The Simple Category of Modules

Mikołaj Konarski
mikon@mimuw.edu.pl

Institute of Informatics, University of Warsaw,
Banacha 2, 02-097 Warszawa, Poland

Research and Development Center, Comarch SA
Puławska 525, 02-844 Warszawa, Poland

Abstract. Dule is a module system for functional programming languages, modeled using elementary category theory and straightforwardly implemented on an simple categorical abstract machine. The focus of Dule is the ease of maintenance of complete programs at the cost of marginalizing code-reuse mechanisms. Fine-grained modularization without prohibitive programming overhead is made possible by introducing mechanisms inspired by category theory, such as default name-driven sharing and implicit module composition.
In this paper we concentrate on the semantics/implementation of our module system in the abstract categorical machine, via Simple Category of Modules (SCM), where modules are morphisms, signatures are objects, module composition is SCM composition and categorical domain lists module parameters. The construction of SCM from a simple 2-categorical model of the abstract machine provides set-theoretic models for Dule and ensures that its implementation is independent of the core language; in particular, it does not require second-order polymorphism nor dependent types (nor even function types). SCM is cartesian and has enough limits to model type sharing; the constructive proof provides implementation of the related module operations. The semantics of Dule is compositional and constructive, enabling a straightforward, faithful implementation that follows and verifies the semantics.

Key words: module systems, type sharing, category theory, semantics of programming languages, categorical abstract machines

1 Introduction

1.1 Background

Modular programming is necessary due to the growing size, complication and the requirement of modifiability of computer programs. The most useful approach appears to be the integration of rigorous module systems into high level programming languages, as in Standard ML [16] or OCaml [14], with their functor modules. However, strictly modular programming style, with explicitly defined modular dependencies, can be sustained only in small programming projects written in these languages [10]. In large projects, managing the many layers of abstraction, introduced with the module hierarchy, turns out to be harder even than manual tracing of each particular dependency in unstructured code. Abstract and fine-grained modular programming is cumbersome, because the headers (e.g., functor module argument headers) to be written are complicated and module applications often trigger global type sharing errors [20]. When a module has n parameters, $O(n^2)$ potential typing conflicts await at each application, making not only creation of modules, but also their maintenance and reuse very costly.

Another problem is the tension between abstraction, expressiveness and applicability of a module. In simplification: the more abstract a signature of a module result is, the easier it is to implement the module, but the harder it is to use the module, having lost access to it's concrete properties. Without specialized module system mechanisms, this global tension can only be solved by making each module excessively powerful and general so that is has answers for any conceivable demands.

However, the usual practical approach is to gradually sacrifice abstraction as the program grows, whereas, especially for ensuring correctness of large programs, the abstraction is crucial [4]. Especially important is abstracting from the module's context; e.g. from the identity of module parameters, as in the functor-based modular programming style. Unfortunately current module systems without functors tend to hard-wire dependencies from other modules, at least revealing their identities or identities of their types and sometimes even the implementation of the types.

Huge collections of interdependent modules themselves require modularization, lest managing the modules becomes as tedious as tending the mass of individual entities in non-modular programs. Grouping of modules performed using the mechanism of submodules often overloads the programmer with type sharing bureaucracy or requires a violation of the abstraction guarantees. Other solutions, some of them using external tools, such as a file-system, tend to be cheaper, but are usually very crude and incur the risk of name-space clashes or even ignore type abstraction.

1.2 The Module System

The Dule project is an experiment in large-scale fine-grained modular programming employing a terse notation based on an elementary categorical model. The Dule module system remedies the known bureaucratic, debugging and maintenance problems of functor-based modular programming (SML, OCaml) by introducing simple modular operations with succinct notation inspired by the simplicity of its semantic categorical model and a modularization methodology biased towards program structuring, rather than code-reuse [1] (the latter is partially supported through an auxiliary layer of the module system, not described here). The same categorical model and its natural extensions induce an abstract machine [5] based on their equational theories and inspire novel functional core language features that sometimes complement nicely the modular mechanisms and sometimes are language experiments on their own (not described here either; see the formal definition of Dule [12]).

The assets of the Dule project are gathered on its homepage at `http://www.mimuw.edu.pl/~mikon/dule.html`, where the formal definition of the current version of the language as well as an experimental compiler with a body of Dule programs can be obtained. The compiler works adequately, but it still does not bootstrap and the lack of low-level libraries precludes practical applications. Various aspects of modular programming in Dule are demonstrated on 10000 lines of fine-grained modular Dule code, including a fine-grained modular rewrite of the main parts of the compiler itself, but surely much more will be needed to discover all scaling problems and fine-tune the modularization methodology.

```
spec DrawChart =                  DrawChart =
~YearTable ~Picture ->              struct
  sig                                 value draw =
    value draw : Picture.t              if YearTable.patents_expired
  end                                     then Picture.ok
                                        else Picture.crash
                                    end
```

Fig. 1. An example signature and module in the full, sugared version of Dule. The application of the module to its arguments (all of which should be defined earlier, not shown here) is implicit; notice also the small signature and module headers.

Below we give an overview of the features of the full version of Dule module system. We refer the reader to the long informal tutorial of Dule available from its homepage that accompanies the Dule formal definition for an illustrated overview. Highlights of Dule:

- module composition with both transparent and non-transparent versions (the latter is the categorical composition from the semantic model)
- various module grouping mechanisms (including module categorical product operations)
- no sharing equations (nor '`with`' clauses nor type abbreviations); default sharing by names
- no substructures (submodules)
- pseudo-higher-order notation for module signatures (later flattened, because there are no higher-order modules; however, type dependencies on parameters remain; not discussed here)
- recursive signatures (flattened in a more complex way; not discussed here)
- (co)inductive modules (mutually dependent modules [6] constructed using inductive types instead of recursive types; not discussed here)
- default implicit module composition
- signatures of transparent functor applications [13] are expressible (actually all signatures of module operations are expressible)

- compositional semantics ensures separate compilation
- no references to environments in the semantics ensures the types from module parameters are abstract; on the other hand the abstraction can be sidestepped in a controlled way and limited scope, if necessary
- no dependencies between declared core language level entities (all dependencies are expressed through modules)

1.3 The Abstract Machine

The compiler of Dule, after type and signature reconstruction, computes the semantics of modular programs in the language of our categorical abstract machine. The semantics, that is the machine code, can then be executed by the machine, yielding results as expected in a programming language. The elementary notion of 2-category [11] with products is the categorical model of the abstract machine. An equational theory of 2-categories is the basis for the typed combinator reduction engine [7] that powers the machine.

