Modules for a model-oriented specification language: A proposal for MetaSoft

Andrzej Tarlecki

Institute of Computer Science
Polish Academy of Sciences
Warsaw, Poland.

This is a preliminary sketch of rather tentative ideas.
I would be grateful for any criticism and/or comment.

1 Introduction

MetaSoft [MSoft 90] is a project intended to provide mathematical foundations, methodological basis and computer support tools for software design, development and validation. One, at present perhaps the most important part of the project is devoted to the development of a formal language to build software specifications. In this paper we propose structuring facilities for this language, necessary to make specifications easier to construct, understand and use.

The MetaSoft specification language is primarily *model-oriented*. The user is intended to construct a *model* of the software system being specified rather than to describe its properties in an axiomatic way. *Axiomatic specifications*, although not banned entirely in MetaSoft, are certainly pushed out to its peripheries. This is much as in VDM [BjJ 78], [BjJ 82], [Jon 86], on which MetaSoft builds in many ways. In MetaSoft, the software model constructed by the user is *denotational*, which for us means simply its *compositionality* [ScS 71]. We refrain from using the full technical machinery of standard denotational semantics with reflexive domains [Sc 76], [Sc 82] and continuations [Gor 79]. The "naive" semantic foundations are provided by [BlT 83]. Most roughly, a model of a software system in MetaSoft consists of an algebra of system denotations, a similar algebra of syntax and a homomorphism from the syntax to the denotations (*denotational semantics*). We propose a methodology where the algebra of denotations is developed in the first place, and an appropriate syntax for it is constructed later [Bl 89], [BlTT 91].

The main task of the MetaSoft specification language mentioned above is to provide convenient means for formal definitions of algebras, such as the algebras of denotations around which software models are built. The core of the language is an applicative, strongly-typed definitional language as presented in [BBP 90a]. Using this language, the user can define one by one semantic domains (or just sets) represented as *symbolic types* and values (constants and functions) of these types. Since an algebra is just an environment of named sets (carriers of the algebra) and functions on these sets (operations of the algebra) this is sufficient to define algebras, and hence model-oriented specifications. A similarly "flat" definitional language forms the core of VDM, currently given a formal semantic definition [Lar 89].

Unfortunately, any application of such a flat definitional language to practical examples results in a long list of definitions, which is usually much too large to be manageable in any

way. Therefore, the language must be augmented by some facilities to conveniently structure such lists of definitions. This is, of course, hardly a new idea. Many specification languages, starting with CLEAR [BG 80] and then including for example CIP-L [Bau 85], Larch [GHW 85], COLD-K [FJKR 87], ASL [SW 83], [Wir 86], ACT TWO [EM 90], RAISE [NHWG 88] and many others, have been designed for the very purpose of supporting building formal specifications in a structured manner.

In our view, much work on structuring specifications done in the area of algebraic specification addressed the fundamental issues of manipulating specifications, rather than more down-to-earth, specific issues of a specification language design. One aspect of this is that much of this work is rather abstract and independent of many details of the underlying formalism to build elementary specifications. This has been made explicit, for example, for the specification language ASL [SW 83] redefined in [ST 88] in the framework of an arbitrary institution [GB 84]. In contrast, the purpose of this proposal is to define structuring mechanisms for a particular definitional language. Of course, we freely build on the more fundamental work mentioned above.

However, the main source of inspiration for the exact design of the modularisation mechanisms presented in this paper comes from the area of programming, rather than specification languages. We believe that this is appropriate for a model-oriented specification language, where the main task is to define a certain algebra by providing explicit definitions of all its components. This is much the same as in any programming language, where the task is to define a coherent collection of data types and programs (functions, procedures) to perform required operations. The need for adequate structuring mechanisms has been recognized in this area quite early, and by now many modern programming languages, starting perhaps with Simula [DMN 70] and including Modula [Wirth 88], CLU [Lis 81], Ada [Ada 80] and Standard ML [MTH 90] provide some notion of a program module to allow the programmer to structure the code being written.

The proposal presented in this paper unashamedly follows the modularisation concepts and ideas of the Standard ML programming language [MacQ 86], [MTH 90]. Of course, there are many differences. Some of them follow from the different underlying core definitional language. Some others result from a different design decisions concerning for example the type-visibility rules of the language. Finally, some generalisations are made as well, for example admitting higher-order parameterised modules.

We have already used the modularisation facilities of Standard ML in another project with aims broadly similar to MetaSoft, namely in the design of the Extended ML specification language and methodology [ST 89], [ST 91a]. It should be stressed, though, that the role of modularisation units (called *functors* and *structures* in ML) in Extended ML is quite different from their interpretation in the current proposal. An Extended ML functor is not a piece of a specification; rather, it forms a phase in the development of an independent programming task specified by the functor heading and (perhaps only partially as yet) implemented by its body. Extended ML functors are not used to structure specifications; they are used to structure program development.

The paper is organised as follows. We start by sketching the overall ideas of the modularisation proposal, phrased in rather general terms of many-sorted signatures, algebras and their arbitrary definitions (Section 2). In Section 3 we recall the core definitional language of MetaSoft as presented in [BBP 90a]. Then, in Section 4 we sketch how the general ideas of Section 2 may be realised in the specific framework of this definitional language. Clearly, in the limited space given here it is impossible to present the proposal in full technical detail — we just attempt to sketch the major decisions, and refer the interested reader to [Tar 92] for all the details and a complete technical definition. Finally, in Section 5 we point out some alternatives to two crucial design decisions taken in the proposal presented in this paper, and discuss a number of further developments we view useful or even necessary for the practical applications of the MetaSoft definitional language with modules.

Acknowledgements

I am grateful to the MetaSoft group, and in particular to Andrzej Blikle, for many stimulating discussions — this paper has been written as a contribution to the MetaSoft project. Jacek Leszczyłowski has influenced some particular technical decisions by constantly nodging me to point out a double role of algebra interfaces in the module language proposed here. Much of the work presented follows what I have learnt in a continuing close collaboration with Don Sannella on many issues of software specification and development. The core of the ideas in this paper crystalized during my stays at the Department of Computer Science of the University of Manchester in numerous discussions with Cliff Jones and John Fitzgerald, which I found very stimulating and fruitful.

The research has been partially supported by a grant from the Polish Academy of Sciences and by a grant from the Wolfson Foundation (during my stays in Manchester in May–July 1990 and April–June 1991).

2 Basic ideas — mechanisms to define algebras

The overall aim of the proposal is to introduce a language to define algebras.

An *algebra* is a collection of data classified into different sorts (*carriers* of the algebra) and functions to build and manipulate these data (*operations* of the algebra). The carriers and the operations are *named.* The collection of the names of carriers and operations, the latter equipped with the sorts of arguments and results, forms a *signature* of the algebra.

We will omit here the standard technical definition of many-sorted algebras and their signatures — for example, [EM 85] is a standard reference for a tedious presentation of all the details of one of the essentially equivalent versions of these concepts and some related theory.

2.1 Simple algebra definitions

The formalism we are designing here is a declarative language in the usual sense, where the user gradually builds up an environment of the entities of interest — here, an environment of algebras with explicitly given signatures. The entities are defined using the mechanisms provided by the language, and then stored in the environment by binding them to *identifiers.*

The basic construction available to the user is to define an algebra by listing its components (carriers and operations) defined in turn using an underlying core language.[1]

```
algebra A : ASig =                          where  signature ASig =
        zielone = integer × {0}                        sorts  zielone, czerwone
        czerwone = integer × {1}                       opns   z_zero : zielone
        z_zero = ⟨0, 0⟩                                       zmien : zielone → czerwone
        zmien.⟨n, 0⟩ = ⟨n, 1⟩                        end
end
```

Two other examples of algebras over the same signature are:

```
algebra A' : ASig =                         algebra A'' : ASig =
        zielone = integer × {0}                     zielone = integer
        czerwone = integer × {1}                    czerwone = bool
        z_zero = ⟨0, 0⟩                             z_zero = 0
        zmien.⟨n, 0⟩ = ⟨n + 1, 1⟩                  zmien.n = is_even(n)
end                                         end
```

[1] We use some *ad hoc,* though hopefully sufficiently clear notation in the examples below and throughout the rest of the paper. No final syntax for the MetaSoft language exists as yet — according to the methodology we support, the mechanisms and their semantics which form the real essence of the formalism are designed first.

Nondescriptive identifiers are used below to name algebra components. For our purposes, they seem at least as mnemonic as $s1$, t_{39}, c'', f, or *factorial.*

Following the notational convention of MetaSoft, in the above definitions and throughout the rest of the paper we use the dot notation to denote function application.

