Recursive Object-Oriented Modules

Keiko Nakata[1] Akira Ito[2] Jacques Garrigue[1]

[1] Kyoto University Research Institute for Mathematical Sciences
[2] Hitachi, Ltd.

keiko@kurims.kyoto-u.ac.jp, garrigue@math.nagoya-u.ac.jp

Abstract

ML-style modules and classes are complementary. The former are better at structuring and genericity, the latter at extension and mutual recursion. We investigate the convergence of both mechanisms by designing an object-oriented calculus based on a nominal module system with mutual recursion. Our modules assume simultaneously the roles of classes with subtyping, nested structures with type members, and simple functors. Flexible inter-module recursion is obtained by allowing free references not constrained by the order of definitions. We closely examine the well-foundedness of the recursion, in the presence of nesting and functors. The presented type system is provably decidable, and ensures the well-foundedness. We also define a call-by-value semantics, for which type soundness is proved.

1. Introduction

ML-style modules offer excellent support for structuring and genericity [7]. The nested structure of modules plays a significant role in the decomposition of large programs. Functors can express advanced forms of parametricity, and abstraction can be controlled by signatures with transparent, opaque, or translucent types [10]. However, they are weak at extension and do not allow mutual recursion.

Classes offer better support for extension and mutual recursion. Inheritance and overriding allow one to build a new class only with extensions and changes to an existing one. With subtyping, the new class can be used in place of the previous one. Mutual recursion is in classes' nature, and thereby recursion at both of value and type level has to be supported.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

FOOL 2005 15 January 2005, Long Beach, California
Copyright © 2005 ACM … $5.00

Much work has been devoted to investigating how to combine both mechanisms [12, 1, 8, 14].

Objective Caml is one example of orthogonal combination. The language is very expressive, but the result is quite complex. Many concepts, such as structures, functors, signatures, classes and class interfaces, are introduced in a single language. Despite some features of modules and classes overlap, they are not merged, and use different syntax. This makes it mind-boggling to use both mechanisms simultaneously.

Recently, to get rid of the inconvenience of the former approach, and to give a theoretical foundation which harmonizes both mechanisms, much effort has been made investigating their convergence. When designing such a language, one has to give careful consideration to matters concerning decidability and well-foundedness.

- As investigated in [12], dependent types play an important role in unifying modules and classes. However, as seen in [13], the combination of subtyping and dependent types makes it a hard task to keep decidability of type checking.
- The unification brings about mutual recursion into modules. We have to be careful about the well-foundedness of the recursion, as recursion might cause circular dependencies between modules [3, 8, 5].

Our ultimate goal is to design a language which unifies modules and classes, and equip it with a sound and decidable type system, which ensures well-foundedness of the recursion. In this paper, as a first step towards that goal, we propose a calculus, called *Room*, based on a nominal module system with mutual recursion. In *Room*, modules assume simultaneously the roles of classes, nested structures, and simple functors. The characteristics of our modules are summarized as follows.

Class role: Objects are created from modules, and modules themselves are types of objects as with Java's classes. Mutual recursion between modules is allowed. Inheritance (with method overriding) and subtyping are provided through an asymetric merging operator.

Structure role: Modules can be nested and have type members.

Functor role: Modules can be parameterized.

To make the type system decidable, we put a restriction on functor arguments that requires them to be unparameterized (hence, *Room* does not have higher-order functors), and to have exactly the same inner modules as are prescribed by the functor types. Although this restriction costs our modules some forms of parametricity compared to ML-functors, we still have enough parametricity at class level; *i.e.* classes parameterized over types and superclasses can be expressed.

In *Room*, mutual recursion is offered by *path*s. Paths are our referencing mechanism. They allow one to refer to any module at any level of nesting, upwards or downwards, notwithstanding the order of the definitions. Moreover, simple cases of functor application are allowed in paths, where the functor and its arguments themselves are paths. Paths give one a high degree of freedom for reference, however (or perhaps for this reason), we are required to pay extensive consideration to the well-foundedness of the recursion.

As a module can be defined as an alias of another module using a path, it might happen that module definitions end up being circularly dependent. It is highly desirable to statically reject such ill-founded modules in order to ensure proper module elaboration, *i.e.* to produce a record of the methods defined in a module. The existing approaches on well-founded recursion for recursive modules [3, 8, 5] do not suit our situation, as their strict restriction on the order of access to module components hinders mutual recursion as seen in object-oriented programming. In this paper, we propose an innovative approach, by considering topological sortability of modules. The restriction on functor arguments enables its static inspection. As a consequence, our type system, once made decidable, ensures the absence of ill-founded modules. The type system is also shown to be sound for a call-by-value semantics.