The abstract mathematical model of the machine helped in proving its properties, in particular type-soundness and confluence. Preserving type information in the machine language offers many benefits, among them the ability to verify type-correctness of any received piece of machine code. The machine is very simple, especially considering that even advanced module operations can be performed exclusively on the machine language module representations, without the need to consult any additional annotations or modules' sources.

It is possible to extend the machine (and its mathematical model) with function types, sum types, inductive and coinductive types and general recursion (the reduction rules of our basic machine are listed in the Appendix, rules for the extended machine can be found in Appendix A.1.4 of [12]). The resulting machine language, with some syntactic sugar and type reconstruction, makes for an interesting core programming language, making available to the user, e.g., raw categorical composition (explicit substitution) and rigorously typed built-in structured (co)recursion. The construction of the module system on top of the abstract machine carries over to such extensions. The machine can also be extended to execute fully typed OCaml or Standard ML code. Inversely, simply typed λ-calculus with products, system F and ML can be presented as 2-categories with products, as needed for modeling our module system. In such presentations, the vertical composition in the 2-categories would be substitution and the composition of 1-morphisms would be type instantiation.

For our purposes, let's define 2-category as comprising of two ordinary categories and, additionally, a family of categories $C(c, e)$. The first of the two categories is the underlying category U, that is, the category of objects and 1-morphisms with the composition of 1-morphisms. Then, for each pair of objects c, e, there is a category $C(c, e)$ of all 1-morphisms with source c and target e as objects and all 2-morphisms between them as morphisms with their vertical composition. Horizontal composition yields the category of objects and 2-morphisms H, with the identity on object c equal to the 2-identity on the 1-identity on object c.

Cat, the category of all (small) categories is an example of a 2-category. By analogy to **Cat** we will call objects 'categories', 1-morphisms 'functors' and 2-morphisms 'transformations'. The 2-categories that are models of our abstract machine have a distinguished category (object) `*` and finite products in the U and H categories and in all categories $C(c, *)$. If we take **Set** (the category of sets and functions) for the distinguished category `*`, then **Cat** has all the required products, so it is a model of our abstract machine. We will denote a finite U-product of categories $c_1, \ldots, c_n$ labeled $i_1, \ldots, i_n$, respectively, by $\texttt{<}i_1 \texttt{ - } c_1\texttt{;}\ \ldots\texttt{;}\ i_n \texttt{ - } c_n\texttt{>}$. Products in $C(c, *)$ will be written $\texttt{\{}i_1 \texttt{ : } f_1\texttt{;}\ \ldots\texttt{;}\ i_n \texttt{ : } f_n\texttt{\}}$.

Below we present our chosen syntax and typing of basic operations of 2-categories with products, which is also the typing of the combinators of the abstract machine. The 'record' operations that appear below are generalized labeled tuples. First, we assign the U-source and U-target categories to functors (which can be seen as types of the core language).

$$\text{(U-identity)}\ \frac{}{\texttt{F\_ID}(c) : c \to c} \qquad \frac{f : c \to d \qquad g : d \to e}{f\ \texttt{.}\ g : c \to e}\ \text{(U-composition)}$$

$$\frac{}{\texttt{F\_PR}(lc, i) : \texttt{<}i \texttt{ - } c\texttt{;}\ \ldots\texttt{>} \to c}\ \text{(U-projection)}$$

$$\frac{f_1 : c \to e_1 \quad \cdots \quad f_n : c \to e_n}{\texttt{<}i_1 \texttt{ : } f_1\texttt{;}\ \ldots\texttt{;}\ i_n \texttt{ : } f_n\texttt{>} : c \to \texttt{<}i_1 \texttt{ - } e_1\texttt{;}\ \ldots\texttt{;}\ i_n \texttt{ - } e_n\texttt{>}}\ \text{(U-record)}$$

$$\frac{f_1 : c \to * \quad \cdots \quad f_n : c \to *}{\{i_1 \texttt{ : } f_1\texttt{;}\ \ldots\texttt{;}\ i_n \texttt{ : } f_n\} : c \to *}\ \text{(C(c, *)-product)}$$

Now we present the $C(c, e)$-domains and codomains of transformations (generalized values of a programming language). Many of the rules below have additional, unwritten premises ensuring that the terms that appear in them have compatible source and target categories. For example, in rule (H-comp) we require that the target of functor f_1 is equal to the source of functor f_2.

$$\text{(H-id)}\ \frac{}{\texttt{T\_ID}(c) : \texttt{F\_ID}(c) \to \texttt{F\_ID}(c)} \qquad \frac{t_1 : f_1 \to h_1 \qquad t_2 : f_2 \to h_2}{t_1 * t_2 : f_1\ \texttt{.}\ f_2 \to h_1\ \texttt{.}\ h_2}\ \text{(H-comp)}$$

$$\text{(C(c, *)-id)}\ \frac{}{(\texttt{:}\ g) : g \to g} \qquad \frac{t : f \to g \qquad u : g \to h}{t\ \texttt{.}\ u : f \to h}\ \text{(C(c, *)-comp)}$$

$$\text{(H-pr)}\ \frac{}{\texttt{T\_PR}(lc, i) : \texttt{F\_PR}(lc, i) \to \texttt{F\_PR}(lc, i)} \qquad \frac{}{i : \{i \texttt{ : } f\texttt{;}\ \ldots\} \to f}\ \text{(C(c, *)-pr)}$$