Once an algebra is introduced into the overall algebra environment, the algebra and its components may be used in further definitions. Algebra components are referred to using the "dot notation". For instance, in the context of the above definition of an algebra A, $A.zielone$ stands for integer $\times$ $\{0\}$, $A.czerwone$ for integer $\times$ $\{1\}$, and $A.zmien$ for the function mapping any $\langle n, 0\rangle \in$ integer $\times$ $\{0\}$ to $\langle n, 1\rangle$. This is consistent with the dot notation used for the function application. In fact, an algebra may be viewed as a "small environment" mapping the names of the components of the algebra (listed in its signature) to their semantic meanings.

For example, in the context of the above definition of A, we can define:

```
algebra B : BSig =                    where signature BSig =
        niebieskie = A.zielone                sorts  niebieskie, czarne, nowy
        czarne = A.czerwone                   opns   cz_zero : czarne
        nowy = bool                                  zmien : niebieskie → czarne
        cz_zero = (A.zmien).(A.z_zero)        end
        zmien.x = (A.zmien).x
end
```

To keep definitions like the one of B above more self-contained, we require the user to indicate explicitly the algebras from the environment that may be referred to in the particular definition. Thus, actually the correct form of the definition of B should be:

```
algebra B : BSig =
  using A
        niebieskie = A.zielone
        czarne = A.czerwone
        nowy = bool
        cz_zero = (A.zmien).(A.z_zero)
        zmien.x = (A.zmien).x
end
```

The signature of A may be grabbed from the environment, so it is not repeated explicitly here.

In the MetaSoft definitional language, algebras are intended to play the role of modules — self-contained, encapsulated specification units, with the access to their actual contents limited by an explicitly given interface. The role of such interfaces is played by algebra signatures.

The signature of an algebra determines the names of the algebra components. We "know" that the above algebra A has exactly four components, named *zielone*, *czerwone*, *z_zero*, and *zmien*. Perhaps even more importantly, the signature determines some information about what the components are and how they can be used. $A.zielone$ and $A.czerwone$ are sorts of data, $A.z\_zero$ is a constant in $A.zielone$, and $A.zmien$ is a function mapping data in $A.zielone$ to data in $A.czerwone$. This information has been used in the definition of B above. The definition of B may be proved well-formed using only the information given in the signature $ASig$ of A.

This is in contrast with the following definitions:

```
algebra B1 : BSig =
  using A
        niebieskie = integer × {0}
        czarne = integer × {1}
        nowy = bool
        cz_zero = (A.zmien).(A.z_zero)
        zmien.x = (A.zmien).x
end
```

```
algebra B2 : BSig =
  using A
        niebieskie = A.zielone
        czarne = A.czerwone
        nowy = bool
        cz_zero = ⟨0, 1⟩
        zmien.⟨n, 0⟩ = ⟨n, 1⟩
end
```

In the definition of $B1$ we have defined *niebieskie* as integer $\times$ $\{0\}$ and *czarne* as integer $\times$ $\{1\}$. The information in the signature *ASig* ensures that *A.zmien* maps *A.zielone* to *A.czerwone*, and *A.z_zero* is in *A.zielone* — but nothing more. Consequently, to justify that the definitions of *cz_zero* and *zmien* in $B1$ are well-formed, that is, to justify that indeed $B1.cz\_zero$ is in $B1.czarne$ and $B1.zmien$ maps $B1.niebieskie$ into $B1.czarne$, we have to look into the "body" of A and check the exact definitions of *zielone* and *czerwone* there. The situation with the definition of $B2$ is quite similar, although the reason that forces us to look into the definitions of *zielone* and *czerwone* in A seems subtly different here.

We consider only the original definition of B as acceptable, and reject the definitions of $B1$ and $B2$. This is the consequence of the intention to use the interfaces (algebra signatures) to control the information available about the algebras defined and to abstract away from the "internal" details of particular algebra definitions. The well-formedness of the definitions using algebras already defined is justified entirely on the basis of the information in the interfaces. In this way an algebra interface insulates the environment using the algebra from the particular algebra definition. Well-formed references to algebra components remain well-formed even if the components of the algebra are changed (as long as the signature remains the same).

This view may be sometimes rather restrictive. In particular, it is impossible to indicate in algebra signatures that some carriers are identical, and that two algebras "share" some of them. For instance, given the algebras A and B, the following attempt to combine them is not well-formed (for example, the definition of *zero2* cannot be shown to yield a value in *kolor2*):

```
signature ABSig =
    sorts kolor1, kolor2, kolor3
    opns  zero1 : kolor1
          zero2 : kolor2
          zmien : kolor1 → kolor2
end
```

```
algebra AB : ABSig =
  using A, B
        kolor1 = A.zielone   ( = B.niebieskie)
        kolor2 = A.czerwone  ( = B.czarne)
        kolor3 = B.nowy
        zero1 = A.z_zero
        zero2 = B.cz_zero
        zmien = A.zmien      ( = B.zmien)
end
```

Without relaxing unduly this rather strict view, we will introduce some mechanism to make such definitions possible (by enriching the information in algebra interfaces) in Section 4.

Similar comments apply to the analysis of two algebras over the same signature, like A and A' over the signature *ASig*. Even though the names of the corresponding carriers are the same (the algebras are over the same signature) the carriers themselves may be quite different (compare A and A' with A'' above). Consequently, we cannot allow operations of one algebra to be applicable to data in the carriers of the other. Just as in the argument above, this applies even if the carriers happen to be identical. For example, in the context of the above definitions, $A'.zmien$ cannot be applied to $A.z\_zero$, the expression $(A'.zmien).(A.z\_zero)$ is not well-formed.

No axioms in interfaces

When arguing about properties of algebras being defined, we may want to state formally such properties directly in algebra definitions and interfaces. For example, we may want to make it explicit that $AB.zero2 = (AB.zmien).(AB.zero1)$ in the algebra AB we attempted to define above. To show this, we need to know that $B.cz\_zero = (A.zmien).(A.z\_zero)$, and so, even this very simple fact needs some rather involved argument, where three algebras defined above are referred to.

This may lead one to be tempted to allow such properties to be put explicitly in algebra interfaces. We believe that in a specification language, like the one we are defining here, there is no need for such "axioms" in interfaces. To support this view, let us point out that in a model-oriented specification language algebras are built to be used as specifications, not to be checked against any specifications. This is quite in contrast with the situation in development

formalisms, like Extended ML, where the starting point is a (property-oriented) specification, and the task is to build an algebra that would satisfy (or "fit") this specification.

In our language, interfaces control the flow of information used to justify the well-formedness of algebra definitions, not their semantic correctness w.r.t. some given *a priori* requirements specification. We assume (cf. Section 3) that in our core definitional language some static "type" information is sufficient to justify the well-formedness of definition. No further semantic properties of algebras and their components need to be explicitly included in interfaces.

Of course, the decision not to put axioms in algebra interfaces should not prevent the user from proving *a posteriori* some properties of the algebras forming a model-oriented specification. We do not give any explicit tool for this in our proposal. In our view, this can be done at a meta-level, for example by labelling the interfaces of particular algebras with some property-oriented information (e.g. formulae of a suitable logic the algebra is checked to satisfy). This is much as in Hoare's verification logic for iterative programs [Hoa 69] where the properties (of the state of computation) label the control points of the program being verified. Notice that, again as in the verification of iterative programs, different intermittent properties of algebra components may be useful to prove different overall properties of the entire specification. Unlike in formalisms to develop software systems, it seems rather unrealistic to expect that every algebra in its role of a model-oriented specification would come with a fixed list of properties stating all and only relevant information about the algebra.

2.2 Parametric algebra definitions

It is practically important to be able to make some algebra definitions reusable by allowing them to be parameterised by other algebras they depend on. This yields "parametric algebras" that are simply functions mapping algebras to algebras. The domain of such a function is determined by algebra interfaces the parameters are required to fit. Notice that such parametric algebras play here the role of parameterised (model-oriented) specifications. This is only superficially in contrast with the ideas of [SST 92], [ST 91b].

For example, the dependence of the algebra B on A (cf. Section 2.1) may be made functional as follows:

algebra $BF(X : ASig) : BSig =$
$\quad niebieskie = X.zielone$
$\quad czarne = X.czerwone$
$\quad nowy = \mathsf{bool}$
$\quad cz\_zero = (X.zmien).(X.z\_zero)$
$\quad zmien.x = (X.zmien).x$
end

Now, the algebra B may be defined by a straightforward application of BF to A, or as we will sometimes say, by *instantiating* the parametric algebra BF with the actual parameter A for the formal parameter X:

algebra $B = BF(X \mapsto A)$

When instantiating a parametric algebra as in the above definition, we require that the actual parameters are explicitly attached to their corresponding formal identifiers used in the parametric algebra definition (consequently, they may be listed in an arbitrary order). This seems well-justified in a formalism to define large entities like algebras or modules (cf. [LB 88] where this point of view is taken, argued for, and pushed even further by treating bindings like $X \mapsto A$ above as usual data the language manipulates).