The rest of this paper is organized as follows. In Section 2, we overview the design of *Room*. Section 3 formally defines *Room*. Section 4 discusses the well-foundedness of module systems. Section 5 presents the notion of normalization of types. We give a decidable type system in Section 6. Dynamic semantics and the type soundness result are in Section 7. In Section 8, we review related work. Section 9 concludes.

2. Overview

In this section, we overview the design of *Room*. We use examples motivated by the Subject/Observer pattern. This is a programming pattern seen in class-based programming. It consists of an observed class, called the subject class, and classes which observe that class, called observer classes, and presents a control flow which ensures that changes in the subject are properly reported to observers. This pattern is often used in practice. For instance, when building an editor, a Data object is observed by Monitor objects, and changes in the Data object are reported to Monitor objects in order to reflect the changes on the screen.

```
SubObP = λV<:Value.{
    Subject = {
        V value,
        up.Observer obsr,
        void notifyObserver(){ obsr.update();},
        void setValue(V v){
            value = v;
            notifyObserver();,
        },
        V getValue(){return value;},
    }
    Observer = {
        up.Subject sub,
        up.Subject getSubject(){return sub;},
        abstract void update(),
    }
}
```

Figure 1. Subject/Observer pattern

We begin by showing the module SubObP in Figure 1, which expresses the pattern. SubObP packs 2 modules, Subject and Observer, and is parameterized by a module variable V, which expresses the type of data handled by Subject. Moreover, the method *update* contained in Observer is left to be implemented. As in Java, we can declare abstract methods by giving only their types.

To interact mutually, Subject and Observer refer to each other, through variables *obsr* of type Observer, and *sub* of type Subject respectively. We use *path*s to refer to modules. For example, a path SubObP(M).Observer refers to the module Observer contained in the module obtained by applying SubObP to M. Using absolute access paths starting from the top-most module, we can locate any modules at any level of nesting. Relative access paths are also available. Here, in the example, up is used to specify the enclosing context.

As seen in this example, types are paths or module variables. To simplify examples, we enrich the core types with void and int.

One can build his own application from SubObP by instantiating V with his own data type, and implementing *update*.

V is instantiated by application. The interface of V, namely the module Value, requires actual values corresponding to V to be subtypes of Value.

Let Value and MyValue' be defined as follows.

```
Value = {}
MyValue' = {
    int data,
    int getData(){return data;}
}
```

We build the module MyValue by merging together the 2 modules.

$$\text{MyValue} = \text{MyValue'} \rightharpoonup \text{Value}$$

Merging is a counterpart to inheritance in class systems. It also induces subtype relations between modules, as inheritance does in Java. Here, MyValue is a subtype of MyValue' and Value.

Then, we apply SubObP to MyValue, which yields MySubject as follows.

$$\text{MySubject} = \text{SubObP(MyValue).Subject}$$

MySubject is the module Subject contained in the module obtained by applying SubObP to MyValue. SubObP(MyValue).Subject is also the type of objects created from it with the `new` operator. Since our type system is nominal, and such parametric types are invariant, if we apply SubObP to another module, say MaValeur, the resulting path expresses another type, *i.e.* SubObP(MaValeur).Subject is not equivalent, nor a subtype, nor a supertype of SubObP(MyValue).Subject, unless MaValeur was defined as an alias for MyValue. As such, we provide some form of type generativity.

Next, we would like to build MyObsr, which acts as the observer of MySubject. Consider the following module.

```
MyObsr' = {
    void update(){
        MyValue value = getSubject().getValue();
        int i = value.getData();
        ...
    },
    abstract MySubject getSubject(),
}
```

MyObsr is created by merging together MyObsr'and SubOb(MyValue).Observer.

$$\text{MyObsr} = \text{MyObsr'} \rightharpoonup \text{SubOb(MyValue).Observer}$$

MyObsr is a module, which has methods *update* from MyObsr' and *getSubject* from Observer. The abstract methods in each module are given implementations by each other's identically named methods.

Finally, we get our own customized subject, MySubject, and observer, MyObsr.

Note that our dependent type system can infer that the type of the return value of *getValue* contained in MySubject is MyValue. Hence, *update* of MyObsr can invoke *getDate* from the value returned by MySubject's *getValue* without requiring the method to be specified in advance in V's interface.

