma_1^{out} \cap \Sigma_2^{out} = \varnothing \\ \Sigma^{fr}{}_i = \Sigma_i^{out} \setminus \Sigma_i^{in}, i = 1,2 \\ \Sigma^{in} = \Sigma_1^{in} \cup \Sigma_2^{in} \\ \Sigma^{fr} = \Sigma^{fr}{}_1 \cup \Sigma^{fr}{}_2 \end{array}$$

$$\text{(M-sem)}\ \frac{[\![M_i]\!] = F_i, i = 1,2}{[\![M_1 \oplus M_2]\!] = \lambda A.fix(\lambda B.F_1((A + B_{|\Sigma^{fr}})_{|\Sigma_1^{in}}) + F_2((A + B_{|\Sigma^{fr}})_{|\Sigma_2^{in}}))}$$

The semantic clause expresses the fact that the definitions in $M_1 \oplus M_2$ are obtained by taking the union of the definitions of M_1 and M_2; moreover, it is necessary to apply the *fix* operator in order to eliminate from the input signature the deferred components of one argument concreted by frozen components of the other.

Consider the following schematic example, where we use `G` to denote a function that maps values for `h` and `f` into a value for `g`, and analogously for `F`.

```
mixin M1 =
 deferred h
 deferred f
 frozen g = G(h,f)
end
mixin M2 =
 deferred h
 deferred g
 f = F(h,g,f)
end
```

Then, `M1` denotes the function that takes a model A_1^{in} assigning values $\mathtt{h}^{A_1^{in}}$ and $\mathtt{f}^{A_1^{in}}$ to `h` and `f`, respectively, and returns a model A_1^{out} assigning the value $\mathtt{G}(\mathtt{h}^{A_1^{in}}, \mathtt{f}^{A_1^{in}})$ to `g`. Analogously, `M2` denotes the function that takes a model A_2^{in} assigning values $\mathtt{h}^{A_2^{in}}$, $\mathtt{g}^{A_2^{in}}$ and $\mathtt{f}^{A_2^{in}}$ to `h`, `g` and `f`, respectively, and returns a model A_2^{out} assigning the value $\mathtt{F}(\mathtt{h}^{A_2^{in}}, \mathtt{g}^{A_2^{in}}, \mathtt{f}^{A_2^{in}})$ to `f`. Then, the mixin $M_1 \oplus M_2$ corresponds to the function defined by:

```
mixin
 deferred h
```

https://doi.org/10.1017/S0960129502003687 Published online by Cambridge University Press

```
 frozen g = G(h,f)
 f = F(h,G(h,f),f)
end
```

which takes a model A^{in} assigning values $\mathtt{h}^{A^{in}}$ and $\mathtt{f}^{A^{in}}$ to `h` and `f`, respectively, and returns a model A^{out} assigning the values $\mathtt{G}(\mathtt{h}^{A^{in}}, \mathtt{f}^{A^{in}})$ and $\mathtt{F}(\mathtt{h}^{A^{in}}, \mathtt{G}(\mathtt{h}^{A^{in}}, \mathtt{f}^{A^{in}}), \mathtt{f}^{A^{in}})$ to `g` and `f`, respectively.

Freeze This operator allows us to make a module independent from the redefinition of some components (Σ^{fr} in the typing rule); hence these components, if they were virtual, become frozen, that is, disappear from the input signature.

The semantics is given by means of the *fix* operator; the intuition is that all the components will refer from now on to the values of the Σ^{fr}-components as they are determined by the current definitions. Note that the only effect of the *freeze* operator is to switch the status of defined components from virtual to frozen; deferred and frozen components are unmodified.

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

$$\text{(F-sem)}\ \frac{[\![M]\!] = F}{[\![\mathbf{freeze}\ \Sigma^{fr}\ \mathbf{in}\ M]\!] = \lambda A.fix(\lambda B.F(A + B_{|\Sigma^{fr} \cap \Sigma^{in}}))}$$

For instance, we can transform the function `lleq` of the previously defined mixin `M` into a frozen function, obtaining a new mixin `freeze lleq in M` that is equivalent to the following:

```
mixin
 deferred leq:int*int→bool
 frozen eq(i1,i2:int):bool=leq(i1,i2) and leq(i2,i1)
 frozen lth(i1,i2:int):bool=not leq(i2,i1)
 local lleq_local(i1,i2,j1,j2:int):bool=
  lth(i1,j1) or (eq(i1,j1) and leq(i2,j2))
 frozen lleq(i1,i2,j1,j2:int):bool=lth(i1,j1) or (eq(i1,j1) and leq(i2,j2))
 frozen llth(i1,i2,j1,j2:int):bool=not lleq_local(j1,j2,i1,i2)
end
```

Hiding This operator allows us to hide some defined components (Σ^{hd} in the typing rule) from the outside: these components are cancelled from the output signature and (those that are virtual) from the input signature too. Hiding deferred components makes no sense since definitions of other components could depend on them. Hiding virtual components requires us first to apply the *fix* operator in such a way that all the other definitions will refer from now on to their current definitions.

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

$$\text{(H-sem)}\ \frac{[\![M]\!] = F}{[\![\mathbf{hide}\ \Sigma^{hd}\ \mathbf{in}\ M]\!] = \lambda A.(fix(\lambda B.F(A + B_{|\Sigma^{hd} \cap \Sigma^{in}})))_{|\Sigma^{out} \setminus \Sigma^{hd}}}$$

https://doi.org/10.1017/S0960129502003687 Published online by Cambridge University Press

Consider the following schematic example:

```
mixin M0 =
 deferred k
 f = F(f,k,h)
 h = H(f,k,h)
 frozen g = G(f,k,h)
end
```

Then, the mixin `hide f,g in M0` corresponds to the following:

```
mixin
 deferred k
 local f = F(f,k,h)
 h = H(f,k,h)
end
```

Restrict This operator allows us to 'throw away' some definitions (Σ^{rs} in the typing rule) in a module, that is, to cancel the corresponding components from the output signature. Restrict is different from hiding, since a virtual component whose definition is thrown away remains in the interface of the module as deferred and can be redefined later, while a hidden component becomes invisible from the outside; However, for frozen components the effect is the same as for hiding.

The semantic clause expresses the fact that some definitions are forgotten, hence the corresponding components are no longer in the output signature (this is formally expressed by the reduct functor):

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

$$\text{(R-sem)}\ \frac{[\![M]\!] = F}{[\![\textbf{restrict}\ \Sigma^{rs}\ \textbf{in}\ M]\!] = \lambda A.(F(A))_{|\Sigma^{out} \setminus \Sigma^{rs}}}$$

For instance, the module `restrict lth,lleq in M` is equivalent to the following:

```
mixin
 deferred leq:int*int→bool
 deferred lleq:int*int*int*int→bool
 frozen eq(i1,i2:int):bool=leq(i1,i2) and leq(i2,i1)
 frozen llth(i1,i2,j1,j2:int):bool=not lleq(j1,j2,i1,i2)
end
```

Overriding This operator allows us to combine two mixins with conflicting defined components, by overriding the definitions of M_1 by the corresponding definitions of M_2. Hence, the overriding operator coincides with the merge operator when there are no components defined in both mixins.

The typing rule is similar to that of the merge operator; however, there is no side condition requiring no conflict of definitions. As for merge, the defined (output) components of $M_1 \Leftarrow M_2$ are the union of those of M_1 and M_2, where components that were

https://doi.org/10.1017/S0960129502003687 Published online by Cambridge University Press

frozen in either M_1 or M_2 are cancelled. Note, however, that frozen components in M_1 are considered only if they are not defined in M_2 (first side condition), since in this case the definition in M_2 would take the precedence. The semantics is that $M_1 \Leftarrow M_2$ corresponds to taking the union (with possible sharing) of the deferred components and the union of definitions of M_1 and M_2, and choosing the definition in M_2 if there is a conflict.

$$\text{(O-ty)}\ \frac{M_i : \Sigma_i^{in} \to \Sigma_i^{out} \qquad i = 1,2}{M_1 \Leftarrow M_2 : (\Sigma_1^{in} \cup \Sigma_2^{in}) \setminus \Sigma^{fr} \to \Sigma_1^{out} \cup \Sigma_2^{out}} \quad \begin{array}{l} \Sigma^{fr}{}_1 = (\Sigma_1^{out} \setminus \Sigma_2^{out}) \setminus \Sigma_1^{in} \\ \Sigma^{fr}{}_2 = \Sigma_2^{out} \setminus \Sigma_2^{in} \\ \Sigma^{fr} = \Sigma^{fr}{}_1 \cup \Sigma^{fr}{}_2 \end{array}$$

$$\text{(O-sem)}\ \frac{[\![M_i]\!] = F_i, i = 1,2}{[\![M_1 \Leftarrow M_2]\!] = \lambda A.fix(\lambda B.(F_1((A + B_{|\Sigma^{fr}})_{|\Sigma_1^{in}}))_{|\Sigma_1^{out} \setminus \Sigma_2^{out}} + F_2((A + B_{|\Sigma^{fr}})_{|\Sigma_2^{in}}))}$$

Consider the two mixins `M1` and `M2` defined schematically as follows:

```
mixin M1 =
 deferred h
 f = F1(f,h)
 frozen g = G(f,h)
end
mixin M2 =
 deferred h
 deferred g
 frozen f = F2(h,g)
end
```

Then, the mixin $M_1 \Leftarrow M_2$ corresponds to the following:

```
mixin
deferred h
frozen g = G(f,h)
frozen f = F2(h,g)
end
```

Functional Composition This operator is a generalisation of the application of parameterised modules, like ML functors; here the formal parameters are the deferred components $\Sigma_2^{in} \setminus \Sigma_2^{out}$ of M_2, whereas the actual parameters are the defined components Σ_1^{out} of M_1. Hence, these components must coincide (first side condition). The second side condition is needed to avoid confusion between deferred components of M_1 and components of M_2: these components must be distinct, since otherwise they would result virtual in $M_2 \circ M_1$. Input components of $M_2 \circ M_1$ are deferred components of M_1 and virtual components of M_2; output components of $M_2 \circ M_1$ are those of M_2. The semantics is expressed in terms of the *fix* operator, in order to handle the virtual components of M_1 correctly.

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

$$\text{(FC-sem)}\ \frac{[\![M_i]\!] = F_i, i = 1,2}{[\![M_2 \circ M_1]\!] = \lambda A.F_2(fix(\lambda B.F_1(A_{|\Sigma_1^{in} \setminus \Sigma_1^{out}} + B_{|\Sigma_1^{in} \cap \Sigma_1^{out}})) + A_{|\Sigma_2^{in} \cap \Sigma_2^{out}})}$$

https://doi.org/10.1017/S0960129502003687 Published online by Cambridge University Press

As an example, consider the two mixins `M1` and `M2` defined schematically as follows:

```
mixin M1 =
 deferred h
 g = G(g,h)
end
mixin M2 =
 deferred g
 f = F(f,g)
end
```

Then, the mixin $M_2 \circ M_1$ corresponds to the following:

```
mixin
deferred h
local g = G(g,h)
f = F(f,g)
end
```

Constants Constant operators, corresponding to basic modules like those shown in the preceding examples, cannot be fixed once and for all in the kernel language, since they obviously depend on the particular core language we are considering, notably on the particular form taken by signatures. In the example language, where, as we have already said, signatures are just sets of typed function symbols, a constant has the general form:

mixin
deferred $D_1 : \tau_1^D, \ \ldots, \ D_k : \tau_k^D$
frozen $F_1 : \tau_1^F = e_1^F, \ \ldots, \ F_l : \tau_l^F = e_l^F$
local $L_1 : \tau_1^L = e_1^L, \ \ldots, \ L_m : \tau_m^L = e_m^L$
$V_1 : \tau_1^V = e_1^V, \ \ldots, \ V_n : \tau_n^V = e_n^V$
end

where the decorated τ are types of the core language (in this case functional types constructed over `int` and `bool`) and the decorated e are well-typed expressions of the core language, possibly containing the names of the module components. More precisely, each e_i^F (respectively, e_i^L, e_i^V) must be a correct core expression of type τ_i^F (respectively, τ_i^L, τ_i^V) where each D_i (respectively, F_i, L_i, V_i) is used as a variable of type τ_i^D (respectively, τ_i^F, τ_i^L, τ_i^V).

Under these assumptions, a constant as above has type

$$\{D_1 : \tau_1^D, \ldots, D_k : \tau_k^D, V_1 : \tau_1^V, \ldots, V_n : \tau_n^V\} \rightarrow \{V_1 : \tau_1^V, \ldots V_n : \tau_n^V, F_1 : \tau_1^F, \ldots F_l : \tau_l^F\}.$$

Note that, as expected, local components do not appear in the module interface.

2.3. *Primitive operators*

In the preceding subsection we have defined a kernel language of mixin modules and its denotational semantics, based on the idea of interpreting a module as a function from input into output components. Actually, all the operators we have presented can be

https://doi.org/10.1017/S0960129502003687 Published online by Cambridge University Press

expressed in terms of three lower-level (families of) operators (as will be formally proved in Section 5) with simple semantics, which correspond to three very primitive ways of manipulating modules: *sum* (corresponding to assembling together two modules), *reduct* (corresponding to renaming both input and output components) and *primitive freeze* (corresponding to connecting some input to some output component). These operators are also lower-level in the sense that they can be formally defined assuming that **Sig** is just a signature category (that is, they do not require any notion of inclusion between signatures).

Sum The sum operators are indexed over coproducts $<j_1, j_2>$ in **Sig**.

$$\text{(Sum-ty)}\ \frac{M_1 : \Sigma^{in} \to \Sigma_1^{out} \qquad M_2 : \Sigma^{in} \to \Sigma_2^{out} \qquad \begin{array}{l} j_1 : \Sigma_1^{out} \to \Sigma_1^{out} + \Sigma_2^{out} \\ j_2 : \Sigma_2^{out} \to \Sigma_1^{out} + \Sigma_2^{out} \end{array}}{M_1 {}^{j_1}\!+\!{}^{j_2} M_2 : \Sigma^{in} \to \Sigma_1^{out} + \Sigma_2^{out}}$$

$$\text{(Sum-sem)}\ \frac{[\![M_i]\!] = F_i, i = 1, 2}{[\![M_1 {}^{j_1}\!+\!{}^{j_2} M_2]\!] = \lambda A.F_1(A)^{j_1}\!+\!{}^{j_2} F_2(A)}$$

By means of these operators it is possible to combine pairs of mixin modules having the same input signature, but different output signatures. The coproducts specify how output signatures are combined together. As an example, consider the mixins `M1` and `M2` defined schematically as follows.

```
mixin M1 =
 deferred h
 f = F1(h,f)
end
mixin M2 =
 deferred h
 f = F2(h,f)
 g = G(h,f)
end
```

If $j_1 : \{\mathtt{f}\} \to \{\mathtt{f1}, \mathtt{f2}, \mathtt{g}\}$ and $j_2 : \{\mathtt{f}, \mathtt{g}\} \to \{\mathtt{f1}, \mathtt{f2}, \mathtt{g}\}$ are defined by $j_1(\mathtt{f}) = \mathtt{f1}$, $j_2(\mathtt{f}) = \mathtt{f2}$, $j_2(\mathtt{g}) = \mathtt{g}$, then the sum $\mathtt{M1}^{j_1}\!+\!{}^{j_2}\mathtt{M2}$ is given by:

```
mixin
 deferred h
 f1 = F1(h,f)
 f2 = F2(h,f)
 g = G(h,f)
end
```

All the input components are shared, while the output components are kept distinct (as happens for the two components `f`). Intuitively, the sum operator represents the most primitive way of combining together two mixins and is the natural extension of the amalgamated sum over models in the core framework (in Ancona and Zucca (1998b) we proved that *MixMod* is a model functor in the usual sense and that this operator is indeed an amalgamated sum for mixin models).

https://doi.org/10.1017/S0960129502003687 Published online by Cambridge University Press

Reduct The reduct operators are indexed over pairs $<\sigma^{in}, \sigma^{out}>$ of morphisms in **Sig**.

$$\text{(Reduct-ty)}\ \frac{M : \Sigma^{in} \to \Sigma^{out}}{{}_{\sigma^{in}|}M_{|\sigma^{out}} : \Sigma'^{in} \to \Sigma'^{out}} \qquad \begin{array}{l}\sigma^{in} : \Sigma^{in} \to \Sigma'^{in} \\ \sigma^{out} : \Sigma'^{out} \to \Sigma^{out}\end{array}$$

$$\text{(Reduct-sem)}\ \frac{[\![M]\!] = F}{[\![{}_{\sigma^{in}|}M_{|\sigma^{out}}]\!] = \lambda A.(F(A_{|\sigma^{in}}))_{|\sigma^{out}}}$$

Intuitively, reduct corresponds to a powerful form of renaming where the input and output components can be renamed separately via σ^{in} and σ^{out}, respectively. As an example, consider the mixin M schematically defined as follows:

```
mixin M =
 deferred h
 f = F(f,g,h)
 g = G(f,g,h)
end
```

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:

```
mixin
 deferred z
 deferred x
 frozen y = F(x,x,x)
end
```

Note that through the reduct it is possible to add dummy input component (like `z`) and to forget output components (like `g`). Again, this operator is the natural extension of the corresponding operator at the level of the core framework (in Ancona and Zucca (1998b) we proved that this operator is indeed the reduct in the usual sense for the model functor *MixMod*).