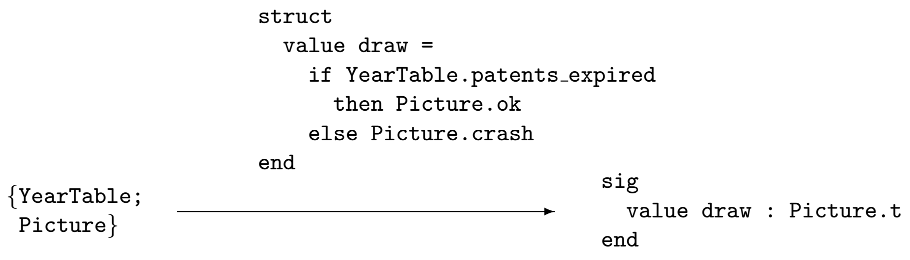
$$\frac{t_1 : f_1 \to h_1 \quad \cdots \quad t_n : f_n \to h_n}{\texttt{<}i_1 \texttt{ = } t_1 \texttt{;} \ \ldots \texttt{;} \ i_n \texttt{ = } t_n\texttt{>} : \texttt{<}i_1 \texttt{ : } f_1 \texttt{;} \ \ldots\texttt{>} \to \texttt{<}i_1 \texttt{ : } h_1 \texttt{;} \ \ldots\texttt{>}} \text{(H-record)}$$

$$\frac{t_1 : f \to h_1 \quad \cdots \quad t_n : f \to h_n}{\{i_1 \texttt{ = } t_1 \texttt{;} \ \ldots \texttt{;} \ i_n \texttt{ = } t_n\} : f \to \{i_1 \texttt{ : } h_1 \texttt{;} \ \ldots \texttt{;} \ i_n \texttt{ : } h_n\}} \text{(C(c, $*$)-record)}$$

In the rules, many combinators are written in abbreviated form and their notation does not contain all the typing information, e.g., the C(c, $*$)-projection. Others are written including full typing annotations, e.g., the H-projection. Later we may sometimes switch between the abbreviated and not abbreviated notation. The $C(c, *)$-composition plays the role of substitution in a programming language and $C(c, *)$-projections can be used as variables, but all these operations are combinators — there are no free variables anywhere — otherwise it wouldn't be an abstract machine, but an interpreter.

2 Simple Category of Modules

The heart of my module system is its mathematical model, the Simple Category of Modules (SCM), which can be based on any 2-category with products that models the core language to be used inside modules (and induces an implementation of the core language on the categorical abstract machine corresponding to the 2-category) . Consequently, no dependent products or sums [18] or second-order polymorphism [2] or even function types are needed for the construction of SCM nor of the operations of the module system, to be presented in the next section. The construction of Simple Category of Modules (SCM) should look quite intuitive to a programmer with a minimal categorical background. The objects of this category are module signatures and the morphisms are modules themselves.



```
struct
  value draw =
    if YearTable.patents_expired
      then Picture.ok
    else Picture.crash
end
```

```
{YearTable;
 Picture}
```

```
sig
  value draw : Picture.t
end
```

Fig. 2. Objects and morphisms in SCM.

The domain of a module is the signature of its parameters and the codomain is the result signature of the module. The identity morphism of SCM is the identity module and the composition is the operation of supplying implementation of parameters to a module. Other kinds of module composition are categorically definable, as will be shown later.

$$\begin{tikzcd}\mathrm{Sign}_1 \arrow[r, "\mathrm{Mod}_1"] & \mathrm{Sign}_2 \arrow[r, "\mathrm{Mod}_2"] & \mathrm{Sign}_3\end{tikzcd}$$

Fig. 3. Composition in SCM.

Categorical products model parameters (for example, the domain of a module is usually the product of signatures of parameter modules) allowing the programmer to express a kind of 'module variables' as projections in SCM. Moreover there is enough equalizers (limits) in SCM to model type sharing specifications. Complex limits built using equalizers model type sharing among parameter modules and also between parameters and the result signature. The kind of equalizers to be used for the semantics of our module language has a simple construction in SCM.

Now we will proceed with the formal definition of SCM. For the rest of this paper let us fix an arbitrary 2-category with products. Objects and morphisms of the SCM will be built from the morphisms of the fixed 2-category. Since the category can be, in particular, **Cat** with `*` equal to **Set**, SCM has a set-theoretic semantics [17] and the signatures and modules can be thought of as (quite complex but not higher-order) functions. Our account here is somewhat simplified. For a completely strict and detailed account see [12].

2.1 Objects and morphisms

Objects of SCM are called signatures and are defined as follows.

Definition 1 *A functor* f : `<`i_1 `-` c_1`;` ...`;` i_k `-` c_k`>` $\rightarrow$ `*` *of the fixed 2-category is a signature if it can be presented as* `{`i_1 `:` f_1`;` ...`;` i_n `:` f_n`}` *for some categories* `lc` $=$ i_1 `-` c_1`;` ...`;` i_k `-` c_k *and functors* `lf` $= f_1, \ldots, f_n$: `<`i_1 `-` c_1`;` ...`;` i_k `-` c_k`>` $\rightarrow$ `*`. *The indexed list of categories* `lc` *is called the type part of* f*, while the indexed list* `lf` *is called the value part of* f*.*

This simple form of signatures resembles the syntax of simple Standard ML or OCaml signatures where `lc` would correspond to names of types and `lf` to types of values in module signature. In general, beside the names of types, `lc` will usually contain names and kinds of parameter modules with nested names of types (in our syntax for base module signatures the information about parameters is represented and passed around in so-called context signatures, see Section 3.1 below). Also, when f is the signature of a group (record) of modules, `lf` is an indexed list of types of value parts of the modules, as defined below. Regardless of the details of the programming mechanisms, our 2-category product operations are enough to capture the diversity with one categorical notion.