We have seen that by combining both mechanisms of ML-modules and classes, the Subject/Observer pattern can be naturally expressed, offering proper extensibility. The unification of the two mechanisms eases their simultaneous use. Moreover, our type system does not require one to explicit the types of recursive modules, whereas this is often a cause of difficulty in existing languages featuring recursive modules.

```
Observer = λC<:ObserverType.{
    ObsrImp = {
        Subject sub,
        void set(Subject s){sub = s; update();...},
        Subject getSubject(){return sub;},
        abstract void update(),
    }
    ObsrMixin = C coerce {getSubject, update}
    Obsr = here.ObsrMixin ⇀ here.ObsrImp
    void main(){
        here.Obsr obsr = new here.Obsr
        ...,
    }
}
ObserverType = {
    required void update
    abstract Subject getSubject,
}
```

Figure 2. Mixin-style programming

Next we will show that the combination also enables easy mixin-style programming.

Observer given in Figure 2 is a module parameterized over the implementation of *update* through the module variable C. Inside ObsrImpl, 2 methods *set* and *getSubject* are defined. By declaring *update* as an abstract method, we can call this method insides bodies of other methods, as we do in *set*. We use `here` in paths to specify the current context.

The interface of the module variable C, namely ObserverType, mentions the *update* method as `required`. As C is a concrete variable, this means that *Observer* may only be applied to modules providing an implementation for *update*. Additionally, the concreteness requires that they do not have any abstract method other than *getSubject*.

We have 2 kinds of module variables, virtual module variables and concrete module variables, to support flexible parameterization. Conceptually, the former are used to parameterize over types as we did in the first example, the latter over implementations as we do here.

The implementation of *update* is instantiated by applying Observer. We coerce C to *getSubject, update* before merging it with ObsrImpl. The coercion operator `coerce` offers a means of access control. ObsrMixin is a module having the same methods as C, but only *getSubject* and *update* are accessible. This coercion is useful, as it avoids unexpected interference even if, for example, C too had a method named *set*.

An object is created in *main* from Obsr with the `new` operator. The restriction imposed by ObserverType on actual

values corresponding to C, ensures that Obsr has no abstract methods.

3. Syntax

The syntax for *Room* is given in Figure 3. M, m, and x are metavariables which range over module names, method names, and variable names, respectively. *Names* is the set of module names. The special variable `this` is assumed to be included in the set of variables. We write $\overline{M}$ or $[M_i]_{i=1}^n$ as a shorthand for $M_1, \cdots, M_n$ $(n \geq 0)$; $\overline{M} = \overline{E}$ or $[M_i = E_i]_{i=1}^n$ for $M_1 = E_1, \cdots, M_n = E_n$; $\lambda\overline{X}{<:}\overline{p}.E$ or $\lambda[X_i{<:}p_i]_{i=1}^n.E$ for $\lambda X_1{<:}p_1.\lambda X_2{<:}p_2.\cdots.\lambda X_n{<:}p_n.E$; $p.M(\overline{q})$ or $p.M([q_i]_{i=1}^n)$ for $p.M(q_1)\cdots(q_n)$.

A module system S is a record of module definitions, method definitions, and method declarations. Modules are defined by module expressions, which are one of *path*, *basic module*, *coercion*, or *merging*.

A path p is a reference to a module, which is obtained by combination of dot notation (access to a module component) and functor application. We use syntactic sugar `here` and `up` to abbreviate respectively the current and the enclosing context, as in Figures 1 and 2. In the module pointed to by $p.M(\overline{p'})$, a path `here`$.q$ (resp. `up`$.q$) is a shorthand for $p.M(\overline{p'}).q$ (resp. $p.q$). A path prefixed with a sequence of `up`'s, such as `up.up`$.M$, can be defined similarly. We usually omit the leading "ϵ." when writing paths.

A *basic module* is a record of module definitions, method definitions, and method declarations. It can be parameterized by module variables, constrained by their interfaces. Interfaces are paths, and denote upper type bounds of modules to which the parameterized modules are to be applied.

Coercion allows visibility control. Programmers may create a new module by hiding some components of an existing one.

Merging is used to define a module by merging together two existing modules. For methods implemented in both modules, the resulting module contains the implementation from the left-hand side of the operator $\rightharpoonup$, *i.e.* the left-hand side overrides the right-hand side.

We have two types of module variables, namely *virtual module variable*s V and *concrete module variable*s C. A virtual module variable may only be used as a type, which is either a path or a module variable, while concrete module variables may freely be used in paths. For instance, one may not create a new object from a virtual variable, but this is allowed with concrete variables as we did in Figure 2. Conceptually, they respectively provide parameterization over types and implementations.