To check that the above definition is well-formed, it is sufficient to realise that the interface of A, the argument for BF in the algebra expression used, coincides with the required interface of the parameter X of BF indicated in its definition. Notice that we do not need to state again

that the signature of $BF(X \mapsto A)$, and hence that of B, is $BSig$ — this can be straightforwardly deduced from the result signature in the declaration of BF.

The parametric algebra BF may be used to define other algebras over the signature $BSig$ by instantiating it with different actual parameters (fitting the required parameter interface). For example (A' and A'' are $ASig$-algebras defined in Section 2.1):

algebra $B' = BF(X \mapsto A')$
algebra $B'' = BF(X \mapsto A'')$

Complete information about how a parametric algebra can be used is given by its heading: the list of formal parameters with their interfaces and the result signature. The former gives the required interfaces of the actual parameters the parametric algebra may be instantiated with, and the latter determines the signature of the result of such instantiations. Thus, a parametric-algebra heading, like $(X : ASig) : BSig$ for BF, is a complete parametric algebra interface. Notice that this is a proper generalisation of the concept of a simple algebra interface (an algebra signature — cf. Section 2.1). A simple algebra may be viewed as a parameterless parametric algebra, and its interface (the signature) as the corresponding parameteric algebra interface with the empty parameter list omitted.

Parametric algebras may be defined using other (possibly parametric) algebras defined in the environment. Just as in Section 2.1, we require then a complete list of algebras used to be given explicitly in the algebra definition.

Let us point out the rather obvious fact that parametric algebras are not algebras, they are functions yielding algebras when applied to actual parameters. Consequently, it makes no sense to refer to "parametric-algebra components". For example, nothing like $BF.niebieskie$ or $BF.zmien$ is well-formed — the function BF as such has no component; it is the result of full instantiation of BF by giving all its required parameters that is a well-defined algebra, and hence has components listed in the result signature. For example, $BF(X \mapsto A).niebieskie$ and $BF(X \mapsto A').cz\_zero$ both make sense and are well-formed.

Since we now have a concept of an interface for parametric algebras, it is straightforward to generalise the notion of a parametric algebra so that parameters that in turn are parametric are allowed. In other words, we introduce higher-order parametrisation, believing that this may be useful in practice (see e.g. [SST 92] for closely related examples and some discussion). Since each parameteric algebra interface contains as its part the interfaces of all its parameters, higher-order parametric algebras form a proper hierarchy, and no form of self-application is allowed.

We do not allow any form of recursive algebra definitions either, so that dependences between algebras defined are hierarchical. We believe that this is methodologically well-justified. Any apparent need for mutual recursion in algebra definitions (X uses an operation $Y.f$ of Y, and Y uses an operation $X.g$ of X) intuitively shows some inadequacy in the design of the overall structure of the definition (if some definitions in X and Y are inseparable, they should not be split between different modules). A direct recursion (an algebra defined in terms of itself, whatever formally this would mean) seems unnecessary, as we believe that it can always be sensibly reduced to a recursion within the algebra (in the definitions of its carriers and operations).

No pushout parameterisation

One standard technique of defining parameterisation in the context of monomorphic algebraic specifications, where an algebraic specification is viewed as a definition of a unique (up to isomorphism) algebra, is based on the concept of a pushout of signatures, algebraic specifications, and then the amalgamated union of algebras. This approach has been perhaps first introduced in the specification language CLEAR [BG 80] and then intensively studied in the area, cf. [EM 85].

One characteristic feature is that in the pushout approach the entire parameter is included in the result of the instantiation. This is in contrast with the approach we pursue here, where if some part of the parameter is to be included in the result, it must be placed there explicitly.

The pushout approach seems fully justified in a formalism used to gradually construct a single specification by adding more and more pieces. The idea would be to incorporate in this specification all potentially useful components, as otherwise they may be lost. We find this quite unnecessary in specification languages shaped after the usual declarative languages, such as the formalism we propose, where a specification once introduced into the environment stays there and is always available. Moreover, it is always possible to access each particular component of all the algebras stored in the environment. Thus, after instantiating a parameterised specification there is no danger that some components of the actual parameter are not available anymore. If needed, they can always be grabbed from the environment and used subject only to the constraints of the "type system". This would not be flexible enough under the type system for algebra components as suggested in this section, where outside an algebra definition only "abstract" sort information is available about the operations of the algebra. The "type-visibility" mechanisms of the current proposal are designed sufficiently permissive with this very purpose in mind — cf. Section 4.

3 MetaSoft core definitional language

In this section we recall the most important aspects of the core definitional language of MetaSoft, as presented in [BBP 90a]. Of course, we cannot repeat here the entire design — the interested reader should refer directly to [BBP 90a]. Only the ideas most relevant to and used in the current proposal of the module layer of the language are sketched.

The core definitional language of MetaSoft is a simple declarative language to build an environment of sets of data and objects in these sets. The former are represented as *symbolic types.*

The starting point for the definition and interpretation of the notion of a symbolic type in MetaSoft is a suitably rich universe $\mathcal{D}$ of domains (sets of data) closed under the standard set-theoretic operators used to construct domains: Cartesian product $\times$, disjoint union $\uplus$, partial function space $\stackrel{\sim}{\rightarrow}$, and mapping space $\stackrel{\sim}{\underset{m}{\rightarrow}}$. It is also assumed that $\mathcal{D}$ contains the empty set $\emptyset$ and a one-element set $\{*\}$. Thus, we have an *algebra of domains*, in which we can interpret in the obvious way all the closed terms built over the type operators mentioned above. If convenient, we may consider additional constants denoting some specific standard domains, like integer or bool (even though they are definable using the operations already mentioned and recursion).

It turns out technically convenient to represent such terms as trees with nodes labelled by the domain operators (with the number of ordered successors of each node equal to the arity of the operator labelling this node). The next step is to generalise such trees and deal with arbitrary pointed (the point indicating the "root") directed graphs with nodes labelled in the similar manner. Now, such graphs may be unambiguously interpreted in the algebra of domains, with cycles used to indicate the least (w.r.t. the inclusion of domains) fixed points, under the additional assumption that the cycles do not contain nodes labelled by the partial function space operator (cf. [BlT 83] for the "naive domain theory" which underlies and justifies this definition).

Moreover, any such (finite) graph may be expanded by unfolding its cycles to a (possibly infinite) tree, which gives the structure of the domain defined by the original graph. We want to identify graphs expanding to isomorphic trees. It turnes out that the resulting equivalence relation is "easily" decidable [BBP 90b]. Any two graphs equivalent in this sense determine the same domain in the algebra $\mathcal{D}$. Notice, however, that the opposite implication does not hold in general: there are non-equivalent graphs that determine identical domains. This is hardly surprising, as in fact the intention is to identify the graphs with the same structure, not just with the same set-theoretic interpretetion.

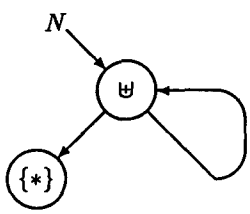
The symbolic types, $st : SymType$, of the definitional langauge of MetaSoft, are defined as equivalence classes of such graphs. Any symbolic type $st : SymType$ determines a domain (a set) $[\![st]\!]$ of *objects of this type.*

In the core definitional language symbolic types are constructed using the available type operators in the standard manner. They may be bound to identifiers, thus gradually building

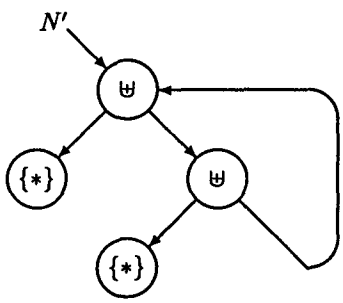
up an environment of types. Recursive definitions may be used to introduce sets of mutually recursive types (graphs with cycles). This is subject to the restriction that no recursion loop may go through the function-space operator.