Definition 2 *A module is a triple of a functor* `f`*, a transformation* `t` *and a signature* `s` *such that there is a signature* `r` *satisfying the following conditions:*

1. `f` : `src r` $\rightarrow$ `src s`
2. `t` : `r` $\rightarrow$ `f . s`

where `src` *produces the source category of a functor and the dot in the second condition is the U-composition. The (uniquely determined) signature* `r` *is the categorical domain of the above module seen as a morphism of SCM and the signature* `s` *is the categorical codomain (also uniquely determined, because given in the triple). Functor* `f` *is called the type part of the module and* `t` *is called the value part.*

We will use the notation `m : r` $\rightarrow$ `s` to mark the categorical domain and the categorical codomain of module `m` in SCM. Concrete examples of triples constituting modules are given below.

2.2 Identity and composition

The identity module on signature `s` in SCM is the triple (`F_ID`(c), `(: s)`, `s`), where category c is the source of `s`. So, for example, an identity on an empty signature will be (`F_ID(<>)`, `(: {})`, `{}`).

Composition in SCM is the operation of supplying implementation of parameters to a module. Let us look at the OCaml code from the Dule compiler that generates the abstract machine code for the module composition operation `m_Comp`. The code should be self-explanatory except for `f_COMP f1 f2`, which is an abstract syntax notation for U-composition written `f1 . f2` in our concrete syntax, `t_FT f1 t2`, which is the multiplication from the left by a functor `f1`, that is H-composition (: `f1`) $*$ `t2` and `t_comp t1 it2`, which is the composition of transformations `t1 . it2`.

```
let m_Comp m1 m2 = (* : r1 -> s2 *)
  let f1 = Dule.type_part m1 (* : r1 -> s1 *) in
  let t1 = Dule.value_part m1 in
  let f2 = Dule.type_part m2 (* : r2 -> s2 *) in
  let t2 = Dule.value_part m2 in
  let f = f_COMP f1 f2 in (* s1 = r2 *)
  let it2 = t_FT f1 t2 in
  let t = t_comp t1 it2 in
  let s2 = Dule.codomain m2 in
  Dule.pack (f, t, s2)
```

The following drawing shows the domains and codomains of the transformations appearing in the code of `m_Comp`, according to Definition 2. Transformations are here depicted by arrows. U-compositions are denoted by horizontal bars (double minuses).

```
     t1    f1          t2    f2          t    f1
r1 ------> --    r2 ------> --    r1 -----> --
           s1                s2               f2
                                              --
                                              s2
```

The drawing below illustrates the value part of the result of module composition. H-composition is here represented by placing the first transformation above the second. Vertical composition is represented by sharing a common domain/codomain. Transformation `(: f1)` is the identity on `f1`.

```
              (: f1)
     t1    f1 ------> f1
r1 ------> --         --
                      f2
           s1 ------> --       s1 = r2
                t2    s2
```

Modularly speaking: `m_Comp` instantiates the values of `m2` with the concrete implementation of the types given in `m1` and then applies the instantiated procedure to the values of `m1`. Consider the following composition written in the concrete syntax of our module language (to be defined shortly):

```
{M = :: {} -> sig type t1 value v1 : t1 end
     struct type t1 = {} value v1 = {} end}
.
:: {M : sig type t1 value v1 : t1 end} ->
     sig value v2 : M.t1 end
struct value v2 = M.v1 end
```

When the generated machine code is executed by the abstract machine, the implementation of value `v2`, which is the composition of projections `M.v1`, is multiplied from the left by the implementation of the types of the first module, that is

```
<M : <t1 :{}>>
```

resulting again in the composition of projections. The composition is then vertically composed (as the second operand) with the implementation of values of the first module, that is

```
{M : {v1 = {}}}
```

resulting in transformation `{}`, which is the value of `v2` in the outcome module.

After some more computation, the result of composition is seen to be expressible in our module language, using the mechanism of context signatures (the signatures written just after `sig`, automatically reconstructed in full Dule, see Section 3.1 below) to retain the signature `M`, as follows.

```
  :: {} ->
       sig {M : sig type t1 value v1 : t1 end}
         value v2 : M.t1
       end
  struct value v2 = {} end
```

Theorem 3 *The above definitions of SCM constituents determine a category, where signatures are objects, modules are morphisms and* `m_Comp` *is the composition.*

Proof. The domains and codomains of modules are well defined. The equalities about identity as the neutral element of composition follow promptly from the properties of identities in 2-categories. The associativity of composition is easy to establish, again using the properties of 2-categories.

2.3 Products

Products are necessary in our model to give semantics to modules that depend on many arguments. It turns out that SCM has (labeled) products.

Theorem 4 *Simple Category of Modules is cartesian.*

The proof (that we omit) is constructive, by defining in OCaml, similarly as we defined `m_Comp`, module operations of product of signatures, projection module and record module. The three operations are well defined; in particular when given correct operands they produce signatures and modules as required in Definition 1 and 2.

Once we have products, we can easily model in SCM a module system similar to the one of Standard ML but with no sharing requirements. Sharing requirements are used in modular programming to ensure that certain types, possibly appearing in distant modules, are equal. In a framework enabling abstraction, such as ours, guarantees of type equality are crucial to enable inter-operation between modules. In particular, sharing equations allow programmers to solve the diamond import problem, that is, express and automatically verify that two types coming from different two modules are compatible, because they originally come from a single module used to construct the two. However, sharing can be difficult to model. In particular, the ordinary labeled products of SCM do not suffice for this task.