Primitive Freeze Primitive freeze operators are indexed over pairs $<<j_1, j_2>, \sigma^{fr}>$ with $<j_1, j_2>$ coproduct and σ^{fr} morphism in **Sig**, respectively, satisfying the side condition in the typing rule.

$$\text{(Prim-Freeze-ty)}\ \frac{M : \Sigma^{in} + \Sigma^{fr} \to \Sigma^{out}}{freeze^{j_1, j_2}_{\sigma^{fr}}(M) : \Sigma^{in} \to \Sigma^{out}} \qquad \begin{array}{l}\sigma^{fr} : \Sigma^{fr} \to \Sigma^{out} \\ j_1 : \Sigma^{in} \to \Sigma^{in} + \Sigma^{fr} \\ j_2 : \Sigma^{fr} \to \Sigma^{in} + \Sigma^{fr}\end{array}$$

$$\text{(Prim-Freeze-sem)}\ \frac{[\![M]\!] = F}{[\![freeze^{j_1, j_2}_{\sigma^{fr}}(M)]\!] = \lambda A.fix(\lambda B.F(A^{j_1 + j_2} B_{|\sigma^{fr}}))}$$

Primitive freeze operators are needed for getting rid of input components in mixin modules; this can be achieved by associating with each input component to be eliminated a defined component in the same mixin module. The morphism σ^{fr} specifies this association, whereas the coproduct $<j_1, j_2>$ shows how the input components are decomposed into those to be frozen and the rest. As an example, let `M` be the previous mixin and let σ be the signature morphism mapping f and g into f, with j_1, j_2 the obvious injections.

https://doi.org/10.1017/S0960129502003687 Published online by Cambridge University Press

Then, $freeze_{\sigma}^{j_1,j_2}(\mathtt{M})$ is given by:

```
mixin
 deferred h
 local f' = F(f',g',h)
 local g' = F(f',g',h)
 frozen f = f'
 frozen g = g'
end
```

The freeze operator defined in Section 2.2 is a higher level version of the primitive freeze operator where σ, j_1, j_2 are specified implicitly by selecting the signature of the components to be frozen, as will be formally shown in Section 3.3.

3. An axiomatic definition for mixin modules

In the preceding section, we have formally defined the syntax and semantics of a kernel language of mixin modules (parametric in the underlying core language). Our aim now is to provide an alternative characterisation of this language in a purely axiomatic way (in the spirit of the seminal paper Bergstra *et al.* (1990)), that is, to define a theory of mixin modules by means of a specification in many-sorted (positive) conditional logic and to derive from this specification a reduction semantics enjoying a normalisation property.

We will present this axiomatisation in two steps. In Section 3.1, we define a specification SP in many-sorted conditional logic providing a minimal characterisation of the three primitive operators defined in Section 2.3. Here 'minimal' should be understood in the sense that these axioms are sufficient and necessary for proving a normal form theorem, that is, that all terms can be reasonably simplified; moreover, some of them (notably those for sum and reduct) corresponds to stating that the operator actually corresponding to the natural extension of the corresponding operator at the core level. In Section 3.2 we show that many equational laws that we intuitively expect to hold for mixin modules can be proved in SP. In Section 5 we will prove that the interpretation previously defined for the three basic operators actually satisfies this specification.

Then, in Section 3.3, we state a further set of axioms that express each operator of the language in terms of the three primitive operators. In this way, the normalisation property (which will be proved in Section 4) obviously applies to the full language. We will, again, prove in Section 5 that the interpretation previously defined for the operators actually satisfies this specification, that is, that their definition as derived operators is sound.

3.1. *Axioms for primitive operators*

We will now show the specification SP providing an axiomatic characterisation of the three primitive operators. Note that the specification is presented in a highly schematic way and not in any real specification language. Moreover, recall that there are no constant mixins, since the choice of constants will depend on the particular core language, as exemplified in Section 2.2; furthermore, in general, the given specification is not intended to completely

https://doi.org/10.1017/S0960129502003687 Published online by Cambridge University Press

spec *SP*

sorts $\{mix(\Sigma^{in}, \Sigma^{out}) \mid \Sigma^{in}, \Sigma^{out}$ signatures in **Sig**$\}$

opns

$\{\_ {}^{j_1}\!+\!{}^{j_2}\_ : mix(\Sigma^{in}, \Sigma_1^{out}), mix(\Sigma^{in}, \Sigma_2^{out}) \to mix(\Sigma^{in}, \Sigma_1^{out\,j_1}\!+\!{}^{j_2}\,\Sigma_2^{out})$
$\mid <j_1, j_2>$ coproduct in **Sig**$\}$

$\{{}_{-\sigma^{in}|}\_{}_{|\sigma^{out}} : mix(\Sigma^{in}, \Sigma^{out}) \to mix(\Sigma'^{in}, \Sigma'^{out})$
$\mid \sigma^{in} : \Sigma^{in} \to \Sigma'^{in}, \sigma^{out} : \Sigma'^{out} \to \Sigma^{out}\}$

$\{\_ freeze_{\sigma^{fr}}^{j_1,j_2} : mix(\Sigma^{in\,j_1}\!+\!{}^{j_2}\,\Sigma^{fr}, \Sigma^{out}) \to mix(\Sigma^{in}, \Sigma^{out})$
$\mid \sigma^{fr} : \Sigma^{fr} \to \Sigma^{out}, <j_1, j_2>$ coproduct in **Sig**$\}$

axioms

(1) $\forall M_1 : mix(\Sigma^{in}, \Sigma_1^{out}), M_2 : mix(\Sigma^{in}, \Sigma_2^{out})\ (M_1\,{}^{j_1}\!+\!{}^{j_2}\,M_2)_{|j_1} = M_1$

(2) $\forall M_1 : mix(\Sigma^{in}, \Sigma_1^{out}), M_2 : mix(\Sigma^{in}, \Sigma_2^{out})\ (M_1\,{}^{j_1}\!+\!{}^{j_2}\,M_2)_{|j_2} = M_2$

(3) $\forall M_1, M_2 : mix(\Sigma^{in}, \Sigma_1^{out} + \Sigma_2^{out})\ M_{1|j_1} = M_{2|j_1} \wedge M_{1|j_2} = M_{2|j_2} \Rightarrow M_1 = M_2$

(4) $\forall M : mix(\Sigma^{in}, \Sigma^{out})\ {}_{id_{\Sigma^{in}}|}M_{|id_{\Sigma^{out}}} = M$

(5) $\forall M : mix(\Sigma^{in}, \Sigma^{out})\ {}_{\sigma_2^{in}|\sigma_1^{in}|}M_{|\sigma_1^{out}|\sigma_2^{out}} = {}_{\sigma_2^{in}\circ\sigma_1^{in}|}M_{|\sigma_1^{out}\circ\sigma_2^{out}}$

(6) $\forall M : mix((\Sigma^{in} + \Sigma_2^{fr}) + \Sigma_1^{fr}, \Sigma^{out})$
$freeze_{\sigma_2^{fr}}^{k_1,k_2}(freeze_{\sigma_1^{fr}}^{j_1,j_2}(M)) = freeze_{[\sigma_2^{fr},\sigma_1^{fr}]}^{l_1,l_2}([l_1,l_2\circ m_1,l_2\circ m_2]|M)$

(7) $\forall M : mix(\Sigma^{in} + \emptyset, \Sigma^{out})\ freeze_{\emptyset_{\Sigma^{out}}}^{j_1,j_2}(M) = {}_{[id_{\Sigma^{in}},\emptyset_{\Sigma^{in}}]|}M$

(8) $\forall M : mix(\Sigma^{in} + \Sigma^{fr}, \Sigma^{out})\ freeze_{\sigma^{fr}}^{j_1,j_2}(M_{|\sigma^{out}}) = (freeze_{\sigma'^{fr}}^{k_1,k_2}({}_{id_{\Sigma^{in}}+\sigma^{in}|}M))_{|\sigma^{out}}$

(9) $\forall M : mix(\Sigma^{in} + \Sigma^{fr}, \Sigma^{out})\ {}_{\sigma^{in}|}(freeze_{\sigma^{fr}}^{j_1,j_2}(M)) = freeze_{\sigma'^{fr}}^{k_1,k_2}({}_{\sigma^{in}+\sigma'^{in}|}M)$

(10) $\forall M_1 : mix(\Sigma^{in} + \Sigma_1^{fr}, \Sigma_1^{out}), M_2 : mix(\Sigma^{in} + \Sigma_2^{fr}, \Sigma_2^{out})$
$freeze_{\sigma_1^{fr}}^{l_1,l_2}(M_1)^{j_1}\!+\!{}^{j_2}freeze_{\sigma_2^{fr}}^{m_1,m_2}(M_2) = freeze_{\sigma_1^{fr}+\sigma_2^{fr}}^{p_1,p_2}([p_1,p_2\circ k_1]|M_1\,{}^{j_1}\!+\!{}^{j_2}\,{}_{[p_1,p_2\circ k_2]|}M_2)$

Fig. 1. Definition of the specification *SP*

characterise the operators, but only to state their very general properties (as explained above), hence it is a *requirement* specification rather than a *design* specification, following the usual terminology (see, for example, Astesiano *et al.* (1999)).

The signature Σ_{SP} of *SP* (see Figure 1) is parametric in the signature category **Sig** of the core framework. The sort symbols are indexed over pairs of signatures and have the form $mix(\Sigma^{in}, \Sigma^{out})$, therefore the set of sorts may be infinite. Intuitively, each element of sort $mix(\Sigma^{in}, \Sigma^{out})$ corresponds to a mixin module whose input and output components are specified by Σ^{in} and Σ^{out}, called the *input* and *output* signatures, respectively.

Some of the operation symbols of the signature are indexed over morphisms in **Sig**, so, as happens for sorts, they may be infinite. However, the operation symbols are partitioned into three classes, corresponding to the three primitive (families of) operators introduced in Section 2.3.

— *Sum.* Sum operators have the form ${}^{j_1}\!+\!{}^{j_2}$ for all coproducts $<j_1, j_2>$ in **Sig**. Note that we also use the same symbol + at the level of signatures to denote the coproduct object. Note also that each symbol ${}^{j_1}\!+\!{}^{j_2}$ is (possibly) overloaded, since the types of arguments and result contain the signature Σ^{in}, which does not depend on the choice of the coproduct $<j_1, j_2>$; however, the operator that is actually applied can always be determined from the types of the arguments.

https://doi.org/10.1017/S0960129502003687 Published online by Cambridge University Press

— *Reduct*. Reduct operators have the form ${}_{\sigma^{in}|-|\sigma^{out}}$ indexed for all pairs of morphisms σ^{in} and σ^{out} in **Sig**.
— *Primitive freeze*. Primitive freeze operators have the form $freeze^{j_1,j_2}_{\sigma^{fr}}$ for all morphisms $\sigma^{fr}:\Sigma^{fr} \to \Sigma^{out}$ and coproducts $<j_1, j_2>$ of Σ^{in} and Σ^{fr} in **Sig**.

Like sorts and operations symbols, axioms in Figure 1 are indexed over signatures and morphisms in **Sig**; hence they are more appropriately called axiom schemas.

To give a better understanding of the axioms, we adopt the following precedence rules: reduct operations have a higher priority than sum operations and sum operations are left associative. Finally, in the axioms, and later, we use the following abbreviations: if $M:mix(\Sigma^{in}, \Sigma^{out})$, $\sigma^{out}:\Sigma'^{out} \to \Sigma^{out}$ and $\sigma^{in}:\Sigma^{in} \to \Sigma'^{in}$, then ${}_{\sigma^{in}|}M$ stands for ${}_{\sigma^{in}|}M_{|id_{\Sigma^{out}}}$ and $M_{|\sigma^{out}}$ stands for ${}_{id_{\Sigma^{in}}|}M_{|\sigma^{out}}$.

We describe now in detail the axioms related to each primitive operators; each group of axioms is repeated below to help the reader.

(1) $\forall M_1:mix(\Sigma^{in}, \Sigma_1^{out}), M_2:mix(\Sigma^{in}, \Sigma_2^{out})\ (M_1 {}^{j_1}\!+^{j_2} M_2)_{|j_1} = M_1.$
(2) $\forall M_1:mix(\Sigma^{in}, \Sigma_1^{out}), M_2:mix(\Sigma^{in}, \Sigma_2^{out})\ (M_1 {}^{j_1}\!+^{j_2} M_2)_{|j_2} = M_2.$
(3) $\forall M_1, M_2:mix(\Sigma^{in}, \Sigma_1^{out} + \Sigma_2^{out})\ M_{1|j_1} = M_{2|j_1} \wedge M_{1|j_2} = M_{2|j_2} \Rightarrow M_1 = M_2.$

Axiom schemas (1), (2) and (3) ensure that the sum operators enjoy the amalgamation property (Ehrig and Mahr 1985); each schema produces a single axiom for each signature Σ^{in} and coproduct $<j_1, j_2>$ of Σ_1^{out} and Σ_2^{out}.

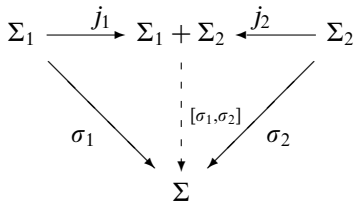
(4) $\forall M:mix(\Sigma^{in}, \Sigma^{out})\ {}_{id_{\Sigma^{in}}|}M_{|id_{\Sigma^{out}}} = M$
(5) $\forall M:mix(\Sigma^{in}, \Sigma^{out})\ {}_{\sigma_2^{in}|\sigma_1^{in}|}M_{|\sigma_1^{out}|\sigma_2^{out}} = {}_{\sigma_2^{in}\circ\sigma_1^{in}|}M_{|\sigma_1^{out}\circ\sigma_2^{out}}$

Axiom schemas (4) and (5) express the functoriality of the reduct operators; schema (4) has to be instantiated over each pair of identity morphisms $id_{\Sigma^{in}}$, $id_{\Sigma^{out}}$; schema (5) over morphisms $\sigma_1^{in}:\Sigma^{in} \to \Sigma_1^{in}$, $\sigma_2^{in}:\Sigma_1^{in} \to \Sigma_2^{in}$, $\sigma_1^{out}:\Sigma_1^{out} \to \Sigma^{out}$ and $\sigma_2^{out}:\Sigma_2^{out} \to \Sigma_1^{out}$.