For example, the following two recursive type definitions introduce symbolic types represented by the graphs given below.

rec_type
$N = \{*\} \uplus N$
end



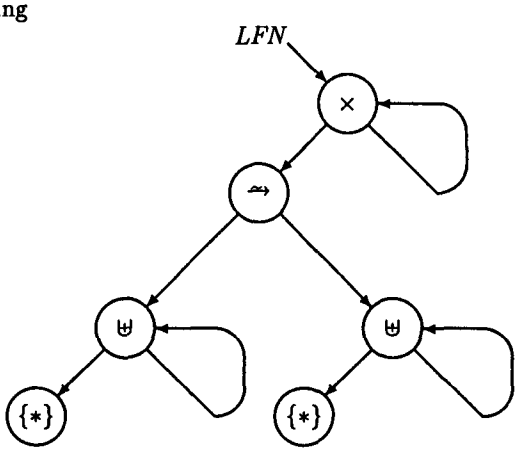
rec_type
$N' = \{*\} \uplus (\{*\} \uplus N')$
end



It is quite straightforward to see that the two graphs are equivalent and so the two symbolic types are equal. In fact, they define (a set which may be identifdied with) the set of natural numbers. Then, the type of finite lists of funtions from N to N may be defined as follows:

rec_type
$LFN = (N \rightharpoonup N) \times LFN$
end

yielding



Notice that the above recursive type does involve the function-space operator, but only in a non-essential way: no recursion goes through this operator.

The definitional language of MetaSoft is *strongly typed.* That is, the objects constructed in the language are always built and then stored in the environment together with their symbolic types. Hence, the environment being constructed consists in fact of two parts: a "dynamic" environment storing the objects bound to identifiers, and a "static" environment storing symbolic types of these objects. The overall invariant of the entire langauge definition is that any object built is an element in the domain determined by the symbolic type of this object. Somewhat informally in

this paper, we will write $x : st$ to denote the fact that the object x is of type st. A consequence of this is that every phrase of the definitional language is first elaborated "statically", to check its well-formedness and compute the types of the objects being defined, and only then evaluated "dynamically" to compute the actual meanings of the entire phrase and of the objects.

The structure of a symbolic type determines the operations that can be used to construct objects of this type and the operations which can be applied to objects of this type. For example, if a type is given as a product of two types, $st = st' \times st''$, then the pairing operation $\langle \_, \_ \rangle$ may be used to construct objects of this type: for $x' : st'$ and $x'' : st''$, $\langle x', x'' \rangle : st$. Similarly, projection operations may be applied to any object of type st, yielding the first and second, respectively, element of the pair this object is. The operations like pairing, used to build the objects of a given type are often referred to as *constructors* for this type, the operations like projections, which may be applied to objects of a given type to build intuitively simpler objects are referred to as *desctructors* for this type. The usual object constructors and desctructors corresponding to the particular type operators are introduced in MetaSoft.

Perhaps function-space types require some further remarks. The only desctructor for function types is the operation of application. Given an object of a function type, say $f : st' \rightharpoonup st''$, the binary operation of application may be applied to f and an object $x : st'$ to yield an object $f.x : st''$. This says that the function $f : st' \rightharpoonup st''$ may be "applied" (via the binary application operation) to any object $x : st'$. λ-abstraction is used to construct objects of function-space types. To evaluate a function expression of the form $\lambda x{:}st'.exp$ to obtain an object of a function type $st' \rightharpoonup st''$, we first "statically" elaborate the body expression exp in the "static" environment enriched by indicating x as an object of type st', check whether the computed symbolic type of exp is st'', and only then define a (partial) function that maps any element $a \in [\![st']\!]$ to the "dynamically" computed value of exp in the environment enriched by mapping x to the value a (in the "dynamic" environment) and to the symbolic type st' (in the "static" environment). Notice that the static elaboration and the dynamic evaluation of the body expression exp cannot be done at the same time here. At the time of evaluation of the λ-expression, when the static analysis must be performed, no value $a \in [\![st']\!]$ can be singled out. The only alternative would be to evaluate this expression for all potential values $a \in [\![st']\!]$ — but the very point of the static analysis is to avoid such a necessity.

Objects of some types may be defined recursively. The interpretation of recursive definitions is given by the least fixed points in the appropriate domains. This is meaningful due to the fact that all the domains definable by symbolic types come equipped with an ordering (non-trivial orderings are introduced for function-space domains and then lifted up through the domain operators in the standard way). It turns out that all the domains form complete partial orders, and all the operations definable in the language are continuous w.r.t. these orderings. Although not all of the domains contain least elements, it is easy to statically identify symbolic types defining domains that do so. Objects of these types may be defined recursively, with the usual least-fixed-point interpretation.

4 Modules for MetaSoft

In this section we present the basic concepts underlying the formal definition of the MetaSoft definitional language with modules in [Tar 92]. The overall design of the language results, in a sense, from recasting the ideas sketched in Section 2 in the specific framework of definitional mechanisms for domains represented by symbolic types and objects in these domains as presented in [BBP 90a] and briefly recalled in Section 3.

Let us list the basic assumptions of the proposal. They will be gradually discussed and explained in more detail throughout the rest of this section.

- A usual declarative language to build an environment of MetaSoft *algebras* is designed.
- A MetaSoft *algebra* is a representation of an algebra in the usual sense. It consists of a

set of named symbolic types, which determine the carriers of the algebra, and of a set of named objects, which correspond to the operations of the algebra. MetaSoft algebras are always constructed with their *signatures.*

- A MetaSoft *signature* contains a set of names of the carriers (or symbolic types) and a set of names of the objects of MetaSoft algebras over this signature. As in algebraic signatures, the names of objects are equipped with the types of the objects. Unlike algebraic signatures, MetaSoft signatures may give some extra information about the required structure and mutual relationships of the symbolic types representing algebra carriers.

- MetaSoft algebra definitions may be parametric, which yields the concept of a MetaSoft *module*, a central concept of the current proposal. MetaSoft modules come always equipped with their *interfaces.*

- The MetaSoft module parameterisation may be of higher-order, that is, there are modules parameterised by modules etc.

- A MetaSoft *module interface* gives the list of the names of algebra and module parameters with the required interfaces as well as the result algebra interface. The interfaces in general contain some extra information about the symbolic types involved, stating the required structure and mutual relationships between the types in the parameters, as well as providing some information about the structure and relationship to the given types of the types in the result interface. In particular, the result algebra interface lists the set of types newly defined by the module.

- A MetaSoft module may be *applied to* (or *instantiated with*) a list of actual algebra and module parameters with interfaces fitting the required parameter interfaces. The result is a MetaSoft algebra with the signature calculated out of the result algebra interface given in the module interface. This is done in a "generative" fashion, that is, the types newly introduced by the module are treated as new each time the module is instantiated.

- MetaSoft modules may use other MetaSoft modules and algebras already defined in the environment. Components of MetaSoft algebras listed in their signatures may be accessed by naming the algebra and its particular component (cf. the "dot notation" of Section 2). Only the type information given in the algebra signature is available about algebra components accessed in such a way.

4.1 Abstract types

As discussed in Section 3, the structure of the type of an object determines the operations that may be applied to this object. In the core definitional language there is no way to hide this structure. The type of an object is always fully visible. On the other hand, as suggested by the discussion in Section 2, it seems important to use module interfaces to limit the flow of information about the types of objects. The goal is to achieve abstraction by allowing only *some* operations that in principle may be applied to an object to be actually used outside the encapsulated module in which the object is defined. In this way the particular definitions of the types and objects within a module may be changed without affecting the way they are used outside, provided that the new definitions still fit the module interface.

To provide appropriate technical tools for this, we propose to extend the concept of a symbolic type as recalled in Section 3 so that no structural information about some types (or their subparts) needs to be given. Consider a new set of *token types*, $tt : TType$, which are just arbitrary new tokens (in the usual sense of VDM definitions — unknown entities, distinct from all the others around). Their primary role is to be used as *abstract types* with an entirely hidden internal structure. Now, extend the set of symbolic types by allowing the token types as constant

type operators to label the nodes of the graphs used in the construction of the symbolic types. Notice that since the new token type operators are constants, no recursion may go through them.

Since in principle we know nothing about the interpretation of the token types, the symbolic types now do not necessarily denote domains in the domain algebra $\mathcal{D}$ on which the semantics of the langauge is built (cf. Sec. 3). We will refer to the symbolic types which do not actually contain any token types as *real* symbolic types. Of course, each real symbolic type determines a domain in $\mathcal{D}$ as before.

A type given as a token type provides no information about its structure. Hence no built-in constructors may be used to define objects of such a type. Similarly, no built-in destructors may be applied to objects of such type either. This seems to make the abstract token types quite useless — apparently we could neither construct nor use objects of such types.

The trick is to give the abstract token types a real identity throughout the entire definition in the MetaSoft langauge. That is, we will assume that whenever the same abstract token type is used in a certain definition, it hides an unknown but always the same type. For example, given an object $a : tt$ of a token type tt and a function $f : tt \rightharpoonup \mathsf{bool}$, we may apply f to a (and obtain an object $f.a : \mathsf{bool}$, if f is defined on a). Similarly, a function $g : \mathsf{bool} \rightharpoonup tt$ (or even simpler, a constant $c : tt$) may be used to construct objects of type tt.

