Call-by-Value Semantics for Mutually Recursive First-Class Modules

Judith Rohloff and Florian Lorenzen

Technische Universität Berlin
Compiler Construction and Programming Languages
Ernst-Reuter-Platz 7, D-10587 Berlin
{judith.rohloff,florian.lorenzen}@tu-berlin.de

Abstract. We present a transformation based denotational semantics for a call-by-value language with first-class, hierarchical and recursive modules. We use the notion of *modules* as proposed in [1]. They merge dynamic data structures with aspects of modularisation and name binding in functional programming languages. *Modules* are first-class values which capture recursive definitions, lexical scoping, hierarchical structuring of programs and dynamically typed data structures in a single construction. We define a call-by-value language ModLang and explain what problems occur in combining nested, recursive and first-class modules. We then show how to solve these problems by defining a dependency analysis to determine the evaluation order, enabling a transformation into an intermediate representation. Finally, we present a denotational call-by-value semantics.

1 Introduction

Modularisation in an essential tool for managing complexity in the design of large scale software. To be effective, module systems must organise code into fine grained hierarchies without impeding reusability. The competing nature of these goals has given rise to several approaches for designing module systems. With parametric modules, we obtain adaptable code that is easy to reuse at different parts of the architecture. First-class modules [2,3,4], i.e. modules that can be arguments and results of functions, allow parametric modules to be defined as normal functions, so no special construct is needed. Furthermore, first-class modules enable runtime reconfiguration of the architecture. With nested modules [5] a fine grained module hierarchy is possible. Disallowing mutually recursive modules [6,7,8] often destroys the natural structure of a program.

Recent approaches [9,8,10] have attempted to combine all three features in a call-by-value language, but none of them have achieved full support. In this paper we describe a way to give a call-by-value semantics to a language with higher-order functions and nested, first-class, mutually recursive modules.

We use the grouping concept proposed in [1] as modules and call the proposed construct *module*. Pepper introduces the concept of *modules* as a unified construct for bindings and data structures. *Modules* are sets of definitions. As first-class values, they can also be used as records. They distinguish from records in that all definitions within a module can see each other. In [1] *modules* are treated coalgebraically and regarded as objects implicitly defined by their observers.

In contrast to this semantic approach, we focus here on an efficient implementation by defining formal semantics and analyses for *modules*. For this purpose we define a small functional call-by-value language ModLang, which is similar to an untyped λ-calculus extended by *modules*.

As *modules* are first-class values, have binding definitions and are allowed to be mutually recursive, it is not possible to give a straightforward semantics. In this paper we describe a way to give a transformation-based call-by-value semantics to *modules*.

The paper starts with a short introduction of the *module* concept. In Sec. 3 we describe why it is not possible to give a straightforward semantics and introduce our solution. In a call-by-value semantics, an evaluation order is needed, as every variable must be bound to a value before it is used. There are two possibilities to obtain this order. Either the programmer has to state all definitions in dependency order (e.g. ML), or the order is determined via dependency analysis (e.g. Opal or Modula-3 [11]). As *modules* are sets of definitions, no order is given. The dependency analysis for our language is described by an example in Sec. 4. In Sec. 5 the result of the dependency analysis is used to transform the input program into an intermediate representation with explicit dependency order. In Sec. 6 we present a denotational semantics for our intermediate representation. Sec. 7 discusses related work and Sec. 8 summarises our results and gives an overview of current and future work.

2 The ModLang Language

ModLang is a functional language with *modules*. We describe the language ModLang by some examples. The syntax is given in Sec. 4.

2.1 Introductory Example: Lists

Abstractions, applications and conditionals have their usual meaning. We focus on *modules* and selections.

Following [1] we define a *module* as a set of named definitions. Listing 1.1 shows the well-known example of lists. The *module* `List` contains five definitions `nil`, `cons`, `head`, `tail` and `enum`.[1] In this example the *module* represents a module encapsulating definitions. Since the order of the definitions is irrelevant, it can be seen as a set — hence the surrounding curly braces.

[1] A usable list implementation would require the discriminators `isNil` and `isCons`. They can be implemented using the *module* discriminator `DEFINES`, which checks if a given *module* defines exactly the given variables. As `DEFINES` is a dynamic check, we do not consider it in this paper.

```
List = {
    nil  = {}
    cons = λx.λxs. { hd=x  tl=xs }
    head = λxs. xs.hd
    tail = λxs. xs.tl
    enum = λn. { enum = λi.
                    IF i==n THEN nil ELSE cons i (enum (i+1))
                }.enum 0
}
```

Listing 1.1. Lists in ModLang

Both constructors, `nil` and `cons`, implement the list elements with *modules* as well — `nil` is the empty *module* and `cons` returns a *module* containing the head `hd` and the rest `tl` of the list. In this case *modules* are used as data structures, which can be built up and changed at runtime.

The function `head` (`tail`) selects the first element (the rest) of the given list via the selection operator "`.`". In this manner these functions abstract the selectors `.hd` and `.tl`.