(6) $\forall M:mix((\Sigma^{in} + \Sigma_2^{fr}) + \Sigma_1^{fr}, \Sigma^{out})$
$freeze^{k_1,k_2}_{\sigma_2^{fr}}(freeze^{j_1,j_2}_{\sigma_1^{fr}}(M)) = freeze^{l_1,l_2}_{[\sigma_2^{fr},\sigma_1^{fr}]}({}_{[l_1,l_2\circ m_1,l_2\circ m_2]|}M)$

Axiom schema (6) shows how composition of primitive freeze operators works; it holds for each morphism $\sigma_1^{fr}:\Sigma_1^{fr} \to \Sigma^{out}$, $\sigma_2^{fr}:\Sigma_2^{fr} \to \Sigma^{out}$ and coproducts $<j_1, j_2>$ of $\Sigma^{in} + \Sigma_2^{fr}$ and Σ_1^{fr}, $<k_1, k_2>$ of Σ^{in} and Σ_2^{fr}, $<l_1, l_2>$ of Σ^{in} and $\Sigma_2^{fr} + \Sigma_1^{fr}$ and $<m_1, m_2>$ of Σ_2^{fr} and Σ_1^{fr}.

If $\sigma_1:\Sigma_1 \to \Sigma$ and $\sigma_2:\Sigma_2 \to \Sigma$ are two signature morphisms, and $<j_1, j_2>$ is a coproduct of Σ_1 and Σ_2, we use $[\sigma_1, \sigma_2]$ to denote the unique morphism from $\Sigma_1 + \Sigma_2$ to Σ making the following diagram commute:



Hence, $[\sigma_2^{fr}, \sigma_1^{fr}]$ denotes the unique morphism h such that $h \circ m_1 = \sigma_2^{fr}$, $h \circ m_2 = \sigma_1^{fr}$;

https://doi.org/10.1017/S0960129502003687 Published online by Cambridge University Press

we will use this notation from now on†. Extending this notation to general coproducts (Adàmek *et al.* 1990, page 170), we have that $[l_1, l_2 \circ m_1, l_2 \circ m_2]$ denotes the unique morphism h such that $h \circ (j_1 \circ k_1) = l_1$, $h \circ (j_1 \circ k_2) = l_2 \circ m_1$ and $h \circ j_2 = l_2 \circ m_2$; indeed, the triple $<j_1 \circ k_1, j_1 \circ k_2, j_2>$ is a coproduct of Σ^{in}, Σ_2^{fr} and Σ_1^{fr}. Note that $[l_1, l_2 \circ m_1, l_2 \circ m_2]$ can be written equivalently as $[[l_1, l_2 \circ m_1], l_2 \circ m_2]$.

(7) $\quad \forall M : mix(\Sigma^{in} + \varnothing, \Sigma^{out})\ freeze^{j_1,j_2}_{\varnothing_{\Sigma^{out}}}(M) = {}_{[id_{\Sigma^{in}}, \varnothing_{\Sigma^{in}}]|}M$

Axiom schema (7) states that primitive freeze is the identity whenever we consider the unique morphism $\varnothing_{\Sigma^{out}}$ from any initial object $\varnothing$ to Σ^{out}. It holds for any morphism $\varnothing_{\Sigma^{out}}$ and coproduct $<j_1, j_2>$ of Σ^{in} and $\varnothing$; the morphism $[id_{\Sigma^{in}}, \varnothing_{\Sigma^{in}}]$ is determined by the coproduct $<j_1, j_2>$, whereas $\varnothing_{\Sigma^{in}}$ denotes the unique morphism from $\varnothing$ to Σ^{in}.

(8) $\quad \forall M : mix(\Sigma^{in} + \Sigma^{fr}, \Sigma^{out})\ freeze^{j_1,j_2}_{\sigma^{fr}}(M_{|\sigma^{out}}) = (freeze^{k_1,k_2}_{\sigma'^{fr}}({}_{id_{\Sigma^{in}} + \sigma^{in}|}M))_{|\sigma^{out}}$

(9) $\quad \forall M : mix(\Sigma^{in} + \Sigma^{fr}, \Sigma^{out})\ {}_{\sigma^{in}|}(freeze^{j_1,j_2}_{\sigma^{fr}}(M)) = freeze^{k_1,k_2}_{\sigma'^{fr}}({}_{\sigma^{in} + \sigma'^{in}|}M)$

Axiom schemas (8) and (9) describe how the primitive freeze operators behave with respect to the reduct operators.

Schema (8) holds for each morphism

$$\sigma^{fr} : \Sigma^{fr} \to \Sigma'^{out}, \quad \sigma'^{fr} : \Sigma'^{fr} \to \Sigma^{out}, \quad \sigma^{in} : \Sigma^{fr} \to \Sigma'^{fr}, \quad \sigma^{out} : \Sigma'^{out} \to \Sigma^{out}$$

such that $\sigma'^{fr} \circ \sigma^{in} = \sigma^{out} \circ \sigma^{fr}$, and coproduct $<j_1, j_2>$ of Σ^{in} and Σ^{fr} and $<k_1, k_2>$ of Σ^{in} and Σ'^{fr}.

If $\sigma_1 : \Sigma_1 \to \Sigma'_1$ and $\sigma_2 : \Sigma_2 \to \Sigma'_2$ are two signature morphisms, we use $\sigma_1 + \sigma_2$ to denote the morphism $[j'_1 \circ \sigma_1, j'_2 \circ \sigma_2] : \Sigma_1 + \Sigma_2 \to \Sigma'_1 + \Sigma'_2$, where j'_1 and j'_2 are the injections of the coproduct $\Sigma'_1 + \Sigma'_2$.

$$\begin{array}{ccccc} \Sigma_1 & \xrightarrow{j_1} & \Sigma_1 + \Sigma_2 & \xleftarrow{j_2} & \Sigma_2 \\ \downarrow{\scriptstyle \sigma_1} & & \vdots{\scriptstyle \sigma_1 + \sigma_2} & & \downarrow{\scriptstyle \sigma_2} \\ \Sigma'_1 & \xrightarrow{j'_1} & \Sigma'_1 + \Sigma'_2 & \xleftarrow{j'_2} & \Sigma'_2 \end{array}$$

Therefore, $id_{\Sigma^{in}} + \sigma^{in}$ denotes the morphism $[k_1 \circ id_{\Sigma^{in}}, k_2 \circ \sigma^{in}]$ determined by $<j_1, j_2>$ and $<k_1, k_2>$; we will use this notation from now on; the same consideration made for the notation $[\_, \_]$ applies here also.

Schema (9) holds for each morphism

$$\sigma^{fr} : \Sigma^{fr} \to \Sigma^{out}, \quad \sigma'^{fr} : \Sigma'^{fr} \to \Sigma^{out}, \quad \sigma^{in} : \Sigma^{in} \to \Sigma'^{in}, \quad \sigma'^{in} : \Sigma^{fr} \to \Sigma'^{fr}$$

such that $\sigma'^{fr} \circ \sigma'^{in} = \sigma^{fr}$ and coproduct $<j_1, j_2>$ of Σ^{in} and Σ^{fr} and $<k_1, k_2>$ of Σ'^{in} and Σ'^{fr}. Analogously to schema (8), the morphism $\sigma^{in} + \sigma'^{in}$ is determined by $<j_1, j_2>$ and $<k_1, k_2>$.

† Actually, in order to avoid ambiguities, the notation should also reveal the two injections of the coproduct, as in $[\sigma_2^{fr}, \sigma_1^{fr}]^{m2,m1}$; however we will avoid this heavier notation, specifying explicitly the two injections whenever they cannot be easily deduced from the context.

https://doi.org/10.1017/S0960129502003687 Published online by Cambridge University Press

(11) $\forall M_1 : mix(\Sigma^{in}, \Sigma_1^{out}), M_2 : mix(\Sigma^{in}, \Sigma_2^{out})\ M_{1|\sigma_1^{out}}{}^{k_1}+^{k_2} M_{2|\sigma_2^{out}} = (M_1{}^{j_1}+^{j_2} M_2)_{|\sigma_1^{out}+\sigma_2^{out}}$

(12) $\forall M : mix(\Sigma^{in}, \Sigma^{out})\ M_{|\sigma_1^{out}}{}^{j_1}+^{j_2} M_{|\sigma_2^{out}} = M_{|[\sigma_1^{out}, \sigma_2^{out}]}$

(13) $\forall M_1 : mix(\Sigma^{in}, \Sigma_1^{out}), M_2 : mix(\Sigma^{in}, \Sigma_2^{out})\ {}_{\sigma^{in}|}(M_1{}^{j_1}+^{j_2} M_2) = {}_{\sigma^{in}|}M_1{}^{j_1}+^{j_2}{}_{\sigma^{in}|}M_2$

(14) $\forall M_1 : mix(\Sigma^{in}, \Sigma_1^{out}), M_2 : mix(\Sigma^{in}, \Sigma_2^{out})\ M_1{}^{j_1}+^{j_2} M_2 = M_2{}^{j_2}+^{j_1} M_1$

(15) $\forall M_1 : mix(\Sigma^{in}, \Sigma_1^{out}), M_2 : mix(\Sigma^{in}, \Sigma_2^{out}), M_3 : mix(\Sigma^{in}, \Sigma_3^{out})$
$(M_1{}^{j_1}+^{j_2} M_2)^{l_1}+^{l_2} M_2 = M_1{}^{m_1}+^{m_2}(M_2{}^{p_1}+^{p_2} M_3)$

(16) $\forall M_1 : mix(\Sigma^{in}, \Sigma_1^{out}), M_2 : mix(\Sigma^{in}, \Sigma_2^{out})\ M_1{}^{id_{\Sigma_1^{out}}}+^{\varnothing_{\Sigma_1^{out}}} M_{2|\varnothing_{\Sigma_1^{out}}} = M_1$

(17) $\forall M_1 : mix(\Sigma^{in}, \Sigma_1^{out}), M_2 : mix(\Sigma^{in}, \Sigma_2^{out})\ M_{1|\varnothing_{\Sigma_1^{out}}} = M_{2|\varnothing_{\Sigma_2^{out}}}$

(18) $\forall M_1 : mix(\Sigma^{in}, \Sigma_1^{out}), M_2 : mix(\Sigma^{in}, \Sigma_2^{out})\ M_1{}^{k_1}+^{k_2} M_2 = (M_1{}^{j_1}+^{j_2} M_2)_{|[j_1, j_2]}$

(19) $\forall M : mix(\Sigma^{in} + \Sigma^{fr}, \Sigma^{out})\ freeze_{\sigma^{fr}}^{j_1, j_2}(M) = freeze_{\sigma'^{fr}}^{k_1, k_2}(_{id_{\Sigma^{in}}+\sigma|}M)$

Fig. 2. Derived laws for specification SP.

(10) $\forall M_1 : mix(\Sigma^{in} + \Sigma_1^{fr}, \Sigma_1^{out}), M_2 : mix(\Sigma^{in} + \Sigma_2^{fr}, \Sigma_2^{out})$
$freeze_{\sigma_1^{fr}}^{l_1, l_2}(M_1)^{j_1}+^{j_2} freeze_{\sigma_2^{fr}}^{m_1, m_2}(M_2) = freeze_{\sigma_1^{fr}+\sigma_2^{fr}}^{p_1, p_2}(_{[p_1, p_2 \circ k_1]|}M_1{}^{j_1}+^{j_2}{}_{[p_1, p_2 \circ k_2]|}M_2)$

Axiom schema (10) describes how the primitive freeze operators behave with respect to the sum operators; it produces a single axiom for each morphism $\sigma_1^{fr} : \Sigma_1^{fr} \to \Sigma_1^{out}$, $\sigma_2^{fr} : \Sigma_2^{fr} \to \Sigma_2^{out}$ and coproducts $<j_1, j_2>$ of Σ_1^{out} and Σ_2^{out}, $<k_1, k_2>$ of Σ_1^{fr} and Σ_2^{fr}, $<l_1, l_2>$ of Σ^{in} and Σ_1^{fr}, $<m_1, m_2>$ of Σ^{in} and Σ_2^{fr}, and $<p_1, p_2>$ of Σ^{in} and $\Sigma_1^{fr} + \Sigma_2^{fr}$; the morphism $\sigma_1^{fr} + \sigma_2^{fr}$ is determined by the coproducts $<k_1, k_2>$ and $<j_1, j_2>$, $[p_1, p_2 \circ k_1]$ by $<l_1, l_2>$ and $[p_1, p_2 \circ k_2]$ by $<m_1, m_2>$.

3.2. *Derived laws*

Several useful laws can be deduced from axioms of specification SP (see Figure 2): distributivity of the reduct with respect to the sum (laws (11), (12) and (13)), commutativity (law (14)), associativity (law (15)) and the existence and uniqueness (with respect to a fixed input signature) of the neutral element of sum (laws (16) and (17)). Finally, laws (18) and (19) formalise the intuition that we can always choose a canonical representative for sum and primitive freeze operators, as will be explained in Section 4.1.

Proof of (11). The axiom schema holds for each morphism

$$\sigma_i^{out} : \Sigma'^{out}_i \to \Sigma_i^{out}, \quad i = 1, 2,$$

and each coproduct $<k_1, k_2>$ and $<j_1, j_2>$ of Σ'^{out}_1 and Σ'^{out}_2 and of Σ_1^{out} and Σ_2^{out}, respectively. By (1) and (2),

$$(M_{1|\sigma_1^{out}}{}^{k_1}+^{k_2} M_{2|\sigma_2^{out}})_{|k_i} = M_{i|\sigma_i^{out}}, \quad i = 1, 2.$$

By (5) and the definition of $\sigma_1^{out} + \sigma_2^{out}$,

$$(M_1{}^{j_1}+^{j_2} M_2)_{|\sigma_1^{out}+\sigma_2^{out}|k_i} = (M_1{}^{j_1}+^{j_2} M_2)_{|j_i|\sigma_i^{out}}, \quad i = 1, 2.$$

https://doi.org/10.1017/S0960129502003687 Published online by Cambridge University Press

By (1) and (2),

$$(M_1 {}^{j_1}\!+^{j_2} M_2)_{|j_i|\sigma_i^{out}} = M_{i|\sigma_i^{out}}, \quad i = 1, 2.$$

Therefore we can conclude the proof by (3). □

Proof of (12). The axiom schema holds for each morphism

$$\sigma_i^{out} : \Sigma_i^{out} \to \Sigma^{out}, \quad i = 1, 2,$$

and each coproduct $<j_1, j_2>$ of Σ_1^{out} and Σ_2^{out}.

By (1) and (2),

$$(M_{|\sigma_1^{out}} {}^{j_1}\!+^{j_2} M_{|\sigma_2^{out}})_{|j_i} = M_{|\sigma_i^{out}}, \quad i = 1, 2.$$

By (5) and the definition of $[\sigma_1^{out}, \sigma_2^{out}]$,

$$M_{|[\sigma_1^{out}, \sigma_2^{out}]|j_i} = M_{|\sigma_i^{out}}, \quad i = 1, 2.$$

Therefore we can conclude the proof by (3). □

Proof of (13). The axiom schema holds for each morphism $\sigma^{in} : \Sigma^{in} \to \Sigma'^{in}$, and each coproduct $<j_1, j_2>$ of Σ_1^{out} and Σ_2^{out}.

By (5),

$$id_{\Sigma'^{in}|\sigma^{in}|}(M_1 {}^{j_1}\!+^{j_2} M_2)_{|id_{\Sigma^{out}}|j_i} = {}_{\sigma^{in}|id_{\Sigma^{in}}|}(M_1 {}^{j_1}\!+^{j_2} M_2)_{|j_i|id_{\Sigma_i^{out}}}, \quad i = 1, 2.$$

By (1) and (2),

$${}_{\sigma^{in}|id_{\Sigma^{in}}|}(M_1 {}^{j_1}\!+^{j_2} M_2)_{|j_i|id_{\Sigma_i^{out}}} = {}_{\sigma^{in}|}M_i, \quad i = 1, 2.$$

By (1) and (2),

$$({}_{\sigma^{in}|}M_1 {}^{j_1}\!+^{j_2} {}_{\sigma^{in}|}M_2)_{|j_i} = {}_{\sigma^{in}|}M_i, \quad i = 1, 2.$$

Therefore we can conclude the proof by (3). □

Proof of (14). The axiom schema holds for each coproduct $<j_1, j_2>$ of Σ_1^{out} and Σ_2^{out}.