Notice that any function $sbst : TType \rightharpoonup SymType$ (we call such functions *substitutions*) extends in the obvious "structural" way to a function $sbst : SymType \rightharpoonup SymType$. The extended function is undefined only on the types containing token types not in the domain of the original substitution. Moreover, substitutions will be implicitly extended to any structure containing symbolic types (e.g., to environments storing symbolic types).

Abstract token types are introduced when a number of type and object definitions are encapsulated to form a module. The structure of real symbolic types may then be partially hidden, by replacing some of them by abstract token types. Throughout a MetaSoft definition a substitution of real symbolic types for the abstract token types that hide them is implicitly built.

4.2 Objects and types

In Section 3 we have used the example of λ-expression defining a function to argue for the need to separate the "static" elaboration from the "dynamic" evaluation of any phrase of the definitional language. This is needed even more in the context of the current module proposal. As we will see in Section 4.4, MetaSoft module definitions are very close to "λ-expressions-in-the-large", and hence force the need for a separate "static" elaboration of all the phrases of the language (since all of them may occur in a module body).

In the core language, for any object, apart from its actual value (the "dynamic" information) its type (the "static" information) is stored as well. The object environment built in the core language consists of two parts: "static", yielding the types of objects, and "dynamic", yielding actual values. These two parts are required to be *consistent* with one another in the sense that the values stored must always be in the domains determined by the symbolic types bound to the corresponding object identifiers.

In the current proposal we extend this idea to all the entities of the language[2]. Perhaps somewhat surprisingly, in particular this includes types defined in MetaSoft. The type environment consists again of a "dynamic" part, storing the real symbolic types denoted by type identifiers, and of a "static" part, storing the symbolic types (possibly containing abstract token types) that represent the currently available information about the actual structure of the type and about its relationship with other types. Of course, these two parts of the type environment must be *consistent* with one another in the sense that the current substitution of real for the abstract token types used maps the "static" to the "dynamic" type environment (that is, the two type environments have the same domains and for every identifier in the domain, the static

[2]With one exception: module interfaces are purely "static" entities, hence no separate "dynamic" information about them is given.

type bound to the identifier in the static environment is mapped to the real type bound to the identifier in the dynamic environment).

For example, suppose that we define

$$\begin{array}{l}\textsf{type } \mathit{luty} = \textsf{integer}\\ \textsf{function } \mathit{dodaj} : \mathit{luty} \rightharpoonup \mathit{luty}\\ \qquad\qquad \mathit{dodaj}.n = n + 5\end{array}$$

and then these definitions are encapsulated (see Section 4.3 below) so that the real interpretation of the type *luty* is hidden by a new abstract token type $\mathit{att} : \mathit{TType}$ with an implicit substitution $\mathit{att} \mapsto \textsf{integer}$. Consequently, even though in the dynamic environments we have $\mathit{luty} \mapsto \textsf{integer}$ and $\mathit{dodaj} \mapsto \lambda n{:}\textsf{integer}.n + 5$, in the static environments $\mathit{luty} \mapsto \mathit{att}$ and $\mathit{dodaj} \mapsto \mathit{att} \rightharpoonup \mathit{att}$. Then, in this context we can define

$$\begin{array}{l}\textsf{type } \mathit{maj} = \mathit{luty} \times \mathit{luty}\\ \textsf{function } \mathit{raz\_dodaj} : \mathit{maj} \rightharpoonup \mathit{maj}\\ \qquad\qquad \mathit{raz\_dodaj}.\langle x, y\rangle = \langle \mathit{dodaj}.x, y\rangle\end{array}$$

Now, the new type *maj* "really" stands for integer × integer, but in the static environment it denotes $\mathit{att} \times \mathit{att}$. The function $\mathit{raz\_dodaj} : (\mathit{att} \times \mathit{att}) \rightharpoonup (\mathit{att} \times \mathit{att})$ is well-defined — the "static" structure of *maj* and the "static" type of *dodaj* provide information sufficient to justify that its definition is well-formed. This is in contrast with the following ill-formed definition:

$$\begin{array}{l}\textsf{function } \mathit{dwa\_dodaj} : \mathit{maj} \rightharpoonup \mathit{maj}\\ \qquad\qquad \mathit{dwa\_dodaj}.\langle x, y\rangle = \langle \mathit{dodaj}.x, y + 5\rangle\end{array}$$

The fact that *luty* stands for (or rather, that *att* hides) the real type integer is not statically visible and cannot be used to justify the well-formedness of the subexpression $y + 5$.

4.3 Algebras and their signatures

A MetaSoft *algebra* consists of two environments: a dynamic type environment, storing the real symbolic types that correspond to carriers of the usual algebras, and a dynamic object environment, storing the actual values corresponding to operations of the usual algebras.

A MetaSoft *signature* consists of two environments: a static type environment, storing the symbolic types (possibly containing abstract token types) that provide information about the "carriers", and a static object environment storing the symbolic types (possibly containing abstract token types) of the "operations".

A MetaSoft *algebra is of a* MetaSoft *signature* if the dynamic type environment of the algebra is consistent with the static type environment of the signature and the dynamic object environment of the algebra is consistent with the static object environment of the signature w.r.t. the current substitution of real types for the abstract token types used.

For example (recall the definition of algebra A in Section 2.1) consider an algebra A given as a pair of dynamic environments

$$\begin{array}{l}A = \langle [\mathit{zielone} \mapsto \textsf{integer} \times \{0\}, \mathit{czerwone} \mapsto \textsf{integer} \times \{1\}],\\ \qquad\qquad\qquad [\mathit{z\_zero} \mapsto \langle 0, 0\rangle, \mathit{zmien} \mapsto \lambda\langle n, 0\rangle{:}\textsf{integer} \times \{0\}.\langle n, 1\rangle]\rangle\end{array}$$

and a signature *ASig* given as a pair of static environments

$$\mathit{ASig} = \langle [\mathit{zielone} \mapsto \mathit{att}_1, \mathit{czerwone} \mapsto \mathit{att}_2], [\mathit{z\_zero} \mapsto \mathit{att}_1, \mathit{zmien} \mapsto \mathit{att}_1 \rightharpoonup \mathit{att}_2]\rangle$$

Then the algebra A clearly is of the signature *ASig* (under the obvious substitution of real types for the two abstract token types involved).

We will also need an auxiliary technical concept of a MetaSoft *algebra interface*. Algebra interfaces are much like algebra signatures, except that some of token types occuring in them

may be indicated as "flexible", not fixed, and in a sense local to the interface rather than common throughout the entire definition. These indicated token types play the role of "place-holders" for symbolic types to replace them later, and are considered to be *bound*. We consider algebra interfaces up to the obvious α-conversion of the bound token types.

Here is an example of an interface definition. Notice that we do not give explicitly the set of bound token types. We simply list the type identifiers the interface is to contain, and if no further information about them is given, they are assigned new token types, subsequently treated as bound in the interface.

```
interface AInt =
   types  zielone
          czerwone
 objects  z_zero : zielone
          zmien : zielone ⇀ czerwone
end
```

This defines (the bound token types of an algebra interface are placed in parentheses in front of the signature component, much as in [HMT 90]):

$$AInt = (tt_1, tt_2)\langle [zielone \mapsto tt_1, czerwone \mapsto tt_2], [z\_zero \mapsto tt_1, zmien \mapsto tt_1 \rightharpoonup tt_2]\rangle$$

In the current proposal there will be no way to directly define algebra signatures. They will be always generated out of a corresponding algebra interface. As a part of this process, bound token types will be replaced by newly generated abstract token types, now considered as fixed and visible throughout the rest of the definition.

For example:

```
algebra A : AInt =
          zielone = integer × {0}
          czerwone = integer × {1}
          z_zero = ⟨0, 0⟩
          zmien.⟨n, 0⟩ = ⟨n, 1⟩
end
```

builds the algebra A with the signature $ASig$ as defined above. The signature $ASig$ is generated out of the algebra interface $AInt$ by substituting new abstract token types att_1 and att_2 for the bound token types tt_1 and tt_2, respectively.

We will say that *a* MetaSoft *signature fits an algebra interface* if there exists a substitution modifying only the bound token types of the interface that maps the static type environment of the interface to the static type environment of the signature, and the static object environment of the interface to the static object environment of the signature. MetaSoft algebras will be always evaluated and stored together with their signatures. We will say that an algebra fits an interface if the algebra signature does so. For example, the algebra A and its signature $ASig$ fit the interface $AInt$ defined above.