The function `enum` forms a list of numbers between `0` and `n`. This definition illustrates several aspects of our language design:

– All variables of the outer *module* can be used on the right hand side of a definition. In this case these are `nil` and `cons`.
– Within the expression an anonymous *module* containing the definition `enum` is defined. We call a *module* anonymous if it is not the top level node of the syntax tree of a right-hand side of a definition.
– In the definition of `enum` of the anonymous *module* the variable `enum` is used. Because of the lexical scoping, the innermost definition is addressed. Thus it is `enum` of the anonymous *module* and not `List.enum`.
– On the right hand side of `List.enum` the selector `.enum` is called. The anonymous *module* and its selection are used as a local recursive "let-in" or "where".

2.2 Functional-Object-Oriented Programming

Our second example shows, how we can use *modules* to program in an object oriented manner. Listing 1.2 shows an eval-apply-interpreter with environments for the untyped λ-calculus. A term is either a variable `Var`, an abstraction `Abs` or an application `App`. These term constructors are quite similar to the "objects-as-closures" implementation [12], but here they are evaluated to *modules* instead of closures. Those *modules* contain an evaluation function `eval`, which uses an environment to compute the value of the term. In doing so, the sub-terms' evaluation functions may also be used.

During the evaluation of a variable, its value is determined from the environment Γ (line 2). To evaluate an abstraction, a closure (i.e. *module*) with the abstract variable `var`, the function body `body` and the current environment `env` must be built (line 3).

```
Term = {
  Var    = λx. { eval = λΓ. Γ.lookup x }
  Abs    = λx.λt. { eval = λΓ. { var=x  body=t  env=Γ } }
  App    = λt1.λt2. { eval  = λΓ. apply (t1.eval Γ) (t2.eval Γ)
                      apply = λf.λa.
                        f.body.eval (f.env.add f.var a) }
}
```

Listing 1.2. An eval-apply-interpreter in a functional-objekt-oriented style

The implementation of the environment is shown in Listing 1.3. An environment is a list of records `{var=x val=v}` using the list implementation in Listing 1.1. An environment has the functions `add` to add a new binding and `lookup` to determine the value of a variable. The constructor `Env` creates an environment containing all bindings of `bdgs`.

In contrast to terms new environments must be created during evaluation. In `add` a new environment is created with `Env` which contains all previous bindings as well as the new binding `Bdg x v`

```
Env = λbdgs. {
  Bdg    = λx.λv. { var=x  val=v }
  add    = λx.λv. Env (List.cons (Bdg x v) bdgs)
  lookup = ... // searching in list
}
```

Listing 1.3. Environment for eval-apply-interpreter

As demonstrated by this example:

- Objects can be realised as *modules*. Methods are definitions within a *module*.
- Constructors of object oriented programming languages can be seen as functions returning *modules*.
- Variables, abstracted by λ within a constructor play the role of fields. It is also possible to define them in definitions, making them accessible from outside by selection.
- The binding rules for *modules* enable methods to have access to the methods and fields of its enclosing object. A special reference like `this` in Java or `self` in Smalltalk does not exist.

2.3 Mutually Recursive Modules

The list example has already shown that definitions may be recursive. Mutually recursive definitions are allowed as well, even across *module* borders. An example for mutually recursive *modules* is given in Listing 1.4. This program will be used as a running example throughout the rest of this paper. In this example the two functions `E.even` and `O.odd` are defined. Jointly they decide whether a given number is even or not. `O.odd` uses `E.even` and vice versa. Moreover these two functions are used by `E.is2even` and `O.is2odd`.

```
{ E = { even    = λn. IF n==0 THEN true ELSE O.odd (n-1)
        is2even = even val
        val     = 2 }
  O = { odd    = λn. IF n==0 THEN false ELSE E.even (n-1)
        is2odd = odd 2 }
}
```

Listing 1.4. Mutually recursive *modules*

3 Towards an Evaluation

In a call-by-name semantics no evaluation order is needed, as we can evaluate all definitions on demand. Therefore, a call-by-name semantics to ModLang can be readily given. Although it is a matter of taste which semantics one prefers, call-by-value languages are easier to combine with parallel programming and side effects. For these reasons, it is desirable to give a call-by-value semantics to ModLang.

As previously mentioned, implementing call-by-value semantics for ModLang requires a dependency analysis. As we combine all three features of our modules it is not easy to find the evaluation order. We describe the difficulties involved by starting with only nested modules and then adding mutual recursion. Finally we make our modules first-class values.

3.1 Nested Modules

We start with nested modules, which enable building up a hierarchy of *modules*. *Modules* are only allowed as top-level expression or as the right hand side of a definition. Definitions are not allowed to be mutually recursive. It follows that within every *module*, an order for all definitions can be found.

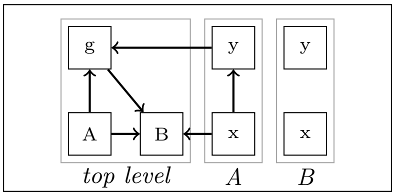
To obtain the evaluation order for the example in Listing 1.5 it is obvious that we first have to evaluate the complete *module* `B` with both definitions. Then the definition `g` can be evaluated, followed by the *module* `A`.

```
{A = {x = B.x + y    y = g}
 g = B.x + B.y
 B = {x = 1    y = 3}
}
```

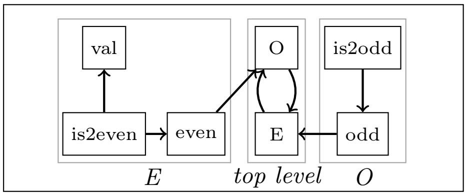
Listing 1.5. Example for evaluation order

For programs without first-class and mutually recursive modules, the order could be determined by regarding only the free variables of all right hand sides and computing a normal dependency graph. The dependency graph for this example is given in Fig. 1(a).