By (1) and (2), $(M_1 {}^{j_1}\!+^{j_2} M_2)_{|j_i} = M_i = (M_2 {}^{j_2}\!+^{j_1} M_1)_{|j_i}$, therefore we can conclude the proof by (3). □

Proof of (15). The axiom schema holds for each coproduct $<j_1, j_2>$ of Σ_1^{out} and Σ_2^{out}, $<l_1, l_2>$ of $\Sigma_1^{out} + \Sigma_2^{out}$ and Σ_3^{out}, $<m_1, m_2>$ of Σ_1^{out} and $\Sigma_2^{out} + \Sigma_3^{out}$ and $<p_1, p_2>$ of Σ_2^{out} and Σ_3^{out} such that $l_1 \circ j_1 = m_1$, $l_1 \circ j_2 = m_2 \circ p_1$ and $l_2 = m_2 \circ p_2$.

By (1) and (2),

$$\begin{aligned} ((M_1 {}^{j_1}\!+^{j_2} M_2)^{l_1}\!+^{l_2} M_3)_{|l_1} &= M_1 {}^{j_1}\!+^{j_2} M_2 \\ ((M_1 {}^{j_1}\!+^{j_2} M_2)^{l_1}\!+^{l_2} M_3)_{|l_2} &= M_3 \end{aligned}$$

By (1), (2), (5) and the hypothesis on the coproducts,

$$\begin{aligned} (M_1 {}^{m_1}\!+^{m_2} (M_2 {}^{p_1}\!+^{p_2} M_3))_{|l_1|j_1} &= (M_1 {}^{m_1}\!+^{m_2} (M_2 {}^{p_1}\!+^{p_2} M_3))_{|m_1} = M_1, \\ (M_1 {}^{m_1}\!+^{m_2} (M_2 {}^{p_1}\!+^{p_2} M_3))_{|l_1|j_2} &= (M_1 {}^{m_1}\!+^{m_2} (M_2 {}^{p_1}\!+^{p_2} M_3))_{|m_2|p_1} = M_2. \end{aligned}$$

Therefore, by (1), (2) and (3),

$$((M_1 {}^{j_1}\!+^{j_2} M_2)^{l_1}\!+^{l_2} M_3)_{|l_1} = (M_1 {}^{m_1}\!+^{m_2} (M_2 {}^{p_1}\!+^{p_2} M_3))_{|l_1}.$$

https://doi.org/10.1017/S0960129502003687 Published online by Cambridge University Press

By (2), (5) and the hypothesis on the coproducts,

$$(M_1{}^{m_1}\!+^{m_2}(M_2{}^{p_1}\!+^{p_2}M_3))_{|l_2} = (M_1{}^{m_1}\!+^{m_2}(M_2{}^{p_1}\!+^{p_2}M_3))_{|m_2|p_2} = M_3$$

Therefore we can conclude the proof by (3). □

Note that $<l_1 \circ j_1, l_1 \circ j_2, l_2>$ (or, equivalently, $<m_1, m_2 \circ p_1, m_2 \circ p_2>$) is a coproduct of Σ_1^{out}, Σ_2^{out} and Σ_3^{out}. Hence, by virtue of law (15), we can adopt the following consistent notation.

Let $J = <j_1, .., j_n>$, $n \geqslant 1$, denote a finite coproduct with $j_i : \Sigma_i^{out} \to \Sigma^{out}$, $i = 1, .., n$, and let $\{M_i : mix(\Sigma^{in}, \Sigma_i^{out})\}_{i=1,..,n}$ be an indexed set of mixin expressions (for $n = 1$ the coproduct reduces to the unique morphism $\varnothing_{\Sigma^{out}}$ for some initial object $\varnothing$ in **Sig**). Then $\sum_J M_i$ is inductively defined by:

$$\sum_{<j_0>} M_0 = M_0,\ j_0 = \varnothing_{\Sigma^{out}}$$

$$\sum_J M_i = \left(\sum_K M_i\right)^{l}\!+^{j_n} M_n, \text{ if } n > 1,\ K = <k_1, .., k_{n-1}>,\ l \circ k_i = j_i \text{ for } i = 1, .., n-1.$$

Finally, the laws (11), (12), (13) and (14) can be easily generalized to this notation.

Proof of (16). The axiom schema holds for each morphism

$$\varnothing_{\Sigma_1^{out}} : \varnothing \to \Sigma_1^{out}, \quad \varnothing_{\Sigma_2^{out}} : \varnothing \to \Sigma_2^{out}$$

and any initial object $\varnothing$ in **Sig**.

By (1) and (4),

$$M_1{}^{id_{\Sigma_1^{out}}}\!+^{\varnothing_{\Sigma_1^{out}}} M_{2|\varnothing_{\Sigma_2^{out}}} = (M_1{}^{id_{\Sigma_1^{out}}}\!+^{\varnothing_{\Sigma_1^{out}}} M_{2|\varnothing_{\Sigma_2^{out}}})_{|id_{\Sigma_1^{out}}} = M_1.$$

□

Note that, by the definition of identity and initial object, $<id_{\Sigma_1^{out}}, \varnothing_{\Sigma_1^{out}}>$ is always a coproduct.

Proof of (17). The axiom schema holds for any pair of morphisms $\varnothing_{\Sigma_1^{out}} : \varnothing \to \Sigma_1^{out}$ and $\varnothing_{\Sigma_2^{out}} : \varnothing \to \Sigma_2^{out}$, with $\varnothing$ any initial object in **Sig**.

By laws (14) and (16),

$$\begin{aligned} M_{1|\varnothing_{\Sigma_1^{out}}} &= M_{1|\varnothing_{\Sigma_1^{out}}}{}^{id_\varnothing}\!+^{id_\varnothing} M_{2|\varnothing_{\Sigma_2^{out}}} \\ &= M_{2|\varnothing_{\Sigma_2^{out}}}{}^{id_\varnothing}\!+^{id_\varnothing} M_{1|\varnothing_{\Sigma_1^{out}}} \\ &= M_{2|\varnothing_{\Sigma_2^{out}}} \end{aligned}$$

□

Proof of (18). The axiom schema holds for any possible choice of coproducts $<j_1, j_2>$ and $<k_1, k_2>$ of Σ_1^{out} and Σ_2^{out}.

By the definition of $[j_1, j_2]$, and by (5), (1) and (2), we have $(M_1{}^{j_1}\!+^{j_2} M_2)_{|[j_1,j_2]|k_i} = M_i$, $i = 1, 2$, and hence, we can conclude the proof by (1), (2) and (3). □

Proof of (19). The axiom schema holds for each morphism $\sigma^{fr} : \Sigma^{fr} \to \Sigma^{out}$, $\sigma'^{fr} : \Sigma'^{fr} \to \Sigma^{out}$, $\sigma : \Sigma^{fr} \to \Sigma'^{fr}$ such that $\sigma'^{fr} \circ \sigma = \sigma^{fr}$, and coproduct $<j_1, j_2>$ of Σ^{in} and Σ^{fr} and $<k_1, k_2>$ of Σ^{in} and Σ'^{fr}.

https://doi.org/10.1017/S0960129502003687 Published online by Cambridge University Press

$$
\begin{array}{rcll}
M_1 \oplus M_2 & = & \mathit{freeze}_{\Sigma^{fr} \cap \Sigma^{in}}({}_{\Sigma^{in}|}M_1 + {}_{\Sigma^{in}|}M_2) & \text{(i)} \\
\textbf{freeze } \Sigma^{fr} \textbf{ in } M & = & \mathit{freeze}_{\Sigma^{fr} \cap \Sigma^{in}}(M) & \text{(ii)} \\
\textbf{restrict } \Sigma^{rs} \textbf{ in } M & = & M_{|\Sigma^{out} \setminus \Sigma^{rs}} & \text{(iii)} \\
\textbf{hide } \Sigma^{hd} \textbf{ in } M & = & (\mathit{freeze}_{\Sigma^{hd} \cap \Sigma^{in}}(M))_{|\Sigma^{out} \setminus \Sigma^{hd}} & \text{(iv)} \\
M_1 \Leftarrow M_2 & = & (\textbf{restrict } \Sigma_1^{out} \cap \Sigma_2^{out} \textbf{ in } M_1) \oplus M_2 & \text{(v)} \\
M_2 \circ M_1 & = & \textbf{hide } \Sigma_1^{out} \textbf{ in } M_1 \oplus M_2 & \text{(vi)}
\end{array}
$$

Fig. 3. Definitions of high-level operations

$$
\begin{array}{rcll}
M_1 \oplus M_2 & = & M_2 \oplus M_1 & \text{(vii)} \\
(M_1 \oplus M_2) \oplus M_3 & = & M_1 \oplus (M_2 \oplus M_3) & \text{(viii)} \\
M_1 \oplus \textbf{restrict } \Sigma_2^{out} \textbf{ in } M_2 & = & M_1 \text{ if } \Sigma_2^{in} \subseteq \Sigma_1^{in} & \text{(ix)} \\
M_1 \Leftarrow M_2 & = & M_1 \oplus M_2 \text{ if } \Sigma_1^{out} \cap \Sigma_2^{out} = \varnothing & \text{(x)} \\
M_1 \Leftarrow M_2 & = & M_2 \text{ if } \Sigma_1^{in} \subseteq \Sigma_2^{in}, \Sigma_1^{out} \subseteq \Sigma_2^{out} & \text{(xi)} \\
M_1 \Leftarrow \textbf{restrict } \Sigma_2^{out} \textbf{ in } M_2 & = & M_1 \text{ if } \Sigma_2^{in} \subseteq \Sigma_1^{in} & \text{(xii)}
\end{array}
$$

Fig. 4. Some properties of high-level operations

This can be obtained easily as a particular instantiation of (8) or (9), by choosing $\sigma^{out} = id_{\Sigma^{out}}$ (in (8)) or by choosing $\sigma^{in} = id_{\Sigma^{in}}$ (in (9)) and then applying (4). □

3.3. *Axioms for derived operations*

In this subsection we provide a formal definition of the operators for combining mixin modules that were introduced in Section 2.2 in terms of the three operators of sum, reduct and primitive freeze.

In Figure 3 we give a set of axioms that state that each high-level operator can be defined from the primitive operations. We use the abbreviation $\mathit{freeze}_{\Sigma^{fr}}$ for $\mathit{freeze}^{j_1, j_2}_{\sigma^{fr}}$ when σ^{fr} is the inclusion from Σ^{fr} into Σ^{out}, and j_1 and j_2 are the inclusions from Σ^{fr} and Σ^{in}, respectively, into Σ^{out}. Note indeed that, by our assumptions on boolean signature categories, if $\Sigma_1 \cap \Sigma_2 = \varnothing$, then $\Sigma_1 \hookrightarrow \Sigma_1 \cup \Sigma_2 \hookleftarrow \Sigma_2$ is a coproduct. In particular, this holds when $\Sigma_2 = \Sigma \setminus \Sigma_1$.

Analogously, if $\Sigma^{in} \subseteq \Sigma'^{in}$ and $\Sigma'^{out} \subseteq \Sigma^{out}$, then ${}_{\Sigma'^{in}|}M_{|\Sigma'^{out}}$ stands for ${}_{\sigma^{in}|}M_{|\sigma^{out}}$, where $\sigma^{in} = i_{\Sigma^{in}, \Sigma'^{in}}$ and $\sigma^{out} = i_{\Sigma'^{out}, \Sigma^{out}}$.

In Figure 4 we state some (intuitively expected) properties of the high-level operators. They all can be derived from the axioms for primitive operations (1)–(19) and the axioms defining derived operations (i)–(vi).

Let us begin by proving that the merge operator is commutative (axiom (vii)). Indeed, by axiom (1) and law (14),

$$
\begin{array}{rcl}
M_1 \oplus M_2 & = & \mathit{freeze}_{\Sigma^{fr} \cap \Sigma^{in}}({}_{\Sigma^{in}|}M_1 + {}_{\Sigma^{in}|}M_2) \\
& = & \mathit{freeze}_{\Sigma^{fr} \cap \Sigma^{in}}({}_{\Sigma^{in}|}M_2 + {}_{\Sigma^{in}|}M_1) \\
& = & M_2 \oplus M_1.
\end{array}
$$

A less trivial proof is the associativity of merge (axiom (viii)). We need the following two lemmas.

https://doi.org/10.1017/S0960129502003687 Published online by Cambridge University Press

Lemma 3.1. Let $M_i : \Sigma^{in} \to \Sigma_i^{out}$, $i = 1, 2$, be two mixins, with $\Sigma_1^{out} \cap \Sigma_2^{out} = \varnothing$, and let $\Sigma^{fr}, \Sigma'^{fr}$ be two signatures such that $\Sigma^{fr} \subseteq \Sigma'^{fr}$, $\Sigma^{fr} \subseteq \Sigma_1^{out} \cap \Sigma^{in}$, $\Sigma'^{fr} \subseteq (\Sigma_1^{out} \cup \Sigma_2^{out}) \cap \Sigma^{in}$. Then,

$$freeze_{\Sigma'^{fr}}(_{\Sigma^{in}|}freeze_{\Sigma^{fr}}(M_1) + M_2) = freeze_{\Sigma'^{fr}}(M_1 + M_2).$$

Proof. Set $\Sigma^{out} = \Sigma_1^{out} \cup \Sigma_2^{out}$. Then by axioms (9), (7), (10) and (6) of SP,

$$\begin{aligned} &freeze_{\Sigma'^{fr}}((_{\Sigma^{in}|}freeze_{\Sigma^{fr}}(M_1)) + M_2) \\ &\qquad = \quad freeze_{[i_{\Sigma'^{fr},\Sigma^{out}}, i_{\Sigma^{fr},\Sigma^{out}}]}(_{id_{\Sigma^{fr}} + i_{\Sigma^{in}\setminus\Sigma^{fr},\Sigma^{in}}|}M_1 + _{\Sigma^{fr}+\Sigma^{in}|}M_2). \end{aligned}$$

Now set $\sigma'^{in} = [i_{\Sigma^{fr},\Sigma'^{fr}}, id_{\Sigma'^{fr}}]$.

By unicity, $i_{\Sigma'^{fr},\Sigma^{out}} \circ \sigma'^{in} = [i_{\Sigma'^{fr},\Sigma^{out}}, i_{\Sigma^{fr},\Sigma^{out}}]$. Hence, by axiom (9)

$$\begin{aligned} &freeze_{[i_{\Sigma'^{fr},\Sigma^{out}}, i_{\Sigma^{fr},\Sigma^{out}}]}(_{id_{\Sigma^{fr}} + i_{\Sigma^{in}\setminus\Sigma^{fr},\Sigma^{in}}|}M_1 + _{\Sigma^{fr}+\Sigma^{in}|}M_2) \\ &\quad = \quad freeze_{\Sigma'^{fr}}(_{id_{\Sigma^{in}\setminus\Sigma'^{fr}} + \sigma'^{in} | id_{\Sigma^{fr}} + i_{\Sigma^{in}\setminus\Sigma^{fr},\Sigma^{in}}|}M_1 + _{id_{\Sigma^{in}\setminus\Sigma'^{fr}} + \sigma'^{in}|\Sigma^{fr}+\Sigma^{in}|}M_2). \end{aligned}$$

Now

$$\begin{aligned} (id_{\Sigma^{in}\setminus\Sigma'^{fr}} + \sigma'^{in}) \circ (id_{\Sigma^{fr}} + i_{\Sigma^{in}\setminus\Sigma^{fr},\Sigma^{in}})_{|\Sigma^{in}\setminus\Sigma^{fr}} &= (id_{\Sigma^{in}\setminus\Sigma'^{fr}} + \sigma'^{in})_{|\Sigma^{in}\setminus\Sigma^{fr}} \\ &= [i_{\Sigma^{in}\setminus\Sigma'^{fr},\Sigma^{in}}, i_{\Sigma'^{fr}\setminus\Sigma^{fr},\Sigma^{in}}] \\ &= i_{\Sigma^{in}\setminus\Sigma^{fr},\Sigma^{in}} \end{aligned}$$

and

$$\begin{aligned} (id_{\Sigma^{in}\setminus\Sigma'^{fr}} + \sigma'^{in}) \circ (id_{\Sigma^{fr}} + i_{\Sigma^{in}\setminus\Sigma^{fr},\Sigma^{in}})_{|\Sigma^{fr}} &= (id_{\Sigma^{in}\setminus\Sigma'^{fr}} + \sigma'^{in})_{|\Sigma^{fr}} \\ &= \sigma'^{in}{}_{|\Sigma^{fr}} \\ &= i_{\Sigma^{fr},\Sigma^{in}}. \end{aligned}$$

Hence $(id_{\Sigma^{in}\setminus\Sigma'^{fr}} + \sigma'^{in}) \circ (id_{\Sigma^{fr}} + i_{\Sigma^{in}\setminus\Sigma^{fr},\Sigma^{in}})_{|\Sigma^{in}\setminus\Sigma^{fr}} = [i_{\Sigma^{in}\setminus\Sigma^{fr},\Sigma^{in}}, i_{\Sigma^{fr},\Sigma^{in}}] = id_{\Sigma^{in}}$.