2.4 Equalizers

Pullback of signatures (categorical limit of diagrams consisting of morphisms with a common codomain) is a good model of a signature of a pair of modules with some sharing between the two. Consider the following example, in which we write '`sharing type`' to mark an ad hoc notation for a sharing equation that, in this case, requires two product types to be equal (and if the product operation is injective in the fixed 2-category, then `M1.t` is equal to `M2.t` and `M2.u` is equal to `{}`).

```
{M1 : sig type t end;
 M2 : sig type t type u end;
 sharing type
   {t : M1.t; u : {}}
   = {t : M2.t; u : M2.u}}
```

Such a signature can be interpreted as a pullback of two morphisms (modules) from signatures `M1` and `M2`, respectively, to a common target (e.g., signature `sig type c end`). The morphisms determine the types to be identified. The first of the morphisms could look as follows:

```
struct type c = {t : M1.t; u : {}} end
```

and the second as follows:

```
struct type c = {t : M2.t; u : M2.u} end
```

However, pullbacks are not adequate for a similar task in case of many signatures. A sharing requirement may refer to components that are absent in some of the (possibly numerous) signatures, while the pullback construction is based on diagrams with morphisms from all the objects. We could overcome the problem by considering multiple sharing equations and adding a trivial equation for each signature absent from the main equation, but there are more elegant and efficient solutions.

A pullback of given morphisms can be constructed as an equalizer (categorical limit of a diagram of morphisms with the same domains and codomains; here generalized from two morphisms to a family of morphisms) of the morphisms prefixed with projections [19]. The projections come from the product of sources of the morphisms. This representation is valid for the labeled pullbacks as well, and involves labeled products and equalizers. If we allow the equalizer to be taken of a smaller family than the one used in the product, we can capture the sharing of components of only the chosen signatures. In fact, this construction is just the construction of the general limit of a diagram ([15], Theorem V.2.1). It turns out that a product of signatures with some sharing is just the categorical limit of the diagram formed by all the signatures and the morphisms representing the sharing requirement.

Let's suppose we are to represent categorically a collection of five signatures with a type shared among three of them.

```
{M1 : sig value v : {} end;
 M2 : sig type t end;
 M3 : sig type t end;
 M4 : sig type u type w end;
 M5 : sig type t end;
 sharing type
   M2.t = M3.t = M5.t}
```

First, we can represent the types to be shared as three morphisms (`t2`, `t3`, `t5`, in this case just identities) into a common target `T`. Then, to construct the limit, we take the product of the five signatures and compose the respective projections (`pi2`, `pi3`, `pi5`) with the three morphisms. The equalizer of the family of the three compositions is the sought limit signature `P`.

```
                     pi1
                M1  -----> M1
                 x   pi2        t2
                M2  -----> M2 ----____
   equalizer     x   pi3             \
P ----------->  M3  -----> M3 ---------> T
                 x   pi4       t3    __/
                M4  -----> M4    __/
                 x   pi5      __/   t5
                M5  -----> M5
```

If there are several sharing equations, the sought signature is again the limit of the diagram, this time containing several families of morphisms, each sharing a codomain. In the construction using product and equalizer, the equalizers representing consecutive sharing equations have to be composed. For a formalization of the main concepts see Section 5.1.4 of [12]. Here we only cite the result that has a constructive proof that guides the implementation of type sharing in our module system.

Lemma 5 *Let us fix an SCM over an arbitrary 2-category. For each categorical diagram in the SCM representing sharing requirement between whole type parts (the collections of all types) of modules, if the type names agree, a product signature with sharing represented by the diagram exists.*

In our module system we will apply a variant of the whole type parts sharing requirements, in which the modules to be equated are determined by names of nested signatures contained in a product. The sharing should take place between type parts of module signatures having the same labels. For instance, the following signature (featuring context signatures, as described in the next section) requires values `M1.v1` and `M2.v2` to have the same type.

```
{M1 : sig {M : sig type c end} type t1 value v1 : M.c end;
 M2 : sig {M : sig type c end} value v2 : M.c end}
```

In the following example, type `M.c` occurring in three context signatures and one main signature of the product will be shared in all four main signatures.

```
{M1 : sig {M : sig type c end} type t1 end;
 M2 : sig {M : sig type c end} end;
 M3 : sig {M : sig type c end} value v : M.c end;
 M : sig type c end}
```

The context signature of `M1` indicates that `M` is imported into `M1`, or rather that module `M1` is supposed to be built from `M`. The name `M`, assigned to one of the main signatures of the product, is interpreted as marking the same module that was used in construction of `M1`. The module `M` is, in this particular case, required to be provided separately as the fourth component of module record, perhaps to be used for building other modules later on.

Our operation, equating whole type parts of modules with the same names, has the same syntax as for the ordinary labeled product, as no explicit sharing specifications need to be written (notice the absence of explicit '`sharing type`' requirements in the above examples). The operation will be called product with name-driven sharing or (ambiguously but succinctly) just product. Every product with name-driven sharing is a categorical limit of a simple sharing diagram and can be constructed as a composition of a number of equalizers of whole type parts of modules (slightly generalized to allow nesting), taken on the product of signatures. The operations of projection with name-driven sharing and record with name-driven sharing are expressible in an analogous way using the operations of the ordinary product and the equalizer. The proof of Lemma 5 shows how to construct the equalizer operations using the abstract machine code.

3 Semantics/Implementation of the Module System

After constructing and analyzing the SCM we use it to develop a module system — a programming-oriented language for SCM (though without signature reconstruction and syntactic sugar it is not yet user-friendly, see [12] for that). If we fix a 2-category for the core language, we can construct the unique SCM built upon that 2-category. Our module system is an extension of the SCM (treated as a partial algebra) by several module operations, most of which are partial. The carriers are not extended and we don't need any additional assumptions on the core language, in particular we do not require function types.