As will be shown in the next section, this approach is insufficient when mutually recursive modules are allowed.



(a) Dependency graph for the example in Listing 1.5.



(b) Dependency graph for the example in Listing 1.4.

Fig. 1. Dependency graphs using free variables

3.2 Mutually Recursive Modules

We now allow nested *modules* to be recursive. As before, they are still not allowed to be defined within an expression. As recursion in a call-by-value semantics only terminates if all definitions are functions we forbid cycles where at least one definition is not a function, as typical in most call-by-value languages. Even with this constraint it is still not easy to find the evaluation order.

To understand the problem with mutually recursive *modules*, we will again consider the even-odd example in Listing 1.4.

In a call-by-value semantics recursion is only possible for λ-abstraction, since those definitions evaluate to closures. Therefore, it is not possible to build up a closure for the entire *module*. This would only be possible if the auxiliary functions `is2even` and `is2odd` were not present. We have to find an order for all definitions within the two *modules* `E` and `O`. For the given example, the call-by-value semantics requires that `even` and `odd` must be evaluated before `is2odd` and `is2even`. It is not possible to find this order by only regarding the free variables. The dependency graph for this example with free variables is shown in Fig. 1(b). There is a cycle between `E` and `O`, which is forbidden as both definitions are *modules* and not functions. This contradicts the goal of allowing mutually recursive *modules*.

To calculate the correct evaluation order we look at dependencies across *module* borders and also consider selection chains (see definition in Sec. 4.2). In this example, a possible evaluation order is: `E.val`, `E.even`, `O.odd`, `E.is2even`, `E`, `O.is2odd`, `O`.

By using the complete selection chain for the dependency, the hierarchy is broken up and the mutually recursive *modules* are handled simultaneously.

In [1] a flattening mechanism is proposed to find an evaluation order. Here the hierarchy is broken up and all definitions occur at the same level. The result of the flattening for example in Listing 1.4 is shown in Listing 1.6. Instead of just one identifier, the name on the left-hand-side of all definitions is the complete path with dot notation. Furthermore, all variables must be replaced by the complete name within the *module*, e.g `even` in the definition `is2even` must be replaced by `E.even`. Renaming is necessary, to ensure unique names after flattening as it could be possible to have the same definition name in two *modules*.

```
E.even    = λn. IF n==0 THEN true ELSE O.odd (n-1)
E.is2even = E.even val
E.val     = 2
O. odd    = λn. IF n==0 THEN false ELSE E.even (n-1)
O.is2odd = O.odd 2
```

Listing 1.6. Example for flattening

After flattening, we can build up a dependency graph. But instead of just using the free variables, we now use the complete selection chain.

Flattening is only possible for mutually recursive nested modules. The technique is not always applicable to first-class values, as shown by the example in the next section.

3.3 An Undecidable Problem

Up to now *modules* were only allowed as top level expressions or as the right-hand-side of a definition. If we allow *modules* to be first-class values, i.e. they are allowed at any position in an expression, then flattening is not possible anymore.

In the previous section we regarded the complete selection chain for the dependencies. This mechanism is quite similar to path resolution, which is undecidable as proven in [9]. We will just give an informal explanation here.

In Listing 1.7 an artificial example is given. For flattening the expression representing the complete selection chain `s.a.b` must be found. In this example this name only exists if the input fulfils some predicate `isValid`. This is generally not decidable.

```
λinput. {
  s      = IF isValid input THEN {a = 1}
                            ELSE {a = {b = 1}}
  output = s.a.b
}
```

Listing 1.7. A problematic example (Using `List` of Listing 1.1)

In this case it is sufficient to recognise that `output` depends on `s`, so the evaluation order is: `s`, `output`. In the case that `s.a` has no selector `b`, a runtime error will occur, which is acceptable and common for failed selection.

Consequentially, there are some cases where we have to regard the selection chain and some cases where parts of the chain should not be regarded. For every selection chain we regard only the "static" part and ignore the "dynamic" part. The static part of a selection chain is the variable and all selectors representing a path within the current *module* hierarchy. In this example it is just the lexically visible variable `s`, `.a.b` is the dynamic part, as these names only occur at runtime. The proper way to calculate the dependencies is described in Sec. 4.

3.4 Solution

As previously mentioned, we start with a special dependency analysis to determine the evaluation order. The result of the dependency analysis is used to

$$\begin{array}{lcl} E & ::= & M \mid T \\ T & ::= & [\lambda\ x\ .\ E]^\ell \mid [E\ E]^\ell \mid x^\ell \mid [E\ .\ x]^\ell \\ M & ::= & \{\ D^*\ \}^\ell \\ D & ::= & x = M \mid x = T \end{array}$$

Fig. 2. Syntax of labelled expressions

transform the whole input program into an intermediate representation with explicit ordering. This intermediate representation allows a simpler definition of the semantics.