Analogously, if j denotes the injection from Σ^{in} to $\Sigma^{in} + \Sigma'^{fr}$, then

$$(id_{\Sigma^{in}\setminus\Sigma'^{fr}} + \sigma'^{in}) \circ j = [i_{\Sigma^{in}\setminus\Sigma'^{fr},\Sigma^{in}}, i_{\Sigma'^{fr},\Sigma^{in}}] = id_{\Sigma^{in}}.$$

Therefore, we have

$$\begin{aligned} &freeze_{\Sigma'^{fr}}(_{id_{\Sigma^{in}\setminus\Sigma'^{fr}} + \sigma'^{in} | id_{\Sigma^{fr}} + i_{\Sigma^{in}\setminus\Sigma^{fr},\Sigma^{in}}|}M_1 + _{id_{\Sigma^{in}\setminus\Sigma'^{fr}} + \sigma'^{in}|\Sigma^{fr}+\Sigma^{in}|}M_2) \\ &\qquad = \quad freeze_{\Sigma'^{fr}}(M_1 + M_2), \end{aligned}$$

which concludes the proof. □

Lemma 3.2. Let $M : \Sigma^{in} \to \Sigma^{out}$ be a mixin and let $\Sigma^{fr}, \Sigma'^{in}$ be two signatures such that $\Sigma^{fr} \subseteq \Sigma^{out} \cap \Sigma^{in}$, $\Sigma^{in} \subseteq \Sigma'^{in}$. Then,

$$freeze_{\Sigma'^{fr}}(_{\Sigma'^{in}|}M) = freeze_{\Sigma'^{fr}\setminus(\Sigma'^{in}\setminus\Sigma^{in})}(_{(\Sigma'^{in}\setminus(\Sigma^{in}\cup\Sigma^{fr}))\cup\Sigma^{in}|}M).$$

Proof. Trivially,

$$freeze_{\Sigma'^{fr}}(_{\Sigma'^{in}|}M) = freeze_{\Sigma'^{fr}}(_{(\Sigma'^{in}\setminus\Sigma^{in})\cup\Sigma^{in}|}M).$$

https://doi.org/10.1017/S0960129502003687 Published online by Cambridge University Press

Now, by axiom (6),

$$freeze_{\Sigma'^{fr}}({}_{(\Sigma'^{in}\setminus\Sigma^{in})\cup\Sigma^{in}|}M)$$
$$= \quad freeze_{\Sigma^{fr}\setminus(\Sigma'^{in}\setminus\Sigma^{in})}(freeze_{\Sigma^{fr}\cap(\Sigma'^{in}\setminus\Sigma^{in})}({}_{(\Sigma'^{in}\setminus(\Sigma^{in}\cup\Sigma^{fr}))\cup((\Sigma'^{in}\setminus\Sigma^{in})\cap\Sigma^{fr})\cup\Sigma^{in}|}M)),$$

which, by axiom (9),

$$= freeze_{\Sigma^{fr}\setminus(\Sigma'^{in}\setminus\Sigma^{in})}(freeze_{\varnothing}({}_{(\Sigma'^{in}\setminus(\Sigma^{in}\cup\Sigma^{fr}))\cup\Sigma^{in}|}M)),$$

which by axiom (7),

$$= freeze_{\Sigma'^{fr}\setminus(\Sigma'^{in}\setminus\Sigma^{in})}({}_{(\Sigma'^{in}\setminus(\Sigma^{in}\cup\Sigma^{fr}))\cup\Sigma^{in}|}M),$$

concluding the proof. □

From Lemmas 3.1 and 3.2 we can derive:

$$(M_1 \oplus M_2) \oplus M_3 = freeze_{\Sigma^{fr}\cap\Sigma^{in}}({}_{\Sigma^{in}|}M_1 + {}_{\Sigma^{in}|}M_2 + {}_{\Sigma^{in}|}M_3) = M_1 \oplus (M_2 \oplus M_3),$$

with $\Sigma^{fr} = \Sigma_1^{fr} \cup \Sigma_2^{fr} \cup \Sigma_3^{fr}$, $\Sigma_i^{fr} = \Sigma_i^{out} \setminus \Sigma_i^{in}$, $\quad i = 1, 2, 3$, $\Sigma^{in} = \Sigma_1^{in} \cup \Sigma_2^{in} \cup \Sigma_3^{in}$. Note that from this proof we can easily derive the following law (for a finite non-empty set of indexes I): $\oplus_{i\in I} M_i = freeze_{\Sigma^{fr}\cap\Sigma^{in}}(\sum_{i\in I} {}_{\Sigma^{in}|}M_i)$, where $M_i : \Sigma_i^{in} \to \Sigma_i^{out}$, $i \in I$, $\Sigma^{fr} = \bigcup_{i\in I}(\Sigma_i^{out} \setminus \Sigma_i^{in})$, $\Sigma^{in} = \bigcup_{i\in I} \Sigma_i^{in}$, $\Sigma_i^{out} \cap \Sigma_j^{out} = \varnothing$, for any $i, j \in I$, $i \neq j$.

The existence of the neutral element (axiom (ix)) derives from axioms (4), (7) and law (16):

$$M_1 \oplus M_{2|\varnothing} = freeze_{(\Sigma_1^{out}\setminus\Sigma_1^{in})\cap\Sigma_1^{in}}({}_{\Sigma_1^{in}|}M_1 + {}_{\Sigma_1^{in}|}M_{2|\varnothing}) = freeze_{\varnothing}(M_1) = M_1.$$

The overriding operator reduces to merge (axiom (x)) when the output signatures are disjoint: $M_1 \Leftarrow M_2 = M_{1|\Sigma_1^{out}\setminus\Sigma_2^{out}} \oplus M_2 = M_{1|\Sigma_1^{out}} \oplus M_2 = M_1 \oplus M_2$.

The existence of the left neutral element for overriding is stated in axiom (xi):

$$M_1 \Leftarrow M_2 = M_{1|\Sigma_1^{out}\setminus\Sigma_2^{out}} \oplus M_2 = M_{1|\varnothing} \oplus M_2 = M_2.$$

From axiom (xi) one can easily deduce idempotency of overriding: $M \Leftarrow M = M$.

The existence of the right neutral element for overriding is stated in axiom (xii):

$$M_1 \Leftarrow M_{2|\varnothing} = M_{1|\Sigma_1^{out}\setminus\varnothing} \oplus M_{2|\varnothing} = M_1.$$

4. Normal form theorem

In this section we show that each mixin expression is provably equal to another mixin expression having a standard form (only one application of the freeze and the ouput reduct operator). Note that, since our approach is parametric in the core language, mixin expressions are not built on top of constants (which depend on the core language), but only on top of variables.

For the sake of simplicity, we omit the coproducts over which sum and freeze operators are indexed.

Definition 4.1. Let X be a family of non-empty sets of variables indexed over

$$\{mix(\Sigma^{in}, \Sigma^{out}) \mid \Sigma^{in}, \Sigma^{out} \text{ in } \mathbf{Sig}\}.$$

Then ME_X denotes the set of all terms inductively defined over the signature of SP and X.

https://doi.org/10.1017/S0960129502003687 Published online by Cambridge University Press

Theorem 4.2. Let M be a term in ME_X. Then there exist a coproduct $J = <j_1, .., j_n>$, variables x_i in X and morphisms σ_i^{in}, σ^{out} and σ^{fr}, $i = 1, .., n$ $(n \geqslant 1)$, such that

$$M = (freeze_{\sigma^{fr}}(\sum_J \sigma_i^{in}|x_i))_{|\sigma^{out}} .$$

Proof. The proof is by induction over the structure of terms.

Basis: let $M \equiv x$, where $\equiv$ denotes syntactic equality, with x in $X_{mix(\Sigma^{in},\Sigma^{out})}$. Then it is immediate to prove, by axioms (4) and (7), that

$$x = (freeze_{\varnothing_{\Sigma^{out}}}(\sum_{<\varnothing_{\Sigma^{out}}>} [id_{\Sigma^{in}}, \varnothing_{\Sigma^{in}}]|x))_{|id_{\Sigma^{out}}},$$

with $\varnothing$ any initial object in **Sig**.

Induction step:

— $M \equiv {}_{\sigma'^{in}}|M'_{|\sigma'^{out}}$: By induction,

$$M' = (freeze_{\sigma^{fr}}(\sum_J \sigma_i^{in}|x_i))_{|\sigma^{out}} .$$

Therefore, by axioms (5) and (9),

$$\sigma'^{in}|((freeze_{\sigma^{fr}}(\sum_J \sigma_i^{in}|x_i))_{|\sigma^{out}})_{|\sigma'^{out}} = (freeze_{\sigma^{fr}}(\sigma'^{in}+id_{\Sigma^{fr}}|\sum_J \sigma_i^{in}|x_i))_{|\sigma^{out}\circ\sigma'^{out}},$$

which, by law (13) and axiom (5),

$$= (freeze_{\sigma^{fr}}(\sum_J (\sigma'^{in}+id_{\Sigma^{fr}})\circ\sigma_i^{in}|x_i))_{|\sigma^{out}\circ\sigma'^{out}} .$$

— $M \equiv freeze_{\sigma'^{fr}}(M')$: By induction

$$M' = (freeze_{\sigma^{fr}}(\sum_J \sigma_i^{in}|x_i))_{|\sigma^{out}} .$$

Therefore, by axioms (8), (4) and (6)

$$freeze_{\sigma'^{fr}}((freeze_{\sigma^{fr}}(\sum_J \sigma_i^{in}|x_i))_{|\sigma^{out}})$$

$$= (freeze_{[\sigma'^{fr},\sigma^{out}\circ\sigma^{fr}]}([l_1,l_2\circ m_1,l_2\circ m_2]|\sum_J \sigma_i^{in}|x_i))_{|\sigma^{out}},$$

which, by law (13) and axiom (5),

$$= (freeze_{[\sigma'^{fr},\sigma^{out}\circ\sigma^{fr}]}(\sum_J [l_1,l_2\circ m_1,l_2\circ m_2]|\sigma_i^{in}|x_i))_{|\sigma^{out}}$$

— $M \equiv M_1 + M_2$: By induction

$$M_k = (freeze_{\sigma_k^{fr}}(\sum_{J_k} \sigma_i^{(k)}|x_i^{(k)}))_{|\sigma_k^{out}},$$

with $J_k = <j_1^{(k)}, .., j_{n_k}^{(k)}>$, $k = 1, 2$.

https://doi.org/10.1017/S0960129502003687 Published online by Cambridge University Press

Therefore, by law (11) and axiom (10),

$$(freeze_{\sigma_1^{fr}}(\sum_{J_1} \sigma_i^{(1)}|x_i^{(1)}))_{|\sigma_1^{out}} \; {}^{l_1}+^{l_2} (freeze_{\sigma_2^{fr}}(\sum_{J_2} \sigma_i^{(2)}|x_i^{(2)}))_{|\sigma_2^{out}}$$

$$=$$

$$(freeze_{\sigma_1^{fr}+\sigma_2^{fr}}(([p_1,p_2\circ k_1]|\sum_{J_1} \sigma_i^{(1)}|x_i^{(1)})^{l_1}+^{l_2}([p_1,p_2\circ k_2]|\sum_{i\in J_2} \sigma_i^{(2)}|x_i^{(2)})))_{|\sigma_1^{out}+\sigma_2^{out}}$$

which, by law (13) and axiom (5),

$$= (freeze_{\sigma_1^{fr}+\sigma_2^{fr}}(\sum_{J} \sigma_i^{in}|x_i))_{|\sigma_1^{out}+\sigma_2^{out}}$$

where $J = <l_1 \circ j_1^{(1)}, .., l_1 \circ j_{n_1}^{(1)}, l_2 \circ j_1^{(2)}, .., l_2 \circ j_{n_2}^{(2)}>$

for all $i = 1, .., n_1 + n_2$, $x_i = \begin{cases} x_i^{(1)} & \text{if } i \leqslant n_1 \\ x_i^{(2)} & \text{if } i > n_1 \end{cases}$ $\quad \sigma_i^{in} = \begin{cases} [p_1, p_2 \circ k_1] \circ \sigma_i^{(1)} & \text{if } i \leqslant n_1 \\ [p_1, p_2 \circ k_2] \circ \sigma_i^{(2)} & \text{if } i > n_1 \end{cases}$

□

4.1. *A term rewriting system for mixin modules*

From the specification SP we can derive a complete, that is, strongly normalizing and confluent, term rewriting system $\mathscr{T}$ where normal forms have the shape specified in Theorem 4.2 (see Figure 5).

The rewriting rules correspond to all axioms used for proving the induction step of Theorem 4.2, including the derived laws (11) and (13). In order to keep the set of rewriting rules as simple as possible, the reduct operators have been split into two different classes, the input reduct operators ${}_{\sigma^{in}|}-$ and the output reduct operators $-_{|\sigma^{out}}$ for any morphism σ^{in} and σ^{out}, respectively.

The right direction of the rewriting rules is driven by Theorem 4.2, so that multiple applications of the reduct and freeze operators can reduce to a unique application (rules (4), (5) and (7)) and applications of the reduct operators over input and output signatures can migrate inside and outside the terms, respectively (all the other rules).

Note that axiom (10) and law (11) have been slightly changed in $\mathscr{T}$ in order to get a confluent system. Rules (1) and (2) correspond to law (11), whereas rules (10) and (11) correspond to axiom (10). In rule (10) the coproduct $<m_1, m_2>$ (see Figure 1) is defined by $m_1 = id_{\Sigma^{in}}$ and $m_2 = \varnothing_{\Sigma^{in}}$, with $\varnothing$ any initial object in **Sig**; analogously, in rule (11), $l_1 = id_{\Sigma^{in}}$ and $l_2 = \varnothing_{\Sigma^{in}}$.

The proof of strong normalisation is also driven by Theorem 4.2; indeed, it suffices to define a reduction ordering that counts all the operations that are in the 'wrong' position inside a given term.

Theorem 4.3. The term rewriting system of $\mathscr{T}$ is strongly normalizing.