Here are some further examples, well-formed in the context of the definitions listed above.

```
interface BInt =
   types  niebieskie = A.zielone
          czarne = A.czerwone
          nowy
 objects  cz_zero : czarne
          zmien : niebieskie ⇀ czarne
end
```

This defines an interface

$$BInt = (tt_3)\langle[niebieskie \mapsto att_1, czarne \mapsto att_2, nowy \mapsto tt_3],\\ [cz\_zero \mapsto att_1, zmien \mapsto att_1 \rightharpoonup att_2]\rangle$$

with only one bound token type, named by *nowy*. The other two types (*niebieskie* and *czarne*) are mapped to the abstract types denoted by *A.zielone* and *A.czerwone*, respectively.

```
algebra B : BInt =
  using A
        niebieskie = A.zielone
        czarne = A.czerwone
        nowy = bool
        cz_zero = (A.zmien).(A.z_zero)
        zmien.x = (A.zmien).x
end
```

This defines the MetaSoft algebra corresponding to the algebra B of Section 2.1 with the signature where *niebieskie* and *czarne* are mapped to the same abstract types as *zielone* and *czerwone*, respectively, in *ASig* (and *nowy* is bound to a new abstract type). Therefore, we can now combine A and B, as unsuccessfullly attempted in Section 2.1:

```
interface ABInt =
   types  kolor1, kolor2, kolor3
 objects  zero1 : kolor1
          zero2 : kolor2
          zmien : kolor1 ⇀ kolor2
end
```

```
algebra AB : ABInt =
  using A, B
        kolor1 = A.zielone
        kolor2 = A.czerwone
        kolor3 = B.nowy
        zero1 = A.z_zero
        zero2 = B.cz_zero
        zmien = A.zmien
end
```

We have chosen to define the interface *ABInt* so that the information that the algebra *AB* shares the types with A and B is not propagated.

Since the interface *AInt* for the algebra A does not provide the information that the types *zielone* and *czerwone* are in fact integer × {0} and integer × {1}, respectively, the following two definitions are not well-formed, similarly as in Section 2.1:

```
algebra B1 : BInt =
  using A
        niebieskie = integer × {0}
        czarne = integer × {1}
        nowy = bool
        cz_zero = (A.zmien).(A.z_zero)
        zmien.x = (A.zmien).x
end
```

```
algebra B2 : BInt =
  using A
        niebieskie = A.zielone
        czarne = A.czerwone
        nowy = bool
        cz_zero = ⟨0, 1⟩
        zmien.⟨n, 0⟩ = ⟨n, 1⟩
end
```

We have used above the most straightforward way to define an algebra by defining one by one all its components, as suggested in Section 2.1. For technical reasons, this is not formally allowed in the current proposal, where he only way to construct algebras is by module instantiation (cf. Section 4.4). The above examples stretch the actual formalism somewhat. However, the possibility of directly defining an algebra in such a way may be easily introduced as a notational convention to abbreviate a definition of a parameterless module and its immediate instantiation (with the empty list of parameters).

4.4 Modules and their interfaces

A MetaSoft *module interface* consists of the following items:

- *a result algebra interface*;
  Any module with this interface, when instantiated with admissible parameters, produces an algebra fitting the result algebra interface.

- *a parameter interface*;
  The parameters of a module may be either algebras or other modules. Hence, this is split into two parts:

  - *an algebra-parameter interface*, given as an environment of signatures, and
  - *a module-parameter interface*, given as an environment of module interfaces.

  The distinction between algebra parameters and module parameters is necessary. Under so-called *generative semantics* of module application (see the description of module instantiation below and some further discussion in Section 5) parameterless modules are quite different from algebras. Two occurrences of an algebra stand for the same algebra, with the same types as carriers; two occurrences of a parameterless module instantiation (no parameters required) may denote different algebras — the new types will be different.

- an indicated set of *flexible token types*;
  These are "flexible" token types of the algebra-parameter interface (all factored out, so that algebra signatures, rather than algebra interfaces form the algebra-parameter interface). The flexible token types will be substituted for in the process of module instantiation in a way determined by the signatures of the actual algebra parameters. The tokens in this set may occur in the interfaces of a number of parameters, thus indicating the requirements that some types are to be shared between actual parameters, and in the result algebra interface, thus indicating the type propagation in any module with this interface.

  The flexible tokens types are regarded as local for the module interface, their occurences are bound, and the module interfaces are considered up to the obvious α-conversion.

  The role of bound token types hidden in the result algebra interface is quite different from the bound token types of the module interface: the latter are used to express type sharing constraints, the former are "place-holders" for new abstract types to be generated.

Consider the following definition of a module interface (see Section 4.3 for *AInt*).

```
interface BFInt =
  params  algs  X : AInt
    result    types  niebieskie = X.zielone
                     czarne = X.czerwone
                     nowy
              objects cz_zero : czarne
                      zmien : niebieskie ↛ czarne
end
```

This introduces a module interface *BFInt* with the result algebra interface listing types *niebieskie*, *czarne* and *nowy*, and objects *cz_zero* and *zmien* of the indicated types. Among the types in the result interface, the first two are propagated from the parameter X. No further information is given about *nowy*, hence it is understood to be a new type generated by the modules fitting this interface, and so is bound to a token type local to the result algebra interface. There are no module parameters, and only one algebra parameter X. The bound token types of its required interface *AInt* are factored out as bound token types of the module interface. Formally, *BFInt* is the following complex tuple (we leave this as an exercise for the reader to carefully interpret all its components):

$$BFInt = (tt_1, tt_2)\langle\langle[X \mapsto XSig], []\rangle, R_{BF}Int\rangle$$

where

$$XSig = \langle[zielone \mapsto tt_1, czerwone \mapsto tt_2], [z\_zero \mapsto tt_1, zmien \mapsto tt_1 \rightharpoonup tt_2]\rangle$$

$$R_{BF}Int = (tt_3)\langle[niebieskie \mapsto tt_1, czarne \mapsto tt_2, nowy \mapsto tt_3], [cz\_zero \mapsto tt_1, zmien \mapsto tt_1 \rightharpoonup tt_2]\rangle$$

Another example is an interface of a two-argument module.

```
interface ABFInt =
  params  algs  X : AInt
                Y :  interface    types  niebieskie = X.zielone
                                         czarne = X.czerwone
                                         nowy
                                  objects  cz_zero : czarne
                                           zmien : niebieskie ⇀ czarne
                      end
    result      types  kolor1, kolor2, kolor3
              objects  zero1 : kolor1
                       zero2 : kolor2
                       zmien : kolor1 ⇀ kolor2
end
```

Here in turn no types are declared to be propagated from the parameters, but the interface indicated for the algebra parameter Y imposes some sharing requirements on the types in the corresponding parameter. The resulting module interface formally is the following.

$$ABFInt = (tt_1, tt_2, tt_3)\langle\langle[X \mapsto XSig, Y \mapsto YSig], []\rangle, R_{ABF}Int\rangle$$

where

$$XSig = \langle[zielone \mapsto tt_1, czerwone \mapsto tt_2], [z\_zero \mapsto tt_1, zmien \mapsto tt_1 \rightharpoonup tt_2]\rangle$$

$$YSig = \langle[niebieskie \mapsto tt_1, czarne \mapsto tt_2, nowy \mapsto tt_3], [cz\_zero \mapsto tt_1, zmien \mapsto tt_1 \rightharpoonup tt_2]\rangle$$

$$R_{ABF}Int = (tt_4, tt_5, tt_6)\langle[kolor1 \mapsto tt_4, kolor2 \mapsto tt_5, kolor3 \mapsto tt_6], [zero1 \mapsto tt_4, zero2 \mapsto tt_5, zmien \mapsto tt_4 \rightharpoonup tt_5]\rangle$$

MetaSoft modules are functions which take actual parameters explicitly bound to formal parameter identifiers as arguments and yield algebras (or generate an error). As with other entities of the language, they are always evaluated and stored together with their interfaces. A module fitting an interface takes actual parameters fitting the parameter interfaces and yields an algebra with a signature fitting the result algebra interface (in the non-error case).

The definition of "a module fitting an interface" implicit in the previous paragraph may seem unpleasantly circular (and hence formally incorrect). However, due to the strict use of parameter interfaces, the MetaSoft modules form a hierarchy where no self-application can occur. Therefore, the above definition is a simple inductive definition over the hierarchy of modules implicitly given by their interfaces.

Module definitions

A module definition consists of a module interface, a list of algebras and modules from the environment to be used in the definition, and a *module body.*

The module body is simply a list of type and object definitions in a slightly extended core language. The extension allows the user to select components of algebras either available in

the environment or built by module instantiation. This is quite straightforward and does not require any further discussion. Let us just point out that recursive type or object definitions involving in an essential way module instantiations (that is, where recursion actually "goes through" a module instantiation) may yield somewhat unexpected results. As a consequence of the generative semantics of module instantiation mixed with the iterative way to generate the least-fixed-point solutions, some of the objects and types defined by mutual recursion may share fewer abstract types than expected.