4 Dependency Analysis

In this section we illustrate our special dependency analysis by means of the running example. We begin with the following definitions.

4.1 Labelled Expressions

The syntax of ModLang is given in Fig. 2. We distinguishes between *modules* and other expressions by splitting E into *modules* M and terms T and we omit conditionals and constants since they are not important for the rest of the paper. Furthermore we annotate all expressions $M \cup T$ with a unique label of the set ℓ and use brackets to clarify which part of an expression is labelled. The inverse of the labelling is the mapping $EXP : \ell \hookrightarrow M \cup T$. We use the labelling to attach additional information to expressions.

4.2 Names

Selector / Name. A *selector* is an identifier with a prefix dot. A *selector chain* is a sequence of selectors.

A *name* is a variable (identifier without a prefix dot) followed by a (possibly empty) selector chain. The set of all names is denoted by N.

Free and Available Names. Every expression E of a program is mapped to two sets using the labels ℓ:

$FN \quad : \ell \hookrightarrow \mathbb{P}N$	set of free names
$AVM : \ell \hookrightarrow \mathbb{P}x$	set of *module*-bound available variables

Free names extend the λ-calculus concept of free variables to names. Names are free iff the leading variable is free. The function $\mathcal{F}$ of Fig. 3 maps every expression to its free names.

Calculation of free variables and free names differs only in equation $(*)$ in Fig. 3. Here, the whole name $x_1 . \cdots . x_m$ is inserted instead of the variable x_1. The mapping FN is the composition $FN = \mathcal{F} \circ EXP$.

$$
\begin{array}{lll}
\mathcal{F} : \ M \cup T \to \mathbb{P}N & & \\
\hline
\mathcal{F}[\![\lambda x \,.\, E]\!] & = \mathcal{F}[\![E]\!] \ominus \{x\} & \\
\mathcal{F}[\![E_1 \ E_2]\!] & = \mathcal{F}[\![E_1]\!] \cup \mathcal{F}[\![E_2]\!] & \\
\mathcal{F}[\![x_1 . \cdots . x_m]\!] & = \{x_1 . \cdots . x_m\} & (*) \\
\mathcal{F}[\![E \,.\, x]\!] & = \mathcal{F}[\![E]\!] & \\
\mathcal{F}[\![x]\!] & = \{x\} & \\
\mathcal{F}[\![\{\!\!\{ \, x_i = E_i{}^{i \in 1..m} \, \}\!\!\}]\!] & = \left\{ N \mid N = y_1 . \cdots . y_p \wedge N \in \bigcup_{i \in 1..m} \mathcal{F}[\![E_i]\!] \wedge y_1 \notin \{x_i{}^{i \in 1..m}\} \right\} &
\end{array}
$$

Fig. 3. Function calculating free names

$$
\begin{array}{lll}
\mathcal{A}^{\alpha} : \ \alpha \to \mathbb{P}x \to (\ell \hookrightarrow \mathbb{P}x) \qquad \text{for } \alpha \in \{E, T, M, D\} & & \\
\hline
\mathcal{A}^{E}[\![T]\!]\Gamma & = \mathcal{A}^{T}[\![T]\!]\Gamma & \\
\mathcal{A}^{E}[\![M]\!]\Gamma & = \mathcal{A}^{M}[\![M]\!]\emptyset & (\dagger) \\
\mathcal{A}^{T}[\![[\lambda x \,.\, E]^{\ell}]\!]\Gamma & = \{\ell \mapsto \Gamma\} \cup (\mathcal{A}^{E}[\![E]\!]\Gamma \setminus \{x\}) & \\
\mathcal{A}^{T}[\![[E_1 \ E_2]^{\ell}]\!]\Gamma & = \{\ell \mapsto \Gamma\} \cup \mathcal{A}^{E}[\![E_1]\!]\Gamma \cup \mathcal{A}^{E}[\![E_2]\!]\Gamma & \\
\mathcal{A}^{T}[\![[E \,.\, x]^{\ell}]\!]\Gamma & = \{\ell \mapsto \Gamma\} \cup \mathcal{A}^{E}[\![E]\!]\Gamma & \\
\mathcal{A}^{M}[\![\{\!\!\{ \, D_i{}^{i \in 1..m} \, \}\!\!\}^{\ell}]\!]\Gamma & = \{\ell \mapsto \Gamma\} \cup \left(\bigcup_{i \in 1..m} \mathcal{A}^{D}[\![D_i]\!](\Gamma \cup \{x_i{}^{i \in 1..m}\}) \right) & \\
 & \qquad \text{where} \quad D_i = x_i = T_i \text{ or } D_i = x_i = M_i & \\
\mathcal{A}^{D}[\![x = T]\!]\Gamma & = \mathcal{A}^{T}[\![T]\!]\Gamma & \\
\mathcal{A}^{D}[\![x = M]\!]\Gamma & = \mathcal{A}^{M}[\![M]\!]\Gamma & (\ddagger)
\end{array}
$$

Fig. 4. Calculation of *module*-bound available variables

Available variables of an expression T are variables introduced by the outer context of T. We are especially interested in variables introduced by a so called uninterrupted *module* hierarchy. The syntax tree of an uninterrupted *module* hierarchy only contains the non-terminals M and D of Fig. 2. This set of variables is called *module-bound available variables*. It is stored in the mapping AVM which is calculated by the family of functions $\mathcal{A}^E$, $\mathcal{A}^T$, $\mathcal{A}^M$, $\mathcal{A}^D$ of Fig. 4 (one for each nonterminal in the grammar of Fig. 2).