Proof. Let us define a map $|\_|$ from the set of terms of $\mathscr{T}$ into the set of natural numbers such that:

— $|\phi l| > |\phi r|$ for any reduction rule $l \to r$ in $\mathscr{T}$ and any substitution ϕ;
— if $|M_1| < |M_2|$, then $|\mathscr{C}[M_1]| < |\mathscr{C}[M_2]|$, for any context $\mathscr{C}[\ ]$ and terms M_1, M_2 of $\mathscr{T}$.

https://doi.org/10.1017/S0960129502003687 Published online by Cambridge University Press

(1) $M_{1|\sigma_1^{out}}{}^{k_1}{+}^{k_2}M_2 \to (M_1{}^{j_1}{+}^{j_2}M_2)_{|\sigma_1^{out}+id_{\Sigma_2^{out}}}$
(2) $M_1{}^{k_1}{+}^{k_2}M_{2|\sigma_2^{out}} \to (M_1{}^{j_1}{+}^{j_2}M_2)_{|id_{\Sigma_1^{out}}+\sigma_2^{out}}$
(3) ${}_{\sigma^{in}|}(M_1{}^{j_1}{+}^{j_2}M_2) \to {}_{\sigma^{in}|}M_1{}^{j_1}{+}^{j_2}{}_{\sigma^{in}|}M_2$
(4) ${}_{\sigma_2^{in}|\sigma_1^{in}|}M \to {}_{\sigma_2^{in}\circ\sigma_1^{in}|}M$
(5) $M_{|\sigma_1^{out}|\sigma_2^{out}} \to M_{|\sigma_1^{out}\circ\sigma_2^{out}}$
(6) ${}_{\sigma^{in}|}(M_{|\sigma^{out}}) \to ({}_{\sigma^{in}|}M)_{|\sigma^{out}}$
(7) $freeze_{\sigma_2^{fr}}^{k_1,k_2}(freeze_{\sigma_1^{fr}}^{j_1,j_2}(M)) \to freeze_{[\sigma_2^{fr},\sigma_1^{fr}]}^{l_1,l_2}({}_{[l_1,l_2\circ m_1,l_2\circ m_2]|}M)$
(8) $freeze_{\sigma^{fr}}^{j_1,j_2}(M_{|\sigma^{out}}) \to (freeze_{\sigma'^{fr}}^{k_1,k_2}({}_{id_{\Sigma^{in}}+\sigma^{in}|}M))_{|\sigma^{out}}$
(9) ${}_{\sigma^{in}|}(freeze_{\sigma^{fr}}^{j_1,j_2}(M)) \to freeze_{\sigma'^{fr}}^{k_1,k_2}({}_{\sigma^{in}+\sigma'^{in}|}M)$
(10) $freeze_{\sigma_1^{fr}}^{l_1,l_2}(M_1){}^{j_1}{+}^{j_2}M_2 \to freeze_{\sigma_1^{fr}+\varnothing_{\Sigma_2^{out}}}^{p_1,p_2}({}_{[p_1,p_2\circ k_1]|}M_1{}^{j_1}{+}^{j_2}{}_{p_1|}M_2)$
(11) $M_1{}^{j_1}{+}^{j_2}freeze_{\sigma_2^{fr}}^{m_1,m_2}(M_2) \to freeze_{\varnothing_{\Sigma_1^{out}}+\sigma_2^{fr}}^{p_1,p_2}({}_{p_1|}M_1{}^{j_1}{+}^{j_2}{}_{[p_1,p_2\circ k_2]|}M_2)$

Fig. 5. The term rewriting system $\mathscr{T}$ derived from SP

Then, by induction over the reduction rules, it is trivial to prove that for any term M_1 and M_2 of $\mathscr{T}$, if $M_1 \overset{+}{\to} M_2$, then $|M_2| < |M_1|$ (with $\overset{+}{\to}$ the transitive closure of $\to$), so it is not possible to have an infinite reduction sequence.

To this end, we define the following map $|\_|$:

$$|x| = 0$$

$$|{}_{\sigma^{in}|}M| = \begin{cases} |M|+1 & \text{if } sym(M) \neq \varnothing \\ |M| & \text{otherwise} \end{cases}$$

$$|M_1{}^{j_1}{+}^{j_2}M_2| = \begin{cases} |M_1|+|M_2|+1 & \text{if } (Freeze \cup OutReduct) \cap sym(M_1{}^{j_1}{+}^{j_2}M_2) \neq \varnothing \\ |M_1|+|M_2| & \text{otherwise} \end{cases}$$

$$|freeze_{\sigma^{fr}}^{j_1,j_2}(M)| = \begin{cases} |M|+1 & \text{if } (Freeze \cup OutReduct) \cap sym(M) \neq \varnothing \\ |M| & \text{otherwise} \end{cases}$$

$$|M_{|\sigma^{out}}| = \begin{cases} |M|+1 & \text{if } OutReduct \cap sym(M) \neq \varnothing \\ |M| & \text{otherwise} \end{cases}$$

where *Freeze* and *OutReduct* are defined by

$$Freeze = \{freeze_{\sigma}^{j_1,j_2} \mid \sigma \text{ morphism in } \mathbf{Sig}, <j_1, j_2> \text{ coproduct in } \mathbf{Sig}\}$$
$$OutReduct = \{{}_{-|\sigma} \mid \sigma \text{ morphism in } \mathbf{Sig}\}$$

and *sym* is the function returning the set of all operator symbols contained in a given term:

$$sym(x) = \varnothing$$
$$sym({}_{\sigma^{in}|}M) = \{{}_{\sigma^{in}|-}\} \cup sym(M)$$
$$sym(M_1{}^{j_1}{+}^{j_2}M_2) = \{{}_-{}^{j_1}{+}^{j_2}{}_-\} \cup sym(M_1) \cup sym(M_2)$$
$$sym(freeze_{\sigma^{fr}}^{j_1,j_2}(M)) = \{freeze_{\sigma^{in}}^{j_1,j_2}\} \cup sym(M)$$
$$sym(M_{|\sigma^{out}}) = \{{}_{-|\sigma^{out}}\} \cup sym(M)$$

Now it is trivial to prove that $|\phi l| > |\phi r|$ for any reduction rule $l \to r$ in $\mathscr{T}$ and any

https://doi.org/10.1017/S0960129502003687 Published online by Cambridge University Press

substitution ϕ. For instance, in rule (1) we have

$$|\phi l| = 1 + |\phi M_1| + |\phi M_2| < |\phi M_1| + |\phi M_2| = |\phi r|.$$

Furthermore, it is easy to show that for any context $\mathscr{C}[\ ]$ and terms M_1, M_2 of $\mathscr{T}$, if $|M_1| < |M_2|$, then $|\mathscr{C}[M_1]| < |\mathscr{C}[M_2]|$, and hence we can conclude the proof. □

Now let us consider confluency. Formally, the term rewriting system defined in Figure 5 is not confluent, except for some rare cases†. For instance, let **Sig** be the category **Set** of small sets and consider the term $M \equiv x_{|\sigma}{}^{k_1}\!+^{k_2} y$ with $x{:}mix(\{i\},\{o'\})$, $y{:}mix(\{i\},\{o\})$, $\sigma{:}\{o\} \to \{o'\}$ and $k_1, k_2{:}\{o\} \to \{o_1, o_2\}$ defined by $k_i(o) = o_i$, $i = 1, 2$.

By applying rule (1), we have $M \to (x^{j_1}\!+^{j_2} y)_{|id_{\{o_1,o_2\}}}$, but also $M \to (x^{j'_1}\!+^{j'_2} y)_{|\sigma'}$, where $<j_1, j_2>$, $<j'_1, j'_2>$ are the two coproducts defined by $j_1(o') = o_1$, $j_2(o) = o_2$, $j'_1(o') = o_2$, $j'_2(o) = o_1$, and σ' is defined by $\sigma'(o_1) = o_2$, $\sigma'(o_2) = o_1$. But both $(x^{j_1}\!+^{j_2} y)_{|id_{\{o_1,o_2\}}}$ and $(x^{j'_1}\!+^{j'_2} y)_{|\sigma'}$ are normal forms, therefore the term rewriting system is not confluent. Clearly, this counter-example works as long as the system allows us to choose any possible coproduct of $\{o'\}$ and $\{o\}$, and the problem can be solved once a unique coproduct is fixed. In this way, for each pair of output signatures Σ_1^{out}, Σ_2^{out} we can restrict ourselves to consider only the canonical representative of the class of operators $\{^{j_1}\!+^{j_2} \mid <j_1, j_2>$ coproduct of Σ_1^{out} and $\Sigma_2^{out}\}$. Note that this is the equivalence class determined by the relation over sum operators defined by:

$$^{j_1}\!+^{j_2} \sim {}^{k_1}\!+^{k_2} \text{ iff } <j_1, j_2> \text{ and } <k_1, k_2> \text{ are coproducts of } \Sigma_1^{out} \text{ and } \Sigma_2^{out}.$$

An analogous problem arises with the freeze operators; by a careful analysis of the rewriting rules from (7) to (11) we can note that, for each pair of signatures Σ^{in}, Σ^{out}, it is possible to choose a freeze operator for the right-hand side ranging over the equivalence class determined by the following relation over freeze operators:

$$freeze_{\sigma^{fr}}^{j_1, j_2} \sim freeze_{\sigma'^{fr}}^{k_1, k_2}$$

iff $\sigma^{fr}{:}\Sigma^{fr} \to \Sigma^{out}$, $\sigma'^{fr}{:}\Sigma'^{fr} \to \Sigma^{out}$, $<j_1, j_2>$ is a coproduct of Σ^{in} and Σ^{fr}, $<k_1, k_2>$ is a coproduct of Σ^{in} and Σ'^{fr} and there exists an isomorphism $\sigma{:}\Sigma^{fr} \to \Sigma'^{fr}$ such that $\sigma'^{fr} \circ \sigma = \sigma^{fr}$ (note the analogy with law (19)).

By laws (18) and (19), for every term M there always exists a canonical term $\overline{M}$, that is, a term built on top of canonical sum and freeze operators such that $M = \overline{M}$. This allows us to consider only canonical reduction sequences, that is, reduction sequences containing only canonical terms.

Finally, we can prove that $\mathscr{T}$ is confluent whenever we restrict ourselves to consider canonical reduction sequences only.

Theorem 4.4. The term rewriting system $\mathscr{T}$ is confluent for canonical reduction sequences.

Proof. It is sufficient to show that each canonical critical pair (M_1, M_2) of $\mathscr{T}$ is convergent, that is, there exists a canonical term M such that $M_1 \overset{*}{\to} M$, $M_2 \overset{*}{\to} M$, with

† For instance, when **Sig** is a partial order with finite colimits.

https://doi.org/10.1017/S0960129502003687 Published online by Cambridge University Press

$\overset{*}{\rightarrow}$ the reflexive and transitive closure of $\rightarrow$ (restricted to canonical terms); therefore, $\mathscr{T}$ is weakly confluent, and by Theorem 4.3 we can conclude that $\mathscr{T}$ is confluent. For reasons of space, we will only show the two most significant cases; the others can be proved routinely in an analogous way.

— The left-hand side of rule (7) unifies with its proper subterm $freeze^{j_1,j_2}_{\sigma_1^{fr}}(M)$ yielding the canonical term $freeze^{k_1,k_2}_{\sigma_3^{fr}}(freeze^{j_1,j_2}_{\sigma_2^{fr}}(freeze^{h_1,h_2}_{\sigma_1^{fr}}(M)))$; then, applying rule (7) to the outer- and inner-most redex, respectively, we obtain the canonical critical pair $<M_1, M_2>$, with

$$\begin{aligned} M_1 &= freeze^{l_1,l_2}_{[\sigma_3^{fr},\sigma_2^{fr}]}({}_{[l_1,l_2\circ m_1,l_2\circ m_2]|}freeze^{h_1,h_2}_{\sigma_1^{fr}}(M)),\\ M_2 &= freeze^{k_1,k_2}_{\sigma_3^{fr}}(freeze^{l'_1,l'_2}_{[\sigma_2^{fr},\sigma_1^{fr}]}({}_{[l'_1,l'_2\circ m'_1,l'_2\circ m'_2]|}M)). \end{aligned}$$

By rule (9),

$$M_1 \rightarrow freeze^{l_1,l_2}_{[\sigma_3^{fr},\sigma_2^{fr}]}(freeze^{h'_1,h'_2}_{\sigma'^{fr}_1}({}_{[l_1,l_2\circ m_1,l_2\circ m_2]+\sigma'^{in}|}M)).$$

By rules (7) and (4),

$$freeze^{l_1,l_2}_{[\sigma_3^{fr},\sigma_2^{fr}]}(freeze^{h'_1,h'_2}_{\sigma'^{fr}_1}({}_{[l_1,l_2\circ m_1,l_2\circ m_2]+\sigma'^{in}|}M)) \overset{+}{\rightarrow}$$

$$freeze^{l''_1,l''_2}_{[[\sigma_3^{fr},\sigma_2^{fr}],\sigma'^{fr}_1]}({}_{[l''_1,l''_2\circ m''_1,l''_2\circ m''_2]\circ([l_1,l_2\circ m_1,l_2\circ m_2]+\sigma'^{in})|}M).$$

Again by rules (7) and (4),

$$M_2 \overset{+}{\rightarrow} freeze^{l'''_1,l'''_2}_{[\sigma_3^{fr},[\sigma_2^{fr},\sigma_1^{fr}]]}({}_{[l'''_1,l'''_2\circ m'''_1,l'''_2\circ m'''_2]\circ[l'_1,l'_2\circ m'_1,l'_2\circ m'_2]|}M).$$

Hence we can conclude by unicity of the canonical term.

— The left-hand side of rule (10) unifies with that of rule (11) yielding the canonical term

$$freeze^{l_1,l_2}_{\sigma_1^{fr}}(M_1)^{j_1}{+}^{j_2}freeze^{m_1,m_2}_{\sigma_2^{fr}}(M_2).$$

Then, applying rules (10) and (11), respectively, we obtain the canonical critical pair $<M_3, M_4>$, with

$$\begin{aligned} M_3 &= freeze^{p_1,p_2}_{\sigma_1^{fr}+\varnothing_{\Sigma_2^{out}}}({}_{[p_1,p_2\circ k_1]|}M_1{}^{j_1}{+}^{j_2}{}_{p_1|}freeze^{m_1,m_2}_{\sigma_2^{fr}}(M_2)),\\ M_4 &= freeze^{p'_1,p'_2}_{\varnothing_{\Sigma_1^{out}}+\sigma_2^{fr}}({}_{p'_1|}freeze^{l_1,l_2}_{\sigma_1^{fr}}(M_1)^{j_1}{+}^{j_2}{}_{[p'_1,p'_2\circ k'_2]|}M_2) \end{aligned}$$

By rules (9), (11) and (4),

$$M_3 \overset{+}{\rightarrow} freeze^{p_1,p_2}_{\sigma_1^{fr}+\varnothing_{\Sigma_2^{out}}}(freeze^{p''_1,p''_2}_{\varnothing_{\Sigma_1^{out}}+\sigma'^{fr}_2}({}_{p''_1\circ[p_1,p_2\circ k_1]|}M_1{}^{j_1}{+}^{j_2}{}_{[p''_1,p''_2\circ k''_2]\circ(p_1+\sigma'^{in})|}M_2)).$$

By rules (7), (4) and (3),

$$freeze^{p_1,p_2}_{\sigma_1^{fr}+\varnothing_{\Sigma_2^{out}}}(freeze^{p''_1,p''_2}_{\varnothing_{\Sigma_1^{out}}+\sigma'^{fr}_2}({}_{p''_1\circ[p_1,p_2\circ k_1]|}M_1{}^{j_1}{+}^{j_2}{}_{[p''_1,p''_2\circ k''_2]\circ(p_1+\sigma'^{in})|}M_2)) \overset{+}{\rightarrow}$$

$$freeze^{l''_1,l''_2}_{[\sigma_1^{fr}+\varnothing_{\Sigma_2^{out}},\varnothing_{\Sigma_1^{out}}+\sigma'^{fr}_2]}(M'_3{}^{j_1}{+}^{j_2}M'_4)$$

https://doi.org/10.1017/S0960129502003687 Published online by Cambridge University Press

with

$$\begin{aligned} M'_3 &= {}_{[l''_1,l''_2\circ m''_1,l''_2\circ m''_2]\circ p''_1\circ[p_1,p_2\circ k_1]|}M_1, \\ M'_4 &= {}_{[l''_1,l''_2\circ m''_1,l''_2\circ m''_2]\circ[p''_1,p''_2\circ k''_2]\circ(p_1+\sigma'^{in})|}M_2. \end{aligned}$$

By rules (9), (10) and (4),

$$M_4 \xrightarrow{+} freeze^{p'_1,p'_2}_{\varnothing_{\Sigma_1^{out}}+\sigma_2^{fr}}(freeze^{p'''_1,p'''_2}_{\sigma'^{fr}_1+\varnothing_{\Sigma_2^{out}}}({}_{[p'''_1,p'''_2\circ k'''_1]\circ(p'_1+\sigma''^{in})|}M_1{}^{j_1}\!+^{j_2}{}_{p'''_1\circ[p'_1,p'_2\circ k'_2]|}M_2)).$$

By rules (7), (4) and (3),

$$freeze^{p'_1,p'_2}_{\varnothing_{\Sigma_1^{out}}+\sigma_2^{fr}}(freeze^{p'''_1,p'''_2}_{\sigma'^{fr}_1+\varnothing_{\Sigma_2^{out}}}({}_{[p'''_1,p'''_2\circ k'''_1]\circ(p'_1+\sigma''^{in})|}M_1{}^{j_1}\!+^{j_2}{}_{p'''_1\circ[p'_1,p'_2\circ k'_2]|}M_2)) \xrightarrow{+}$$

$$freeze^{l'''_1,l'''_2}_{[\varnothing_{\Sigma_1^{out}}+\sigma_2^{fr},\sigma'^{fr}_1+\varnothing_{\Sigma_2^{out}}]}(M''_3{}^{j_1}\!+^{j_2}M''_4),$$

with

$$\begin{aligned} M''_3 &= {}_{[l'''_1,l'''_2\circ m'''_1,l'''_2\circ m'''_2]\circ[p'''_1,p'''_2\circ k'''_1]\circ(p'_1+\sigma''^{in})|}M_1, \\ M''_4 &= {}_{[l'''_1,l'''_2\circ m'''_1,l'''_2\circ m'''_2]\circ p'''_1\circ[p'_1,p'_2\circ k'_2]|}M_2. \end{aligned}$$

Hence we can conclude by unicity of the canonical term. □

5. Soundness

In this section we prove that the specification presented in Section 3 is sound with respect to the model previously defined in Ancona and Zucca (1998b) (summarised in Section 2 and called in this section the *standard* model). In other words, we prove that the interpretations where mixins are seen as functions from input to output components satisfy all the expected properties.