The semantics of our module system is compositional and environment-free, which implies that the modules may be compiled separately. The correctness and the result of a given module operation depend only on the target code (SCM morphisms, abstract machine code) obtained from the operands, so the original source code can be compiled only once and then forgotten. The lack of any environments also ensures that module parameters are abstract. Upon supplying arguments the abstraction can be retained or overcome, depending on the operations used (composition vs. instantiation).

3.1 Basic operations

We start an overview of the typing and semantics of operations of our module system. In the typing rules we do not formulate additional definedness side conditions, hence the rules do not completely capture definedness properties of the operations. They only indicate whether a raw term belongs to the language and what its domain and codomain are. The complete definedness conditions are given and discussed in detail in the formal definition of Dule [12], where also the abstract machine code for the operations is given.

In the formal definition we also argue that each case of partiality of any of the modular operations corresponds to a class of modular programming errors. We prove that the semantics of the operations is well defined; in particular, their results belong to the set of objects and morphisms of the SCM. We check that the declared parameter and result signatures of modules coincide with SCM domains and codomains of the morphisms the module expressions denote. The proofs are straightforward, if long, because they were constructed alongside the construction of the semantics. The proof of well-definedness of the semantics of the module language constitutes a major part of the proof of correctness of the Dule compiler.

Here are the rules for the already defined identity and composition operations and for base modules that use elements available from an argument satisfying signature r to define core language types and values as specified in s.

$$\text{(identity)}\ \frac{}{(\texttt{:}\ s) : s \to s} \qquad \frac{m_1 : r_1 \to s \qquad m_2 : s \to s_2}{m_1\ \texttt{.}\ m_2 : r_1 \to s_2}\ \text{(composition)}$$

$$\frac{s = \texttt{sig}\ r'\ \texttt{type}\ i\ \ldots\ \texttt{value}\ j\ \texttt{:}\ f\ \ldots\ \texttt{end}}{(\texttt{::}\ r\ \texttt{->}\ s\ \ \texttt{struct type}\ i\ \texttt{=}\ g\ \ldots\ \texttt{value}\ j\ \texttt{=}\ t\ \ldots\ \texttt{end}) : r \to s}\ \text{(base)}$$

Signatures of the same form as s, in the last rule, are called base signatures and r' inside them can specify less components than the corresponding r. Such r', occurring inside s, is called a context signature, on which types of values in s may depend (if r is `{}`, it is usually omitted in writing; in sugared Dule syntax r disappears altogether). Base signatures can be viewed as products with name-driven sharing (as defined in the next subsection) of micro-signatures $f, \ldots$, while base modules are isomorphic to records with name-driven sharing of micro-modules that only define one type or one value.

3.2 Cartesian structure

As told in Section 2.3, SCM is cartesian, which enables parameterization by named modules. Inside a modular expression with product domain the parameter modules are treated as 'locally' abstract. When several module expressions are put within a module record with a product domain, they all share the same locally abstract arguments. However, this abstraction does not prevent specifying equalities between individual types.

For the semantics of our module system we choose to employ implicit sharing of whole type parts of modules, determined by names of components, as illustrated in Section 2.3. When signatures are grouped in a product, all top-level context types (types inherited from context signatures) with the same name are to be shared. In the result, top level type components of a product of signatures are the sets of types defined in the signatures themselves indexed by the signature names, together with all their context types.

This particular merger of labeled product and equalizer, will be called product with name-driven sharing, or just product. The notation for the operations is the same as for the ordinary labeled categorical product. This setup results in concise syntax, with somewhat limited expressiveness, which is recoverable with the help of instantiation operation (e.g., to rename module parameters) described later on.

$$\frac{}{i : \{i \texttt{ : } r\texttt{;} \ldots\} \to r} \text{ (projection)}$$

$$\frac{m_1 : r \to s_1 \quad \cdots \quad m_n : r \to s_n}{\{i_1 \texttt{ = } m_1\texttt{; } \ldots\texttt{; } i_n \texttt{ = } m_n\} : r \to \{i_1 \texttt{ : } s_1\texttt{; } \ldots\texttt{; } i_n \texttt{ : } s_n\}} \text{ (record)}$$

Whenever the product operations are defined, they satisfy all the equalities required of a categorical product. Moreover, whenever they are undefined their signature operands show that the programmer tried to impose a contradicting sharing requirement, or their module operands show that the programmer violated the sharing requirements he had imposed earlier.

Observation 6 *The product signature with the projection modules form a categorical cone over the diagram of sharing requirement representing name-driven sharing among the product operands.*

3.3 Specialization, instantiation and trimming

There are two modular operations: instantiation and trimming, that have no clear categorical meaning in the context of SCM, though they have a simple mathematical definition in terms of 2-categories with products.

$$\text{(instantiation) } \frac{m_1 : r_1 \to s \qquad m_2 : s \to s_2}{m_1 \texttt{ | } m_2 : r_1 \to m_1 \texttt{ | } s_2} \qquad \frac{m_1 : r_1 \to s_1}{m_1 \texttt{ :> } r_2 : r_1 \to r_2} \text{ (trimming)}$$

The operation $m_1 \texttt{ | } s_2$ specializes signature s_2 to a narrower field of use — restricted to the types of module m_1. The operation $m_1 \texttt{ | } m_2$ instantiates module m_2 so that it has more concretely defined manner of operation and more strictly specified field of operation, determined by module m_1.

The instantiation operates on the value parts of operand modules in exactly the same way as the module composition `m_Comp` does. Yet instantiation is not a good candidate for an alternative composition in SCM: it is not always defined even if codomain of the first operand agrees with the domain of the second. A second phenomenon differentiating instantiation from composition is that the codomain of the whole operation may differ from the codomain of the second operand. This implies that instantiation is not associative.

Despite not having the pleasant properties of composition, instantiation has a profound programming meaning. While composition corresponds to non-transparent functor application [8], where arguments are used as tools only, instantiation corresponds to transparent application [13], where arguments specialize the output signature of the functor.