The second argument of function $\mathcal{A}^E$ is the set of *module*-bound available variables of the outer context of an expression E. The mapping AVM of a program E is defined as $AVM = \mathcal{A}^E[\![E]\!]\emptyset$, since the outer context is empty.

In equation (†) the set of *module*-bound available variables is cleared because at that point a new hierarchy starts. In contrast, Γ in equation (‡) is not modified. The possibility to define the function $\mathcal{A}^\alpha$ in this simple form was one of the reasons to distinguish between *modules* and other expressions in the syntax.

4.3 Analysis

In Sec. 5, expressions E will be transformed into a new intermediate form with an explicit evaluation order. This transformation needs the dependency order of all expressions. In this section, we will describe the calculation of this ordering

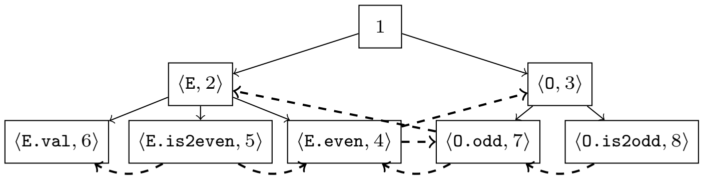


Fig. 5. *Module* tree (without dashed edges) and dependency graph for the example in Listing 1.4

Index	*FN*	*AVM*	*UNM*
4	O.odd	E, O, even, is2even, val	O.odd
5	even,val	E, O, even, is2even, val	even,val
6		E, O, even, is2even, val	
7	E.even	E, O, odd, is2odd	E.even
8	odd	E, O, odd, is2odd	odd

Fig. 6. Mapping label to *module*-bound used names

with the help of the running example. The complete function definitions can be found in [13].

The evaluation order has to be calculated for every uninterrupted *module* hierarchy. The analysis is split into the following three phases:

1. Build *module* tree
2. Calculate the dependency edges
3. Calculate the strongly connected components (e.g. Sharir's algorithm [14])

We use our earlier example of recursive *modules* (see Listing 1.4) to illustrate the analysis.

At the beginning a *module* tree for this uninterrupted *module* hierarchy is built. The tree is shown in Fig. 5. Every node represents a definition. The left-hand side is represented by the complete name within this *module*-hierarchy and the right-hand side by its label. The root of each *module* tree is a special node. It has no name, as the *module* is anonymous. The label is that of the outermost *module*; in our running example it is 1. The children of a node are the definitions of its associated *module*.

In phase two, all dependency edges have to be calculated. We have to consider all leaves. In our example, these are all nodes with labels 4 – 8. First, the set of all free names, where the variable is in the set of *module*-bound available variables must be calculated. This set is called *module-bound used names* (*UNM*). In Fig. 6 all three sets are calculated for each label. In this example, the set of *module*-bound used names is identical to the set of free names. This is true in all cases, as names from the outer context may be used.

For every name in the set *UNM*, the representing node has to be found. The tree is searched upwards from the current node to find the variable that corresponds to the name. For example, we consider the node with label 4 and the *module*-bound used name `O.odd`.

The parent node has no child with the definition `O` so we must look at the next parent. That one has such a child — the node with label 3. Starting with this node, the selector chain is considered. A suitable child must be found for every selector in the chain. In this case, the next selector is `.odd` and the suitable child is node 6. As there are no further selectors, node 6 is the desired node. Therefore, a dependency edge is inserted from 4 to 6. If there is no suitable child, there are two possibilities:

– The current node represents a *module*: this is an error, because the selector will never exist.
– The current node represents another expression: this means the current node is the one we have been searching for (e.g. this is the case for the example in Sec. 3.3[2]).

As all dependency edges are calculated for the *module* tree the recognised static part is equal to the path in the tree.

Additional edges are added from the start node x to all ancestors of the found node y, except for nodes which are also ancestors of x. In this example, an edge from 4 to 3 must be added. These additional edges are necessary to properly handle recursive *modules*, in this case of `E` and `O`. The complete dependency graph is illustrated in Fig. 5.

Sharir's algorithm [14] is used to calculate the strongly connected components (scc) in dependency order for this graph. The result is a list of the sets of nodes. As the top-level *module* is always the last component in the scc list and is never recursive, we can ignore this component. In our example the nodes 2–8, except 6, are in one scc. The result is the following list:

$$\{\langle \texttt{E.val}, 6\rangle\},$$
$$\{\langle \texttt{E}, 2\rangle, \langle \texttt{O}, 3\rangle, \langle \texttt{E.even}, 4\rangle, \langle \texttt{E.is2even}, 5\rangle, \langle \texttt{O.odd}, 7\rangle, \langle \texttt{O.is2odd}, 8\rangle\}$$

The result of the dependency analysis is used in the transformation described in the next Sec. 5.