As expected, the proof is independent of the particular core framework on which the standard model is instantiated; in other words, every standard model built on a framework satisfying the properties in Definition 2.3 of Section 2 satisfies all axioms of SP.

Of course, since the specification SP is in many-sorted (positive) conditional logic, it also trivially includes non-standard models (that is, models where mixins are not seen as functions).

Theorem 5.1. For any core framework $\mathscr{F}$, the standard model $\mathscr{M}(\mathscr{F})$ built on top of $\mathscr{F}$ is a model of the algebraic specification SP.

Proof. We show that each axiom in SP is satisfied by $\mathscr{M}(\mathscr{F})$.

(1) For all $A \in Mod(\Sigma^{in})$:
By the definition of reduct and functoriality of Mod),

$$(M_1{}^{j_1}\!+^{j_2}M_2)_{|j_1}(A) = ((M_1{}^{j_1}\!+^{j_2}M_2)(A))_{|j_1}.$$

By the definition of sum,

$$((M_1{}^{j_1}\!+^{j_2}M_2)(A))_{|j_1} = (M_1(A)^{j_1}\!+^{j_2}M_2(A))_{|j_1}.$$

https://doi.org/10.1017/S0960129502003687 Published online by Cambridge University Press

By the amalgamation property,

$$(M_1(A)^{j_1}{+}^{j_2}M_2(A))_{|j_1} = M_1(A).$$

(2) This case is symmetric to (1).

(3) Let us assume by hypothesis that $M_{1|j_i} = M_{2|j_i}$, $i = 1, 2$. Then, for all $A \in Mod(\Sigma^{in})$:
By the definition of reduct and functoriality of *Mod*,

$$M_1(A)_{|j_i} = M_2(A)_{|j_i}, \quad i = 1, 2.$$

By the amalgamation property,

$$M_1(A) = M_2(A).$$

(4) For all $A \in Mod(\Sigma^{in})$, by the definition of reduct and functoriality of *Mod*,

$${}_{id_{\Sigma^{in}}|}M_{|id_{\Sigma^{out}}}(A) = M(A).$$

(5) For all $A \in Mod(\Sigma_2^{in})$:
By the definition of reduct,

$${}_{\sigma_2^{in}|\sigma_1^{in}|}M_{|\sigma_1^{out}|\sigma_2^{out}}(A) = (M(A_{|\sigma_2^{in}|\sigma_1^{in}}))_{|\sigma_1^{out}|\sigma_2^{out}}.$$

By functoriality of *Mod*,

$$(M(A_{|\sigma_2^{in}|\sigma_1^{in}}))_{|\sigma_1^{out}|\sigma_2^{out}} = (M(A_{|\sigma_2^{in}\circ\sigma_1^{in}}))_{|\sigma_1^{out}\circ\sigma_2^{out}}.$$

By the definition of reduct

$$(M(A_{|\sigma_2^{in}\circ\sigma_1^{in}}))_{|\sigma_1^{out}\circ\sigma_2^{out}} = {}_{\sigma_2^{in}\circ\sigma_1^{in}|}M_{|\sigma_1^{out}\circ\sigma_2^{out}}(A).$$

(6) For all $A \in Mod(\Sigma^{in})$:
By the definition of freeze,

$$freeze^{k_1,k_2}_{\sigma_2^{fr}}(freeze^{j_1,j_2}_{\sigma_1^{fr}}(M))(A) = fix(\lambda X.fix(\lambda Y.M((A^{k_1}{+}^{k_2}X_{|\sigma_2^{fr}})^{j_1}{+}^{j_2}Y_{|\sigma_1^{fr}}))).$$

By property (3) of *fix*,

$$fix(\lambda X.fix(\lambda Y.M((A^{k_1}{+}^{k_2}X_{|\sigma_2^{fr}})^{j_1}{+}^{j_2}Y_{|\sigma_1^{fr}})))$$
$$= fix(\lambda X.M((A^{k_1}{+}^{k_2}X_{|\sigma_2^{fr}})^{j_1}{+}^{j_2}X_{|\sigma_1^{fr}})).$$

By the amalgamation property,

$$fix(\lambda X.M((A^{k_1}{+}^{k_2}X_{|\sigma_2^{fr}})^{j_1}{+}^{j_2}X_{|\sigma_1^{fr}}))$$
$$= fix(\lambda X.M((A^{l_1}{+}^{l_2}X_{|[\sigma_2^{fr},\sigma_1^{fr}]})_{|[l_1,l_2\circ m_1,l_2\circ m_2]})).$$

By the definition of freeze and reduct,

$$fix(\lambda X.M((A^{l_1}{+}^{l_2}X_{|[\sigma_2^{fr},\sigma_1^{fr}]})_{|[l_1,l_2\circ m_1,l_2\circ m_2]}))$$
$$= (freeze^{l_1,l_2}_{[\sigma_2^{fr},\sigma_1^{fr}]}({}_{[l_1,l_2\circ m_1,l_2\circ m_2]|}M))(A).$$

https://doi.org/10.1017/S0960129502003687 Published online by Cambridge University Press

(7) For all $A \in Mod(\Sigma^{in})$:
By the definition of freeze,

$$freeze^{j_1,j_2}_{\varnothing_{\Sigma^{out}}}(M)(A) = fix(\lambda X.M(A^{j_1}+^{j_2}X_{|\varnothing_{\Sigma^{out}}})).$$

By the amalgamation property and finite cocompleteness of Mod,

$$fix(\lambda X.M(A^{j_1}+^{j_2}X_{|\varnothing_{\Sigma^{out}}})) = fix(\lambda X.M(A_{|[id_{\Sigma^{in}},\varnothing_{\Sigma^{in}}]})).$$

By property (1) of fix and the definition of reduct,

$$fix(\lambda X.M(A_{|[id_{\Sigma^{in}},\varnothing_{\Sigma^{in}}]})) = ({}_{[id_{\Sigma^{in}},\varnothing_{\Sigma^{in}}]|}M)(A).$$

(8) For all $A \in Mod(\Sigma^{in})$:
By the definition of freeze and reduct:

$$freeze^{j_1,j_2}_{\sigma^{fr}}(M_{|\sigma^{out}})(A) = fix(\lambda X.(M(A^{j_1}+^{j_2}X_{|\sigma^{fr}}))_{|\sigma^{out}}).$$

By property (2) of fix and functoriality of Mod,

$$fix(\lambda X.(M(A^{j_1}+^{j_2}X_{|\sigma^{fr}}))_{|\sigma^{out}}) = (fix(\lambda Y.M(A^{j_1}+^{j_2}Y_{|\sigma^{out}\circ\sigma^{fr}})))_{|\sigma^{out}}.$$

By the amalgamation property and $\sigma'^{fr} \circ \sigma^{in} = \sigma^{out} \circ \sigma^{fr}$,

$$(fix(\lambda Y.M(A^{j_1}+^{j_2}Y_{|\sigma^{out}\circ\sigma^{fr}})))_{|\sigma^{out}} = (fix(\lambda Y.M((A^{k_1}+^{k_2}Y_{|\sigma'^{fr}})_{|id_{\Sigma^{in}}+\sigma^{in}})))_{|\sigma^{out}}.$$

By the definition of freeze and reduct,

$$(fix(\lambda Y.M((A^{k_1}+^{k_2}Y_{|\sigma'^{fr}})_{|id_{\Sigma^{in}}+\sigma^{in}})))_{|\sigma^{out}} = (freeze^{k_1,k_2}_{\sigma'^{fr}}({}_{id_{\Sigma^{in}}+\sigma^{in}|}M))_{|\sigma^{out}}.$$

(9) For all $A \in Mod(\Sigma'^{in})$:
By the definition of freeze and reduct,

$${}_{\sigma^{in}|}(freeze^{j_1,j_2}_{\sigma'^{fr}}(M))(A) = fix(\lambda X.M(A_{|\sigma^{in}}{}^{j_1}+^{j_2}X_{|\sigma'^{fr}})).$$

By the amalgamation property,

$$fix(\lambda X.M(A_{|\sigma^{in}}{}^{j_1}+^{j_2}X_{|\sigma'^{fr}})) = fix(\lambda X.M((A^{k_1}+^{k_2}X_{|\sigma'^{fr}})_{|\sigma^{in}+\sigma'^{in}})).$$

By the definition of freeze,

$$fix(\lambda X.M((A^{k_1}+^{k_2}X_{|\sigma'^{fr}})_{|\sigma^{in}+\sigma'^{in}})) = (freeze^{k_1,k_2}_{\sigma'^{fr}}({}_{\sigma^{in}+\sigma'^{in}|}M))(A).$$

(10) For all $A \in Mod(\Sigma^{in})$:
By the definition of freeze and sum,

$$(freeze^{l_1,l_2}_{\sigma_1^{fr}}(M_1)^{j_1}+^{j_2}freeze^{m_1,m_2}_{\sigma_2^{fr}}(M_2))(A)$$

$$=$$

$$fix(\lambda X_1.M_1(A^{l_1}+^{l_2}X_{1_{|\sigma_1^{fr}}}))^{j_1}+^{j_2}fix(\lambda X_2.M_2(A^{m_1}+^{m_2}X_{2_{|\sigma_2^{fr}}}))$$

By the amalgamation property and property (2) of fix.,

$$fix(\lambda X_1.M_1(A^{l_1}+^{l_2}X_{1_{|\sigma_1^{fr}}}))^{j_1}+^{j_2}fix(\lambda X_2.M_2(A^{m_1}+^{m_2}X_{2_{|\sigma_2^{fr}}}))$$

$$=$$

$$fix(\lambda X.M_1(A^{l_1}+^{l_2}X_{|j_1|\sigma_1^{fr}})^{j_1}+^{j_2}M_2(A^{m_1}+^{m_2}X_{|j_2|\sigma_2^{fr}}))$$

https://doi.org/10.1017/S0960129502003687 Published online by Cambridge University Press

By the amalgamation property,

$$fix(\lambda X.M_1(A^{l_1}+^{l_2}X_{|j_1|\sigma_1^{fr}})^{j_1}+^{j_2}M_2(A^{m_1}+^{m_2}X_{|j_2|\sigma_2^{fr}}))$$

$$=$$

$$fix(\lambda X.M_1((A^{p_1}+^{p_2}X_{|\sigma_1^{fr}+\sigma_2^{fr}})_{|[p_1,p_2\circ k_1]})^{j_1}+^{j_2}M_2((A^{p_1}+^{p_2}X_{|\sigma_1^{fr}+\sigma_2^{fr}})_{|[p_1,p_2\circ k_2]})).$$

By the definition of freeze, sum and reduct,

$$fix(\lambda X.M_1((A^{p_1}+^{p_2}X_{|\sigma_1^{fr}+\sigma_2^{fr}})_{|[p_1,p_2\circ k_1]})^{j_1}+^{j_2}M_2((A^{p_1}+^{p_2}X_{|\sigma_1^{fr}+\sigma_2^{fr}})_{|[p_1,p_2\circ k_2]}))$$

$$=$$

$$(freeze^{p_1,p_2}_{\sigma_1^{fr}+\sigma_2^{fr}}({}_{[p_1,p_2\circ k_1]|}M_1{}^{j_1}+^{j_2}{}_{[p_1,p_2\circ k_2]|}M_2))(A). \qquad \square$$

Theorem 5.1 states that the standard model for the three primitive mixin operations summarised in Section 2 is a model of SP. However, we can also prove a similar result when considering the high-level operations defined in Section 2 and the corresponding axioms defined in Figure 3.

Theorem 5.2. For any core framework $\mathscr{F}$, the standard model built on top of $\mathscr{F}$ and extended to high-level operations is a model of the axioms defined in Figure 3.

Proof. The only non-trivial case concerns the operation of functional composition; proofs for the other high-level operations can be obtained easily by applying the semantic definitions.

In order to prove the soundness of axiom (vi), we first show that the two expressions $M_2 \circ M_1$ and **hide** Σ_1^{out} **in** $(M_1 \oplus M_2)$ have the same type (recall that $M_i : \Sigma_i^{in} \to \Sigma_i^{out}$, $i = 1, 2$, and that $\Sigma_1^{out} = \Sigma_2^{in} \setminus \Sigma_2^{out}$ and $(\Sigma_1^{in} \setminus \Sigma_1^{out}) \cap \Sigma_2^{out} = \varnothing$). To this end we use the following (easy to prove) properties of boolean categories:

— $(\Sigma_1 \cup \Sigma_2) \setminus \Sigma_3 = (\Sigma_1 \setminus \Sigma_3) \cup (\Sigma_2 \setminus \Sigma_3)$
— $\Sigma_1 \setminus (\Sigma_2 \cup \Sigma_3) = (\Sigma_1 \setminus \Sigma_2) \setminus \Sigma_3$
— $\Sigma_1 \cap \Sigma_2 = \varnothing \Rightarrow \Sigma_1 \setminus \Sigma_2 = \Sigma_1$
— $\Sigma_1 \setminus (\Sigma_1 \setminus \Sigma_2) = \Sigma_1 \cap \Sigma_2$
— $\Sigma_1 \subseteq \Sigma_2 \Rightarrow \Sigma_1 \setminus \Sigma_2 = \varnothing$

By the side conditions, $\Sigma_1^{out} = \Sigma_2^{in} \setminus \Sigma_2^{out}$, therefore $\Sigma_1^{out} \cap \Sigma_2^{out} = \varnothing$, so the expression $M_1 \oplus M_2$ is well-typed and has type $\Sigma_1^{in} \cup (\Sigma_2^{in} \cap \Sigma_2^{out}) \to \Sigma_1^{out} \cup \Sigma_2^{out}$. Indeed,

$$\begin{aligned}
\Sigma_1^{in} \cap \Sigma_2^{fr} &= ((\Sigma_1^{in} \setminus \Sigma_1^{out}) \cap (\Sigma_2^{out} \setminus \Sigma_2^{in})) \cup ((\Sigma_1^{in} \cap \Sigma_1^{out}) \cap (\Sigma_2^{out} \setminus \Sigma_2^{in})) = \varnothing;\\
\Sigma_2^{in} \cap \Sigma_2^{fr} &= \Sigma_2^{in} \cap (\Sigma_2^{out} \setminus \Sigma_2^{in}) = \varnothing;\\
\Sigma_1^{in} \cap \Sigma_1^{fr} &= \Sigma_1^{in} \cap (\Sigma_1^{out} \setminus \Sigma_1^{in}) = \varnothing;\\
(\Sigma_2^{in} \cap \Sigma_2^{out}) \cap \Sigma_1^{fr} &= (\Sigma_2^{in} \cap \Sigma_2^{out}) \cap (\Sigma_1^{out} \setminus \Sigma_1^{in}) = \varnothing;\\
(\Sigma_2^{in} \setminus \Sigma_2^{out}) \setminus \Sigma_1^{fr} &= \Sigma_1^{out} \setminus (\Sigma_1^{out} \setminus \Sigma_1^{in}) = \Sigma_1^{out} \cap \Sigma_1^{in}.
\end{aligned}$$