To statically elaborate such a module definition, that is, to check whether the definition is well-formed and the module being defined indeed fits the interface, we first have to check whether the current environment actually stores the algebras and modules listed as "to be used" by the module being defined. Then, a "static" algebra and module environment is formed by cutting down the current environment (actually, its static part) to the "to be used" algebra and module identifiers, and adding the formal parameter identifiers with signatures and interfaces indicated in the parameter interface. Some renaming of the flexible token types of the module interface may be necessary to avoid unintended clashes with the abstract types already in use. The next step is to statically elaborate the module body in this environment to obtain the static type and object environments generated by the type and object definitions. Finally, we have to check whether these environments fit the corresponding parts of the result algebra interface.

If all this is successful, then the module defined is a function computed as follows.

Given actual parameters for the module, explicitly bound to the formal parameter identifiers, we first check whether the actual algebra-parameter signatures fit the algebra-parameter interface w.r.t. a substitution for the flexible token types of the module interface. This substitution is then used to replace the flexible token types in the module-parameter interface and in the result algebra interface. The next step is to check that the actual module-parameter interfaces coincide with the formal module-parameter interfaces[3]. If this is successful then an environment is constructed by selecting the "to be used" components of the global environment (they must be there because the static elaboration has been successful) and adding the actual parameters bound to the formal parameter identifiers. The module body is evaluated in this environment (no static error may occur now!). The result consists of a type environment and of an object environment, which fit the result algebra interface.

The result of the module instantiation is the algebra obtained by cutting out the appropriate parts of the dynamic type and object environments computed by the body evaluation. The signature of this result algebra is obtained from the result algebra interface by replacing its local "place-holder" token types by some newly generated ones, which do not clash with any abstract types already in use. These new abstract types are generated anew each time the module is instantiated, even if the same actual parameter list has already been used in some previous instantiation. Notice that the result algebra interface, not the static environments generated by the body elaboration, is used here, so that the signature of the result of module instantiation may be calculated entirely on the basis of the module interface and of the actual parameter signatures and interfaces. Thus, the static elaboration of a module instantiation may be performed without referring to the module body at all, as expected and required.

Here are simple examples of two modules fitting the interfaces given above:

[3]This requirement may be somewhat relaxed by allowing the actual parameter interfaces to be "more general" than the formal parameter ones. The idea is quite simple: the actual parameter interface must ensure that the actual parameter may be used as a module with the interface given by the formal parameter interface. For example, we can allow the actual parameter module to require fewer parameters or less sharing than indicated by the corresponding formal parameter interface, or looking at the potential results, the actual parameter module may ensure more components or more sharing in its results than required by the corresponding formal parameter interface.

```
module BF : BFInt =
        niebieskie = X.zielone
        czarne = X.czerwone
        nowy = bool
        cz_zero = (X.zmien).(X.z_zero)
        zmien.x = (X.zmien).x
end
```

```
module ABF : ABFInt =
        kolor1 = X.zielone
        kolor2 = X.czerwone
        kolor3 = Y.nowy
        zero1 = X.z_zero
        zero2 = Y.cz_zero
        zmien = X.zmien
end
```

It is easy to check that indeed the bodies of the two modules are well-formed in the environment formed by binding the algebra-parameter signatures to the corresponding formal parameter identifiers, and the results of their elaboration fit the corresponding result algebra interfaces.

The modules may now be instantiated (we use the algebra A introduced in Section 4.3).

$$\textsf{algebra } B = BF(X \mapsto A)$$

It is again easy to check that the actual parameter indeed fits the parameter signature of the module interface under a uniquely determined substitution of types for the flexible token types ($[tt_1 \mapsto att_1, tt_2 \mapsto att_2]$). As described above, this substitution is then propagated to the result algebra interfaces, so that the signature of the algebra B defined above is the following:

$$BSig = \langle [niebieskie \mapsto att_1, czarne \mapsto att_2, nowy \mapsto att_3], \\ [cz\_zero \mapsto att_1, zmien \mapsto att_1 \stackrel{\sim}{\to} att_2] \rangle$$

where att_3 is a new abstract token type generated during the module instantiation. Further instantiations of BF, even with the same parameter, lead to new MetaSoft algebras:

$$\textsf{algebra } BB = BF(X \mapsto A)$$

BB is an algebra over a new signature:

$$BBSig = \langle [niebieskie \mapsto att_1, czarne \mapsto att_2, nowy \mapsto att'_3], \\ [cz\_zero \mapsto att_1, zmien \mapsto att_1 \stackrel{\sim}{\to} att_2] \rangle$$

where att'_3 is a new, distinct abstract token type generated during the module instantiation.

Either of the above algebras fitting the interface $BInt$ may be used as a parameter for the module ABF.

$$\textsf{algebra } AB = ABF(X \mapsto A, Y \mapsto B) \qquad \textsf{algebra } ABB = ABF(X \mapsto A, Y \mapsto BB)$$

Since in the interface of ABF no type propagation has been declared, the signatures of the two algebras thus defined use newly generated, distinct abstract types for $kolor1$, $kolor2$ and $kolor3$.

5 Further extensions and final remarks

Let me point out two major doubts about the presented proposal.

The first doubt concerns the major design decision on the module semantics in the language proposed. Namely, we have adopted here the so-called generative semantics of module instantiation. This means that each application of a module to an argument generates new abstract types, different from the ones generated before, even by the same module for identical arguments (of course, this may be overridden by an explicit sharing declaration in the result algebra interface). The are at least two other options:

- Define modules as functions which yield the same results, including abstract types, when given the same arguments. This would be perhaps conceptually most clear, but not very practical. Namely, this would lead to a type system with full dependent types (dependent on static as well as dynamic objects, including modules as well!). Thus, no reasonable static analysis would be decidable.

- Explore in full the fact that the underlying core type system does not admit dependent types. New types defined in a module may depend only on the static components of the module arguments. Consequently, two module instantiations would yield the same static results provided that their arguments have identical static parts. This should be decidable, and so it seems that such a type discipline may be imposed (although the details are far from clear, and much work would be necessary — cf. [Mog 89] for related theory).

  Unfortunately, this may lead to intuitively doubtfull identification of types which are not intended to be identified. For example, given a module that produces a type of stacks of a bounded depth, where the depth is a part of the module parameter, the user would not be prevented from using the stack operations produced by the module when applied to a parameter setting the depth to 100000, to stacks built using the operations produced by the same module applied to the same parameter but with the depth changed to 10.

The second doubt is on the boderline between the technicalities and design of the language. Namely, following the decisions made in [BBP 90] for the core MetaSoft langauge, we have assumed here that all the objects definable in the language are continuous, and hence any recursive definitions of objects of appropriate types make sense. In our view, there is no reason to adopt such a strong assumption of "continuity of everything" in a *specification* language. For example, there seems to be no reason to prevent the user from testing the identity of objects of any type, including function types (it is not as clear if testing the identity of objects of abstract types should be allowed, though...). Consequently, it may be justified to resign this assumption and work with potentially non-continuous objects and operations. This would require, however, a major change in the semantics of the recursive object definitions.

The proposal presented here should be extended in a number of important directions.

First, one of the original motivations which led us to believe that the "naive domain theory" underlying MetaSoft is sufficient for practical purposes was that it turned out possible to describe standard hierarchies of higher-order objects (such as procedures with procedural parameters in modern programming languages or modules of the current proposal) as long as some type discipline is imposed to prevent any form of self-application (cf. [BIT 83]). However, this requires dealing in the definitional langauge with infinite hierarchies of domains and their unions. A proposal to include such hierarchies of symbolic types in the MetaSoft core language has been recently formulated in [BBP 91]. Although there should be no major problems to incorporate such an extended core language into our module proposal, the details are not entirely clear. The infinite type hierarchies introduce dependent types, albeit in a very limited, statically-checkable form.

Then, one aspect in which our proposal is more restrictive than the modularisation facilities of Standard ML is that we have not allowed MetaSoft algebras with MetaSoft algebras as components. This is just for the simplicity of the first definition — no technical difficulties are envisaged, and in fact we plan to include both algebra and module components in MetaSoft algebras (and module bodies) in the final proposal.

A practical issue which has become apparent even in the extremely simplified examples used in this paper is that writing module interfaces is often a very tedious task. A separate, non-trivial sublanguage for defining interfaces should be carefully chosen as a part of the complete proposal.