5 Transformation

In this section we describe the transformation based on the dependency order of Sec. 4.3. Every abstract syntax tree (Fig. 2) is transformed into a tree with explicit dependency order (Fig. 7).

In this syntax, *modules* are represented by a sequence `Seq` of definitions in dependency order. Mutually recursive *modules* are combined in one `Seq` and all their definitions are children of this node. Every definition is either recursive (`RecDef`) or non-recursive (`Def`) or a *module* (`Seq`). All expressions T are represented by S.

[2] The *module* tree for the outer *module* does not contain `a` and therefore the dependency edge for the free name `s.a.b` is the edge from `output` to `s` and so the dependency analysis gives the order `s`, `output`.

$$\begin{aligned} A &::= C \mid S \\ S &::= \lambda\, x\ .\ A \mid x \mid A\ .\ x \mid \ldots \\ C &::= \texttt{Def}\ N\ S \mid \texttt{RecDef}\ (N\ S)^{+} \mid \texttt{Seq}\ C^{*} \end{aligned}$$

Fig. 7. Abstract syntax with explicit dependency order

Only the transformation of *modules* is non-trivial. First, the strongly connected components have to be calculated for the whole *module* as described in the previous section. The result is used to transform the complete *module* hierarchy into the syntax of Fig. 7. For lack of space we omit the concrete transformation function, which can be found in [13].

The result of the transformation for the running example is given in Listing 1.8. The top level group is represented with the outer `Seq`, the mutually recursive *modules* `E` and `O` are combined in the inner `Seq`. In the inner `Seq` all definitions of both *modules* are given in dependency order.

```
Seq (Seq (Def E.val, 2)
         (RecDef (E.even, λn. IF n==0 THEN true ELSE O.odd (n-1))
                 (O.odd, λn. IF n==0 THEN false ELSE E.even (n-1)))
         (Def E.is2even, even val)
         (Def O.is2odd, odd 2))
```

Listing 1.8. Transformed even-odd example.

During the transformation the definitions for mutual recursive *modules* are ordered and so mutual recursion is allowed as long as all mutual recursive definitions are functions (see Sec. 3.2).

We will give a denotational semantics for the result of this transformation in the next section.

6 Denotational Semantics

The result of the transformation is mapped to an interpretation by the evaluation function $\mathcal{E}$. We use the established techniques of denotational semantics [15,16,17] without going into detail.

6.1 Semantic Domain and Auxiliary Functions

We map every expression A to a value V from the following semantic domains:

Values: $\mathsf{V} = \mathsf{N} + \mathsf{S} + \mathsf{B} + \mathsf{H} + \mathsf{F}$

Modules: $\mathsf{H} = x \hookrightarrow \mathsf{V}$ Functions: $\mathsf{F} = \mathsf{V} \to \mathsf{V}$

The set of all values consists of numbers N, strings S, booleans B, *module* values H and function values F. *Module* values H are partial functions with finite domain, mapping variables to values. Although the *module* value can be seen as an evaluation context that maps free variables of an expression to a value, we use a special evaluation context Γ in order to avoid confusion.

$$\mathcal{E}: \; A \rightarrow \Gamma \rightarrow \mathsf{H} \rightarrow \mathsf{V}$$

$$\begin{aligned}
&\mathcal{E}[\![\texttt{Def}\ N\ S]\!]\Gamma\ \mathsf{H} && = N \mapsto\!\!\!> (\mathcal{E}[\![S]\!](\Gamma \lhd (\mathsf{H}|_N))\ \emptyset)\\
&\mathcal{E}[\![\texttt{RecDef}\ (N_i\ S_i)^{i\in 1..n}]\!]\Gamma\ \mathsf{H} && = v_1 + \dots + v_n\\
&&& \text{where} \quad \Phi\,\langle \mathsf{F}_i\rangle^{i\in 1..n} = \langle \mathcal{E}[\![S_i]\!](\Gamma \lhd (\mathsf{H}'|_{N_i}))\ \emptyset\rangle^{i\in 1..n}\\
&&& \qquad\qquad \text{where} \quad v'_i = N_i \mapsto\!\!\!> F_i\\
&&& \qquad\qquad\qquad\qquad \mathsf{H}' = \mathsf{H} + v'_1 + \dots + v'_n\\
&&& \qquad \left\langle \mathsf{F}_i^{\,i\in 1..m} \right\rangle = \mathsf{FIX}\Phi\\
&&& \qquad v_i = N_i \mapsto\!\!\!> F_i\\
&\mathcal{E}[\![\texttt{Seq}\ C_i^{\,i\in 1..n}]\!]\Gamma\ \mathsf{H} && = v_1 + \dots + v_n\\
&&& \text{where} \quad v_i = \mathcal{E}[\![C_i]\!]\Gamma\ (\mathsf{H}_{i-1} + v_{i-1})\\
&&& \qquad\qquad \mathsf{H}_0 = \mathsf{H}\\
&\mathcal{E}[\![E\,.\,x]\!]\Gamma\ \mathsf{H} && = \begin{cases} v, & \text{if } x \multimap v \in (\mathcal{E}[E]\Gamma\ \mathsf{H})\\ \text{ERROR}, & \text{otherwise}\end{cases}\\
&\dots
\end{aligned}$$