Therefore,

$$\begin{aligned}
(\Sigma_1^{in} \cup \Sigma_2^{in}) \setminus (\Sigma^{fr}{}_1 \cup \Sigma^{fr}{}_2) &= ((\Sigma_1^{in} \cup \Sigma_2^{in}) \setminus \Sigma_2^{fr}) \setminus \Sigma_1^{fr}\\
&= (\Sigma_1^{in} \cup \Sigma_2^{in}) \setminus \Sigma_1^{fr}
\end{aligned}$$

https://doi.org/10.1017/S0960129502003687 Published online by Cambridge University Press

$$\begin{aligned} &= \Sigma_1^{in} \cup (\Sigma_2^{in} \setminus \Sigma_1^{fr}) \\ &= \Sigma_1^{in} \cup (\Sigma_1^{out} \cap \Sigma_1^{in}) \cup (\Sigma_2^{in} \cap \Sigma_2^{out}) \\ &= \Sigma_1^{in} \cup (\Sigma_2^{in} \cap \Sigma_2^{out}). \end{aligned}$$

Now $\Sigma_1^{out} \subseteq \Sigma_1^{out} \cup \Sigma_2^{out}$, hence the expression **hide** Σ_1^{out} **in** $(M_1 \oplus M_2)$ is well-typed and has type $(\Sigma_1^{in} \setminus \Sigma_1^{out}) \cup (\Sigma_2^{in} \cap \Sigma_2^{out}) \to \Sigma_2^{out}$. Indeed,

$$(\Sigma_1^{out} \cup \Sigma_2^{out}) \setminus \Sigma_1^{out} = \Sigma_2^{out}$$

and

$$(\Sigma_1^{in} \cup (\Sigma_2^{in} \cap \Sigma_2^{out})) \setminus \Sigma_1^{out} = (\Sigma_1^{in} \setminus \Sigma_1^{out}) \cup (\Sigma_2^{in} \cap \Sigma_2^{out}).$$

Then we can show that the two expressions have the same semantics; by axioms (i), (iv) and (6) we have:

$$\begin{aligned} &\textbf{hide } \Sigma_1^{out} \textbf{ in } (M_1 \oplus M_2) \\ &\qquad = \quad \textbf{hide } \Sigma_1^{out} \textbf{ in } (freeze_{\Sigma^{in} \cap \Sigma^{fr}}(\Sigma^{in}|M_1 + \Sigma^{in}|M_2)) \\ &\qquad = \quad (freeze_{\Sigma_1^{out} \cap (\Sigma^{in} \setminus \Sigma^{fr})}(freeze_{\Sigma^{in} \cap \Sigma^{fr}}(\Sigma^{in}|M_1 + \Sigma^{in}|M_2)))_{|\Sigma^{out} \setminus \Sigma_1^{out}} \\ &\qquad = \quad (freeze_{(\Sigma_1^{out} \cap (\Sigma^{in} \setminus \Sigma^{fr})) \cup (\Sigma^{in} \cap \Sigma^{fr})}(\Sigma^{in}|M_1 + \Sigma^{in}|M_2)_{|\Sigma_2^{out}} \end{aligned}$$

where $\Sigma^{in} = \Sigma_1^{in} \cup \Sigma_2^{in}$.

Now

$$\begin{aligned} (\Sigma_1^{out} \cap (\Sigma^{in} \setminus \Sigma^{fr})) \cup (\Sigma^{in} \cap \Sigma^{fr}) &= \Sigma_1^{out} \cap (\Sigma_1^{in} \cup (\Sigma_2^{in} \cap \Sigma_2^{out}) \cup (\Sigma_1^{out} \setminus \Sigma_1^{in})) \\ &= (\Sigma_1^{out} \cap \Sigma_1^{in}) \cup (\Sigma_1^{out} \setminus \Sigma_1^{in}) \\ &= \Sigma_1^{out}. \end{aligned}$$

Therefore, by definition and by the amalgamation property:

$$\begin{aligned} &(freeze_{\Sigma_1^{out}}(\Sigma^{in}|M_1 + \Sigma^{in}|M_2))_{|\Sigma_2^{out}} \\ &\qquad = \quad \lambda A.(fix(\lambda B.F_1((A + B_{|\Sigma_1^{out}})_{|\Sigma_1^{in}}) + F_2((A + B_{|\Sigma_1^{out}})_{|\Sigma_2^{in}})))_{|\Sigma_2^{out}} \\ &\qquad = \quad \lambda A.(fix(\lambda B.F_1(A_{|\Sigma_1^{in} \setminus \Sigma_1^{out}} + B_{|\Sigma_1^{in} \cap \Sigma_1^{out}}) + F_2(A_{|\Sigma_2^{in} \cap \Sigma_2^{out}} + B_{|\Sigma_1^{out}})))_{|\Sigma_2^{out}} \end{aligned}$$

Set

$$f = fix(\lambda B.F_1(A_{|\Sigma_1^{in} \setminus \Sigma_1^{out}} + B_{|\Sigma_1^{in} \cap \Sigma_1^{out}}) + F_2(A_{|\Sigma_2^{in} \cap \Sigma_2^{out}} + B_{|\Sigma_1^{out}})).$$

Then, by property (1) of *fix* and the amalgamation property, we obtain

$$\begin{aligned} \lambda A.(F_1(A_{|\Sigma_1^{in} \setminus \Sigma_1^{out}} + f_{|\Sigma_1^{in} \cap \Sigma_1^{out}}) + F_2(A_{|\Sigma_2^{in} \cap \Sigma_2^{out}} + f_{|\Sigma_1^{out}}))_{|\Sigma_2^{out}} \\ = \quad \lambda A.F_2(A_{|\Sigma_2^{in} \cap \Sigma_2^{out}} + f_{|\Sigma_1^{out}}). \end{aligned}$$

Finally, we can conclude by property (2) of *fix* since

$$f_{|\Sigma_1^{out}} = fix(\lambda B.F_1(A_{|\Sigma_1^{in} \setminus \Sigma_1^{out}} + B_{|\Sigma_1^{in} \cap \Sigma_1^{out}})).$$

□

https://doi.org/10.1017/S0960129502003687 Published online by Cambridge University Press

6. Conclusion

We have given an axiomatic characterisation of the notion of a mixin module, following the spirit of the seminal paper Bergstra *et al.* (1990). Intepreting axioms as rewriting rules, we have obtained a confluent and strongly normalizing rewriting system, which provides a reduction semantics for the language. Moreover, we have shown that the given axiomatisation is sound with respect to the model previously defined in Ancona and Zucca (1998b) where mixins are interpreted as functions from input to output components. Finally, we have used the axioms to prove many other expected properties of the mixin operators.

This paper is meant to be a continuation of Ancona and Zucca (1998b); a preliminary version was presented in Ancona and Zucca (1998a). In Ancona and Zucca (1997) and Ancona (2000) we explored instantiations of the formal framework presented in Ancona and Zucca (1998b) and here in concrete cases. In Ancona and Zucca (1997) we gave the translation in terms of our operators of various overriding mechanisms present in programming languages (including invocation via `super`), and are able, in this way, to formalise the relation between these different mechanisms. In Ancona (2000), a concrete mixin language was obtained by fixing as core language a simple functional language. Finally, a comprehensive survey of a large part of our work on mixins is given in Ancona (1998).

Even though inspired by Bergstra *et al.* (1990), the work presented here is different with respect to the well-established algebraic treatment of module composition. Here we address the problem of combining together programming rather than specification modules, so we have to consider, together with classical operators (like export and renaming), new operators (like freeze) related to the notions of module modification and extension that hardly make sense in the context of specifications.

Several papers (Cardelli 1997; Harper and Lillibridge 1994; Jones 1996; Leroy 1994; Leroy 2000) have pointed out the importance of modularity mechanisms independent of the underlying core language and supporting the notions of separate compilation and linking. As far as we are aware, our proposal of axiomatisation for mixins is the first that supports these principles.

What is missing in this paper is, on the one hand, a true calculus for mixins and, on the other, a more strict integration of our framework with the notions of type system and type checking.

For the first point, the work presented here already provides a language of mixin expressions with a reduction semantics. However, this language is, more appropriately, a language schema; indeed, it needs to be instantiated over a core language providing, for example, mixin constants corresponding to basic mixin definitions, as we have shown in Section 2.2. In Ancona and Zucca (1999) and Ancona and Zucca (2001) we took a less abstract approach, assuming that mixin constants are collections of definitions of components that are expressions of the core language (this corresponds to taking as signatures typed families of symbols). However, the definition of module expressions is still parametric in that of core expressions; this independence was achieved by adopting an approach based on implicit substitutions. In this way, we were able to define a true

calculus of mixins, that is, an analogue of λ-calculus where the basic combinators, instead of application and λ-abstraction, are the mixin primitives (sum, reduct and freeze), together with a form of dot notation for accessing mixin components. We proved confluence and soundness of the type system for this calculus, and we also showed that λ-calculus can be effectively encoded in this mixin calculus, as the result about functional composition proved in this paper already suggests. Other related proposals for module calculi can be found in Wells and Vestergaard (2000) and Machkasova and Turbak (2000).

As for type checking, an important point to be investigated is how to integrate the mixin-based approach with the possibility of specifying various kinds of *type constraints* in signatures: for instance, subtyping constraints, type sharing and *manifest types* (Leroy 1994). A promising direction seems to be to use some formal notion of signature parameterised by a *type system*, as defined in Ancona (1999).

References

Adàmek, J., Herrlich, H. and Strecker, G. (1990) *Abstract and Concrete Categories*, Pure and Applied Mathematics, John Wiley & Sons.

Ancona, D. (1998) *Modular Formal Frameworks for Module Systems*, Ph. D. thesis, Dipartimento di Informatica, Università di Pisa.

Ancona, D. (1999) An algebraic framework for separate type-checking. In: Fiadeiro, J. (ed.) WADT'98 (13th Workshop on Algebraic Development Techniques). *Springer-Verlag Lecture Notes in Computer Science* **1589** 1–15.

Ancona, D. (2000) MIX(FL): a kernel language of mixin modules. In: Rus, T. (ed.) AMAST 2000 – Algebraic Methodology And Software Technology. *Springer-Verlag Lecture Notes in Computer Science* **1816** 454–468.

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

Ancona, D. and Zucca, E. (1998a) An algebra of mixin modules. In: Parisi Presicce, F. (ed.) Recent Trends in Algebraic Development Techniques (12th Intl. Workshop, WADT'97 - Selected Papers). *Springer-Verlag Lecture Notes in Computer Science* **1376** 92–106.

Ancona, D. and Zucca, E. (1998b) A theory of mixin modules: Basic and derived operators. *Mathematical Structures in Computer Science* **8** (4) 401–446.

Ancona, D. and Zucca, E. (1999) A primitive calculus for module systems. In: Nadathur, G. (ed.) PPDP'99 - Principles and Practice of Declarative Programming. *Springer-Verlag Lecture Notes in Computer Science* **1702** 62–79.

Ancona, D. and Zucca, E. (2001) A calculus of module systems. *Journal of Functional Programming* (to appear).

Astesiano, E., Broy, M. and Reggio, G. (1999) Algebraic specification of concurrent systems. In: Astesiano, E., Kreowski, H.-J. and Krieg-Brueckner, B. (eds.) *Algebraic Foundations of Systems Specification, IFIP State-of-the-Art Report*, Berlin, Springer Verlag.

Banavar, G. and Lindstrom, G. (1996) An application framework for module composition tools. In: ECOOP '96. *Springer-Verlag Lecture Notes in Computer Science* **1098** 91–113.

Bergstra, J. A., Heering, J. and Klint, P. (1990) Module algebra. *Journ. ACM* **37** (2) 335–372.

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: ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1990. *SIGPLAN Notices* **25** (10) 303–311.
Bracha, G. and Lindstrom, G. (1992) Modularity meets inheritance. In: *Proc. International Conference on Computer Languages*, San Francisco, IEEE Computer Society 282–290.
Cardelli, L. (1997) Program fragments, linking, and modularization. In: *ACM Symp. on Principles of Programming Languages 1997*, ACM Press 266–277.
Cook, W. (1989) *A Denotational Semantics of Inheritance*, Ph. D. thesis, Dept. Comp. Sci., Brown University.
Crary, K., Harper, R. and Puri, S. (1999) What is a recursive module? In: *PLDI'99 - ACM Conf. on Programming Language Design and Implementation.*
Duggan, D. and Sourelis, C. (1996) Mixin modules. In: Intl. Conf. on Functional Programming, Philadelphia. *SIGPLAN Notices* **31** (6) 262–273.
Duggan, D. and Sourelis, C. (1998) Parameterized modules, recursive modules, and mixin modules. In: *1998 ACM SIGPLAN Workshop on ML*, Baltimore Maryland, ACM Press 87–96.
Ehrig, H. and Mahr, B. (1985) *Fundamentals of Algebraic Specification 1. Equations and Initial Semantics*, EATCS Monograph in Computer Science **6**, Springer-Verlag.
Findler, R. B. and Flatt, M. (1998) Modular object-oriented programming with units and mixins. In: *Intl. Conf. on Functional Programming 1998.*
Flatt, M., Krishnamurthi, S. and Felleisen, M. (1998) Classes and mixins. In: *ACM Symp. on Principles of Programming Languages 1998* 171–183.
Goguen, J. A. and Burstall, R. M. (1992) Institutions: Abstract model theory for computer science. *Journ. ACM* **39** 95–146.
Harper, R. and Lillibridge, M. (1994) A type theoretic approach to higher-order modules with sharing. In: *ACM Symp. on Principles of Programming Languages 1994*, ACM Press 127–137.
Jones, M. P. (1996) Using parameterized signatures to express modular structure. In: *ACM Symp. on Principles of Programming Languages 1996*, St. Petersburg Beach, Florida, ACM Press 68–78.
Leroy, X. (1994) Manifest types, modules and separate compilation. In: *ACM Symp. on Principles of Programming Languages 1994*, ACM Press 109–122.
Leroy, X. (2000) A modular module system. *Journ. of Functional Programming* **10** (3).
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) 1–30.
Machkasova, E. and Turbak, F. A. (2000) A calculus for link-time compilation. In: Smolka, G. (ed.) European Symposium on Programming 2000. *Springer-Verlag Lecture Notes in Computer Science* **1782** 260-274.
Milner, R., Tofte, M. and Harper, R. (1990) *The Definition of Standard ML*, The MIT Press.
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., Sokołowski, S. and Tarlecki, A. (1992) Towards formal development of programs from algebraic specifications: Parameterisation revisited. *Acta Informatica* **29** (8) 689–736.
Sannella, D. and Tarlecki, A. (1986) Extended ML: an institution-independent framework for formal program development. In: Proc. Workshop on Category Theory and Computer Programming. *Springer-Verlag Lecture Notes in Computer Science* **240** 364–389.
Sannella, D. and Tarlecki, A. (1988) Towards formal development of programs from algebraic specifications: Implementations revisited. *Acta Informatica* **25** 233–281.
Sannella, D. and Wallen, L. (1992) A calculus for the construction of modular Prolog programs. *Journ. of Logic Programming* **12** 147–177.
Wells, J. B. and Vestergaard, R. (2000) Confluent equational reasoning for linking with first-class primitive modules. In: Smolka, G. (ed.) European Symposium on Programming 2000. *Springer-Verlag Lecture Notes in Computer Science* **1782** 412-428.