The trimming operation performs a coercion of a module to a given signature. If necessary, some type and value components of an operand module are removed, and only those mentioned in the operand signature remain. Ideally trimming should be absent from the language, but it is indispensable when we want to present an instantiated module as if it had not been instantiated. Moreover, when a larger module is used in a role of a smaller one, trimming relieves the user from writing the interface module.

3.4 Linking

The linking operation is the Dule standard way of supplying modules with arguments. It is defined (just as two other, simpler grouping operations, omitted here due to space constraints) using the already described operations of our module system. This witnesses the power of the system, as well as greatly simplifies proving properties of linking — in particular, the proof of its well-definedness and the adequacy of its declared domains and codomains.

$$\frac{\begin{array}{c} m_1 : \{i_{k_1} \mathtt{:}\ s_{k_1}\mathtt{;} \ldots \mathtt{;}\ j_1 \mathtt{:}\ r_1\mathtt{;} \ldots\} \to s_1 \\ \vdots \\ m_n : \{i_{k_n} \mathtt{:}\ s_{k_n}\mathtt{;} \ldots \mathtt{;}\ j_n \mathtt{:}\ r_n\mathtt{;} \ldots\} \to s_n \end{array}}{\begin{array}{c} \mathtt{link}\ \{i_1\ \mathtt{=}\ m_1\mathtt{;}\ \ldots\mathtt{;}\ i_n\ \mathtt{=}\ m_n\} : \\ \{j_1 \mathtt{:}\ r_1\mathtt{;}\ \ldots\mathtt{;}\ j_n \mathtt{:}\ r_n\mathtt{;}\ \ldots\} \to \{i_1 \mathtt{:}\ s_1\mathtt{;}\ \ldots\mathtt{;}\ i_n \mathtt{:}\ s_n\} \end{array}} \ \text{(linking)}$$

The domains of linking operands have to be product signatures, and needn't be strictly equal to each other. It suffices if there are equal components at the same labels. The domains may contain components not present in the domain of the whole linking expression but each of these has to be a codomain of one of the operand modules, indexed by the name of the module. These components will not be left over in the domain of the linking operation, but will be the places at which compositions with other operands occur immediately. Like in the module record operation, the codomain of the linking expression is just the product of codomains of all operand modules.

The linking operation eases the grouping of modules (already enabled by the module record operation) that alleviates the need for submodules, as known from conventional module systems. The linking operation fits well with our choice of sharing mechanism. The module product operation assumes that components with the same names are shared, while the linking operation assumes parameters are implemented by modules of the same names. Inside linking expression, supplying implementation of the parameters is automatically performed with multiple compositions and the user is free from writing the compositions by hand. There is also no need to artificially introduce submodules for the purpose of specifying their sharing with others; together with implicit composition this greatly reduces common modular bureaucracy. Linking facilitates naming and applying modules, but the language remains first-order, readily compilable to the abstract machine language and the implementation avoids copying or storing the source code of modules.

References

1. Anya Helene Bagge, Martin Bravenboer, Karl Trygve Kalleberg, Koen Muilwijk, and Eelco Visser. Adaptive Code Reuse by Aspects, Cloning and Renaming. Technical Report UU-CS-2005-031, Institute of Information and Computing Sciences, Utrecht University, 2005.
2. E. S. Bainbridge, P. J. Freyd, A. Scedrov, and P. J. Scott. Functorial polymorphism. *Theoretical Computer Science*, 70(1):35–64, January 1990.
3. Dave Berry. Lessons from the design of a Standard ML library. *Journal of Functional Programming*, 3(4):527–552, October 1993.
4. Michel Bidoit, Donald Sannella, and Andrzej Tarlecki. Architectural specifications in Casl. *Formal Aspects of Computing*, 13:252–273, 2002.
5. G. Cousineau, P. L. Curien, and M. Mauny. The Categorial Abstract Machine. *Science of Computer Programming*, 8:173–202, 1987.
6. Karl Crary, Robert Harper, and Sidd Puri. What is a Recursive Module? In *SIGPLAN Conference on Programming Language Design and Implementation*, 1999.
7. P.-L. Curien. Categorical Combinators. *Information and Control*, 69(1-3):189–254, 1986.
8. Robert Harper and Mark Lillibridge. A Type-Theoretic Approach to Higher-Order Modules with Sharing. In *Proceedings of the ACM Conference on Principles of Programming Languages*, pages 123–137, Portland, Oregon, January 1994.
9. Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-Order Modules and the Phase Distinction. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 341–354, San Francisco, CA, January 1990.
10. Robert Harper and Benjamin C. Pierce. Advanced module systems: a guide for the perplexed. In *Proceedings of the ACM Sigplan International Conference on Functional Programming (ICFP-00)*, volume 35.9 of *ACM Sigplan Notices*, pages 130–130, N.Y., September 18–21 2000. ACM Press.
11. C. B. Jay. An introduction to categories in computing. Technical Report UTS-SOCS-93.9, University of Technology, Sydney, 1993.
12. Mikołaj Konarski. *Application of Category-Theory Methods to the Design of a System of Modules for a Functional Programming Language.* PhD thesis, MIMUW, 2007.
13. Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *Proc. 22nd symp. Principles of Programming Languages*, pages 142–153. ACM Press, 1995.
14. Xavier Leroy. The Objective Caml system: Documentation and user's manual, 2000. With Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. Available from `http://caml.inria.fr`.
15. S. MacLane. *Categories for the Working Mathematician.* Springer-Verlag, 1971.
16. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* The MIT Press, 1997.
17. John C. Reynolds. Polymorphism is not set-theoretic. In Gilles Kahn, David B. MacQueen, and Gordon D. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, Berlin, 1984. Springer-Verlag.
18. Claudio V. Russo. Non-dependent Types for Standard ML Modules. In *Principles and Practice of Declarative Programming*, pages 80–97, 1999.
19. D. Sannella and A. Tarlecki. Category theory. In *Foundations of Algebraic Specifications and Formal Program Development*, chapter 3. Cambridge University Press, to appear. See `http://wwwat.mimuw.edu.pl/~tarlecki/book/`.