Fig. 8. Evaluation function

The environment Γ maps all free variables to a value. The operator $\lhd$ combines two environments Γ_1 and Γ_2 where the right mapping overrides definitions of the left. For syntactical distinction between the mapping within a *module* value and the environment, the environment mapping uses $\mapsto$ and the *module* value $\multimap$. Furthermore, we define three auxiliary functions for *module* values:

– The operation $\mapsto\!\!\!>$ constructs a *module* value for a given name and value.

$$x_1\,.\,\cdots\,.\,x_n \mapsto\!\!\!> v = \begin{cases} x_1 \multimap (x_2\,.\,\cdots\,.\,x_n \mapsto\!\!\!> v), & \text{if } n > 1\\ x_1 \multimap v, & \text{otherwise}\end{cases}$$

– The operator $+$ combines two *module* values.

$$\begin{aligned}\mathsf{H}_1 + \mathsf{H}_2 = &\{x \multimap (v_1 + v_2) \mid x \multimap v_1 \in \mathsf{H}_1 \wedge x \multimap v_2 \in \mathsf{H}_2\}\\ &\cup\{x \multimap v_1 \mid x \multimap v_1 \in \mathsf{H}_1 \wedge x \multimap v_2 \notin \mathsf{H}_2\}\\ &\cup\{x \multimap v_2 \mid x \multimap v_2 \in \mathsf{H}_2 \wedge x \multimap v_1 \notin \mathsf{H}_1\}\end{aligned}$$

– The projection $|$ creates the environment Γ for the given Name.

$$\mathsf{H}|_{x_1\,.\,\cdots\,.\,x_n} = \begin{cases} \Gamma' \lhd v|_{x_2\,.\,\cdots\,.\,x_n}, & \text{if } x_1 \multimap v \in \mathsf{H}\\ \Gamma' & \text{if } n \geq 1\\ \emptyset, & \text{otherwise}\end{cases}$$
$$\text{where} \quad \Gamma' = \{x \mapsto v \mid x \multimap v \in \mathsf{H}\}$$

6.2 Evaluation Function

Figure 8 shows the evaluation function for *modules* and definitions. The evaluation function has three arguments: the expression to evaluate, a normal environment mapping variables to values and a *module* value H representing the current *module* hierarchy. All definitions are evaluated to a *module* value representing the path within the current *module* hierarchy. The right-hand side of each definition is evaluated within a fresh (empty) hierarchy and an environment where the current scope is visible. Therefore the current environment is enriched by the projection of the current hierarchy into the scope of the definition. The projection adds all definitions of the current hierarchy that are visible for this definition. Recursive definitions are evaluated, as usual, via a fixpoint operator.

Modules are evaluated to a *module* value containing all inner definitions. The definitions are evaluated in evaluation order, i.e. in the order of the given list. The values of the children are added to the hierarchy value and the next child is evaluated. The result is the combination of all child values.

The result of a selection $E.x$ is either the value v, if x maps to v in the result of the evaluation of E or it is an error. There are two possible errors: the value of E is not a *module* or it does not contain an x.

The rest of the evaluation function is omitted as it is the usual definition with an environment and the hierarchy value is ignored.

7 Related Work

There are various approaches for flexible modularisation in statically and dynamically typed functional languages. We focus here on those combining nested mutually recursive modules and some kind of abstraction for modules — either special functors or first-class modules.

In [2] a module system for Haskell is proposed, where records and modules are joined in one concept. These record-modules are, as in our approach, first-class values with dot notation. As Haskell is typed, these records are typed as well. The type of every record must be given and provides information about all defined names of the module. Allowed selections are detectable by type inference for every expression. As Haskell is lazy it is not necessary to calculate the dependency order.

In [10] a calculus is proposed for first-class modules that are allowed to be mutually recursive. Their approach unifies classes and objects, so this construct equates to our *modules*. In contrast, an undecidable type system is proposed and no explicit semantics is given.

SML provides a special module language. The flexibility of ML modules is given by module functions. Using these functions, one can create new modules and change modules. Mutually recursive modules are not natively supported, but some extensions do allow it.

One of the first approaches for recursive modules in ML was mixin modules [18]. Mixins are, as all ML modules, not first-class values. The dependencies to other modules must be declared within the module. Missing definitions are assigned by gluing modules over the function *sum*. This approach differs from ours, as all dependencies must be given and so the dependency order is explicit.

In [19] Owens and Flatt describe the concept of "units". Units are first-class nested modules and it is possible to define recursive units by a compound similar to mixins. When two or more mutual recursive units are combined a new unit is created containing all definitions of the combined units in the given order and in this order the units are evaluated. This suits well for mutual recursive functions spread over module borders, but it does not allow all kinds of mutual recursive modules. For instance, our even-odd example is not possible, since both modules contain a definition that is not a function. In our approach these inner definitions are sorted for a proper evaluation during the transformation.