We have allowed here no form of axiomatic requirements in module interfaces. It would be useful to follow one of the standard VDM techniques and allow *invariants* to be imposed on types defined in modules (and hence potentially used in module interfaces as well). In the construction of the universe of domains for VDM in [TW 90] we have proposed that type invariants should be imposed on types in a sense *a posteriori*. They would not be taken into account by the type analysis ensuring the well-formedness of the definitions at all, but the appropriate proof obligations would be generated. It seems that in order to meet this proof obligations some "dynamic" properties of the algebras defined would have to be stated and proved, perhaps in the way suggested towards the end of Section 2.1.

Finally, it would be interesting to try to generalise the current proposal to deal with "loose algebras", that is, to use the formalism proposed here to define, store and manipulate property-oriented specifications interpreted as classes of algebras rather than as single algebras.

References

[Ada 80] *The Programming Language Ada: Reference Manual.* LNCS 106, Springer 1980.

[Bau 85] Bauer, F.L. *et al* *The Wide Spectrum Language CIP-L.* LNCS 183, Springer 1985.

[BBP 90a] Bednarczyk, M., Borzyszkowski, A., Pawłowski, W. Towards the semantics of the definitional language of MetaSoft. In: *VDM and Z — Formal Methods in Software Development*, Proc. VDM-Europe Symp. 1990, Kiel, LNCS 428, pp. 477–503, Springer 1990.

[BBP 90b] Bednarczyk, M., Borzyszkowski, A., Pawłowski, W. Recursive definitions revisited. In: *VDM and Z — Formal Methods in Software Development*, Proc. VDM-Europe Symp. 1990, Kiel, LNCS 428, pp. 452–476, Springer 1990.

[BBP 91] Bednarczyk, M., Borzyszkowski, A., Pawłowski, W. Towards the semantics of the definitional language of MetaSoft: Dependent types. Technical report, Institute of Computer Science PAS, Gdańsk, December 1991.

[BjJ 78] Bjørner, D., Jones, C.B. *The Vienna Development Method: The Meta-Language.* Springer LNCS 61, 1978.

[BjJ 82] Bjørner, D., Jones, C.B. *Formal Specification and Software Development.* Prentice Hall 1982.

[Bl 89] Blikle, A. Denotational engineering. *Science of Computer Programming* 12(1989), pp. 207–253.

[BlT 83] Blikle, A., Tarlecki, A. Naive denotational semantics. In: *Information Processing 83*, Proc. IFIP World Congress'83, Paris 1983, R.E.A.Mason, ed., pp. 345–355, North-Holland 1983.

[BlTT 91] Blikle, A., Tarlecki, A., Thorup, M. On conservative extensions of syntax in system development. In: *Images of Programming*, dedicated to the memory of A. P. Ershov, D.Bjørner, V.Kotov, eds., pp. 209–233, North-Holland 1991.

[BG 80] Burstall, R.M., Goguen, J.A. The semantics of CLEAR, a specification language. Proc. *of Advanced Course on Abstract Software Specification*, Copenhagen, LNCS 86, pp. 292–332, Springer 1980.

[DMN 70] Dahl, O.-J., Myrhaug, B., Nygaard, K. *Simula 67 common base language.* Report S-22, Norwegian Computing Center, Oslo, 1970.

[EH 85] Ehrig, H., Mahr, B. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics.* Springer 1985.

[EM 90] Ehrig, H., Mahr, B. *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints.* Springer 1990.

[FJKR 87] Feijs, L.M.G., Jonkers, H.B.M., Koymans, C.P.J., Renardel de Lavalette, G.R. *Formal definition of the design language COLD-K.* METEOR Report t7/PRLE/7, Philips Research Laboratories, April 1987.

[GB 84] Goguen, J.A., Burstall, R.M. Introducing institutions. Proc. *Logics of Programming Workshop*, Carnegie-Mellon, LNCS 164, pp. 221-256, Springer 1984.

[Gor 79] Gordon, M.J.C. *The denotational description of Programming Languages: An Introduction.* Springer 1979.

[GHW 85] Guttag, J.V., Horning, J.J., Wing, J. *Larch in five easy pieces.* Report 5, DEC Systems Research Center, Palo Alto 1985.

[Hoa 69] Hoare, C.A.R. An axiomatic basis for computer programming. *Communications of the ACM* 12(1969), pp. 576–580, 583.

[Jon 86] Jones, C.B. *Systematic Software Development Using VDM.* Prentice Hall 1986.

[LB 88] Lampson, B., Burstall, R. Pebble, a kernel language for modules and abstract data types. *Information and Computation* 76(1988) pp. 278–346.

[Lar 89] Larsen, P.G. The Dynamic Semantics of the BSI/VDM Specification Language. Technical report, Dept. of Computer Science, Technical University of Denmark, Lyngby, October 1989.

[Lis 81] Liskov, B.H. *et al.* *CLU Reference Manual.* LNCS 114, Springer 1981.

[MacQ 86] MacQueen, D.B. Modules for Standard ML. In: Harper, R., MacQueen, D.B. and Milner, R. Standard ML. Report ECS-LFCS-86-2, Univ. of Edinburgh 1986.

[MTH 90] Milner, R., Tofte, M., Harper, R. *The Definition of Standard ML.* MIT Press 1990.

[Mog 89] Moggi, E. A category-theoretic account of program modules. Proc. *Category Theory and Computer Science*, D.H.Pitt, D.E.Rydeheard, P.Dybjer, A.M.Pitts, A.Poigné, eds., LNCS 389, 101–117, Springer 1989.

[MSoft 90] *Project* MetaSoft. Project description, Institute of Computer Science, Polish Academy of Sciences, Warsaw, March 1990.

[NHWG 88] Nielsen, M., Havelund, K., Wagner, K.R., George, Ch. The RAISE language, method and tools. In: *VDM — The Way Ahead*, Proc. VDM-Europe Symp. VDM'88, Dublin, R.Bloomfield, L.Marshall, R.Jones, eds., LNCS 328, pp. 376–405, Springer 1988.

[SST 92] Sannella, D., Sokołowski, S., Tarlecki, A. Toward formal development of programs from algebraic specifications: parameterisation revisited. *Acta Informatica*, to appear; also Bericht 6/90, Informatik, Technische Universität Bremen, April 1990.

[ST 88] Sannella, D., Tarlecki, A. Specifications in an arbitrary institution. *Information and Computation* 76(1988), pp. 165–210.

[ST 89] Sannella, D., Tarlecki, A. Toward formal development of ML programs: foundations and methodology. Report ECS-LFCS-89-71, Laboratory for Foundations of Computer Science, Dept. of Computer Science, Univ. of Edinburgh 1989; extended abstract in Proc. Colloq. *Current Issues in Programming Languages*, 3rd Joint Conf. *Theory and Practice of Software Development* TAPSOFT'89, Barcelona, LNCS 352, pp. 375–389, Springer 1989.

[ST 91a] Sannella, D., Tarlecki, A. Extended ML: past, present and future. Proc. 7th Intl. Workshop *Specification of Abstract Data Types*, Wusterhausen/Dosse, LNCS 534, pp. 297–322, Springer 1991.

[ST 91b] Sannella, D., Tarlecki, A. A kernel specification formalism with higher-order parameterisation. Proc. 7th Intl. Workshop *Specification of Abstract Data Types*, Wusterhausen/Dosse, LNCS 534, pp. 274–296, Springer 1991.

[SW 83] Sannella, D., Wirsing, M. A kernel language for algebraic specification and implementation. Proc. Intl. Conf. *Foundations of Computation Theory*, Borgholm, Sweden, LNCS 158, pp. 413–427, Springer 1983.

[Sc 76] Scott, D. Data types as lattices. *SIAM Jour. on Computing* 5(1976), pp. 522–587.

[Sc 82] Scott, D. Domains for denotational semantics. Proc. ICALP'82, LNCS 140, Springer 1982.

[ScS 71] Scott, D., Strachey, Ch. Towards a mathematical semantics for computer languages. Technical report, Oxford University Computing Laboratory 1971.

[Tar 92] Tarlecki, A. Modules for MetaSoft: a technical definition. Technical report, Institute of Computer Science PAS, Warsaw, in preparation.

[TW 90] Tarlecki, A., Wieth, M. A naive domain universe for VDM. In: *VDM and Z — Formal Methods in Software Development*, Proc. VDM-Europe Symp. 1990, Kiel, LNCS 428, pp. 552–579, Springer 1990.

[Wir 86] Wirsing, M. Structured algebraic specifications: a kernel language. *Theoretical Computer Science* 42(1986) pp. 123–249.

[Wirth 88] Wirth, N. *Programming in Modula-2* (third edition). Springer 1988.