20. Perdita Stevens. Experiences with the ML Module System, or, Why I Hate ML. Transparencies for a talk given to the Edinburgh ML Club and Glasgow Functional Programming group. `http://www.dcs.ed.ac.uk/home/pxs/talksEtc.html`, 1998.

Appendix: Abstract Machine Execution

Below, we abuse notation, writing multiplications by a functor, that is horizontal compositions with $C(c,e)$-identity, without the identity term constructor, as in $f * \texttt{<}i\ \texttt{=}\ u\texttt{;}\ \ldots\texttt{>}$, which is intended to mean $\texttt{T\_id}(f) * \texttt{<}i\ \texttt{=}\ u\texttt{;}\ \ldots\texttt{>}$. Our abstract machine executes machine code by combinator reduction as follows, where we list the reduction rules for functors first.

$$f\ \texttt{.}\ \texttt{F\_ID}(d) \to f \tag{1}$$
$$\texttt{F\_ID}(c)\ \texttt{.}\ g \to g \tag{2}$$
$$f\ \texttt{.}\ (g_1\ \texttt{.}\ g_2) \to (f\ \texttt{.}\ g_1)\ \texttt{.}\ g_2 \tag{3}$$
$$\texttt{<}i\ \texttt{:}\ f_i\texttt{;}\ \ldots\texttt{>}\ \texttt{.}\ \texttt{F\_PR}(ld,i) \to f_i \tag{4}$$
$$f\ \texttt{.}\ \texttt{<}i\ \texttt{:}\ g\texttt{;}\ \ldots\texttt{>} \to \texttt{<}i\ \texttt{:}\ f\ \texttt{.}\ g\texttt{;}\ \ldots\texttt{>} \tag{5}$$
$$f\ \texttt{.}\ \texttt{\{}i\ \texttt{:}\ g\texttt{;}\ \ldots\texttt{\}} \to \texttt{\{}i\ \texttt{:}\ f\ \texttt{.}\ g\texttt{;}\ \ldots\texttt{\}} \tag{6}$$

Below are the reduction rules for transformations.

$$f * \texttt{<}i\ \texttt{=}\ u\texttt{;}\ \ldots\texttt{>} \to \texttt{<}i\ \texttt{=}\ f * u\texttt{;}\ \ldots\texttt{>} \tag{7}$$
$$f * \texttt{T\_id}(g) \to \texttt{T\_id}(f\ \texttt{.}\ g) \tag{8}$$
$$f * (u_1\ \texttt{.}\ u_2) \to (f * u_1)\ \texttt{.}\ (f * u_2) \tag{9}$$
$$t * \texttt{F\_ID}(d) \to t \tag{10}$$
$$t * (f\ \texttt{.}\ g) \to (t * f) * g \tag{11}$$
$$\texttt{<}i\ \texttt{=}\ t_i\texttt{;}\ \ldots\texttt{>} * \texttt{F\_PR}(ld,i) \to t_i \tag{12}$$
$$\texttt{T\_id}(f) * \texttt{F\_PR}(ld,i) \to \texttt{T\_id}(f\ \texttt{.}\ \texttt{F\_PR}(ld,i)) \tag{13}$$
$$(t_1\ \texttt{.}\ t_2) * \texttt{F\_PR}(ld,i) \to (t_1 * \texttt{F\_PR}(ld,i))\ \texttt{.}\ (t_2 * \texttt{F\_PR}(ld,i)) \tag{14}$$
$$t * \texttt{<}i\ \texttt{:}\ g\texttt{;}\ \ldots\texttt{>} \to \texttt{<}i\ \texttt{=}\ t * g\texttt{;}\ \ldots\texttt{>} \tag{15}$$
$$\texttt{T\_ID}(c) \to \texttt{T\_id}(\texttt{F\_ID}(c)) \tag{16}$$
$$\texttt{T\_PR}(lc,i) \to \texttt{T\_id}(\texttt{F\_PR}(lc,i)) \tag{17}$$
$$t\ \texttt{.}\ \texttt{T\_id}(g) \to t \tag{18}$$
$$\texttt{T\_id}(f)\ \texttt{.}\ t \to t \tag{19}$$
$$t\ \texttt{.}\ (u_1\ \texttt{.}\ u_2) \to (t\ \texttt{.}\ u_1)\ \texttt{.}\ u_2 \tag{20}$$
$$\texttt{\{}i\ \texttt{=}\ t_i\texttt{;}\ \ldots\texttt{\}}\ \texttt{.}\ \texttt{T\_pr}(lg,i) \to t_i \tag{21}$$
$$t\ \texttt{.}\ \texttt{\{}i\ \texttt{=}\ u\texttt{;}\ \ldots\texttt{\}} \to \texttt{\{}i\ \texttt{=}\ t\ \texttt{.}\ u\texttt{;}\ \ldots\texttt{\}} \tag{22}$$
$$f * \texttt{T\_pr}(i\ \texttt{:}\ g\texttt{;}\ \ldots, j) \to \texttt{T\_pr}(i\ \texttt{:}\ f\ \texttt{.}\ g\texttt{;}\ \ldots, j) \tag{23}$$
$$f * \texttt{\{}i\ \texttt{=}\ u\texttt{;}\ \ldots\texttt{\}} \to \texttt{\{}i\ \texttt{=}\ f * u\texttt{;}\ \ldots\texttt{\}} \tag{24}$$
$$t * u \to (f * u)\ \texttt{.}\ (t * h) \tag{25}$$