In [8] Russo extends Mini-SML by recursive modules. As for all ML-languages, the evaluation order must be given by the programmer. All forward referenced structures must be introduced at the beginning of the recursive structure. During evaluation, those forward references are assumed to be undefined. After evaluation, they are updated in the heap. In this approach an exception is raised if the forward reference is used. Therefore, all occurrences of forward references must be under abstraction. Our running example is not possible in this approach. Although it is type correct, the evaluation would raise an exception, as the tests `is2even` and `is2odd` are not abstractions.

Garrigue and Nakata study in [7] applicative modules with polymorphic functors and recursion based on paths. They use path resolution to find and handle recursion. As they have proven in [9], path resolution is not decidable for flexible systems such as ours.

In his PhD thesis [4], Claus Reinke developed a module system for functional call-by-value languages. His modules are called frames and are quite similar to our *modules*. In particular, they are dynamically typed first-class values. Frames may contain a set of definitions and these definitions may be mutually recursive. Recursion over frame boundaries is not allowed and all names of other frames must be explicitly imported. These imports give the evaluation order for frames. Mutually recursive modules can only be implemented via functions where the relevant modules are parameters.

8 Conclusion and Future Work

The introduced concept of *modules* is a very flexible, expressive and homogeneous mechanism for dynamic data structures, bindings and modularisation. Using *modules* one can define hierarchical, mutually recursive first-class modules. Using a special dependency analysis a transformation based call-by-value semantics is given.

Finally, we have already extended the introduced language ModLang by *module* morphisms. They allow *modules* to not only be created at runtime, but also to be extended or restricted. This improves the flexibility and reuseability of *modules*. Furthermore, we have a mechanism for imports based on a control flow analysis and the ability to control visibility when using imports and exports.

At the moment, we are working on a translation into an untyped functional language with "let", "letrec" and dynamic records. In addition, we are developing a concept for separate compilation. Furthermore, we will study correctness properties of our algorithms.

Acknowledgement. We would like to thank Peter Pepper and Christoph Höger for helpful discussions and comments about language design and implementation. We are very grateful to Doug Smith of Kestrel Institute, Palo Alto, where most of the presented work was done. Furthermore, we would like to thank the TFP reviewers for the helpful review and Joe Harjung for proof reading and his comments.

References

1. Pepper, P., Hofstedt, P.: Funktionale Programmierung – Sprachdesign und Programmiertechnik. Springer (2006)
2. Peyton Jones, S.L., Shields, M.B.: First class modules for Haskell. In: 9th International Conference on Foundations of Object-Oriented Languages (FOOL 9), Portland, Oregon, pp. 28–40 (January 2002)
3. Russo, C.V.: First-class structures for standard ml. Nordic J. of Computing 7, 348–374 (2000)
4. Reinke, C.: Functions, Frames, and Interactions – completing a lambda-calculus-based purely functional language with respect to programming-in-the-large and interactions with runtime environments. PhD thesis, Universität Kiel (1997)
5. Blume, M.: Hierarchical Modularity and Intermodule Optimization. PhD thesis, Princeton University (November 1997)
6. Crary, K., Harper, R., Puri, S.: What is a recursive module? SIGPLAN Not. 34(5), 50–63 (1999)
7. Nakata, K., Garrigue, J.: Recursive modules for programming. SIGPLAN Not. 41, 74–86 (2006)
8. Russo, C.V.: Recursive structures for standard ml. SIGPLAN Not. 36(10), 50–61 (2001)
9. Nakata, K., Garrigue, J.: Path resolution for nested recursive modules. Higher-Order and Symbolic Computation, 1–31 (May 2012)
10. Odersky, M., Cremet, V., Röckl, C., Zenger, M.: A nominal theory of objects with dependent types. In: Cardelli, L. (ed.) ECOOP 2003. LNCS, vol. 2743, pp. 201–224. Springer, Heidelberg (2003)
11. Cardelli, L., Donahue, J., Glassman, L., Jordan, M., Kalsow, B., Nelson, G.: Modula-3 report (revised). ACM SIGPLAN Notices 27(8), 15–42 (1992)
12. Abelson, H., Sussman, G.J., Sussman, J.: Structure and Interpretation of Computer Programs, 2nd edn. MIT Press (1996)
13. Lorenzen, F., Rohloff, J.: Gruppen: Ein Ansatz zur Vereinheitlichung von Namensbindung und Modularisierung in strikten funktionalen Programmiersprachen (Langfassung). Technical Report 2011-12, TU Berlin (2011)
14. Sharir, M.: A strong-connectivity algorithm and its applications in data flow analysis. Computers & Mathematics with Applications 7(1), 67–72 (1981)
15. Tennent, R.D.: The denotational semantics of programming languages. Commun. ACM 19(8), 437–453 (1976)
16. Gordon, M.J.C.: The Denotational Description of Programming Languages. Springer (1979)
17. Stoy, J.E.: Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. Computer Science Series. MIT Press (1981)
18. Hirschowitz, T., Leroy, X.: Mixin modules in a call-by-value setting. ACM Trans. Program. Lang. Syst. 27, 857–881 (2005)
19. Owens, S., Flatt, M.: From structures and functors to modules and units. SIGPLAN Not. 41(9), 87–98 (2006)