Methods are either defined or declared. We have two qualifiers for method declarations, `abstract` and `required`. Using `required` in interfaces, we can express implementation requirements on parameters, as we did in Figure 2.

Program expressions are either variables, method calls, or object creations.

S	$::=$	$\{\overline{M} = \overline{E}, \overline{met}\}$	*module system*
E	$::=$	p	*path*
	$\mid$	$\lambda\overline{X} <: \overline{p}.\{\overline{M} = \overline{E}, \overline{met}\}$	*basic module*
	$\mid$	p `coerce` $\{\overline{M}, \overline{m}\}$	*coercion*
	$\mid$	$p \rightharpoonup p$	*merging*
p, q, r	$::=$	$\epsilon \mid C \mid p.M \mid p(p) \mid p(V)$	*path*
X	$::=$	$C \mid V$	*module var.*
τ	$::=$	$p \mid V$	*type*
met	$::=$	$\tau\ m(\tau\ x)\{e\}$	*met. definition*
	$\mid$	`abstract` $\tau\ m(\tau\ x)$	*abstract met.*
	$\mid$	`required` $\tau\ m(\tau\ x)$	*required met.*
e	$::=$	$x \mid e.m(e) \mid$ `new` p	*program expr.*
P	$::=$	(S, e)	*program*

Figure 3. Syntax for *Room*

A program is a pair of a module system and a program expression.

Any module system is assumed to satisfy the following three conditions: 1) all module variables are bound, where the definition of bound variables is as usual; 2) all bound variables differ from each other; 3) all basic modules contain no duplicate method declarations and definitions for any method name, no duplicate module definitions for any module name.

For simplicity, we leave out several features, which would be important to build a practical language from *Room*, like static methods, instance and class variables, the "super"operator, constructors and others.

4. Well-founded module system

Paths give one a high degree of freedom for references, with absolute or relative access, allowing functor application in it. We can naturally express mutual recursion with them, in the presence of functors and nesting. However, we have to make sure that module systems are properly defined.

As a module itself may be defined as an alias or a composition of other modules using paths, it might happen that module definitions end up being mutually dependent. For example, consider the following module system,

$$\begin{array}{l}\{\mathrm{M}_1 = \mathrm{M}_2 \rightharpoonup \mathrm{M}_3,\\ \ \mathrm{M}_2 = \mathrm{M}_1 \rightharpoonup \mathrm{M}_4\}\end{array}$$

which is clearly ill-founded.

It is highly desirable to statically reject such ill-founded module systems while accepting mutual recursion in general. The question is how to define decidable "well-foundedness" in our situation. On the one hand, we would like to access to components of partially evaluated modules, *i.e* access to components of a module should be allowed before the evaluation of some other components of the module is yet completed. This is necessary to support mutual recursion as seen in object-oriented programming. On the other hand, we would like to statically reject circular dependencies be-

$$
\begin{aligned}
&dp(p, \lambda X{<:}q.E) = dp(p,q) \cup dp(p,E) \cup dp(X,q)\\
&dp(p, \{[M_i = E_i]_{i=1}^n, \overline{met}\}) = \textstyle\bigcup_{1\le i\le n} dp(p.M_i, E_i)\\
&dp(p, q_1 \rightharpoonup q_2) = dp(p,q_1) \cup dp(p,q_2)\\
&dp(p, q\ \texttt{coerce}\ \{\overline{M}, \overline{m}\}) = dp(p,q)\\
&dp(p,q) = \{(p,r) \mid r \in \mathit{flats}(q)\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{flats}(p) = \mathit{flat}(p) \cup \textstyle\bigcup_{q\in \mathit{args}(p)} \mathit{flats}(q)\\
&\mathit{flat}(\epsilon) = \epsilon\\
&\mathit{flat}(X) = X\\
&\mathit{flat}(p.M) = \mathit{flat}(p).M\\
&\mathit{flat}(p(V)) = \mathit{flat}(p)\\
&\mathit{flat}(p(q)) = \mathit{flat}(p)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{args}(\epsilon) = \emptyset\\
&\mathit{args}(X) = \emptyset\\
&\mathit{args}(p.M) = \mathit{args}(p)\\
&\mathit{args}(p(q)) = \{q\} \cup \mathit{args}(p)\\
&\mathit{args}(p(V)) = \{V\} \cup \mathit{args}(p)
\end{aligned}
$$

Figure 4. Extraction of the base relation

tween modules in order to ensure proper module elaboration, *i.e.* to produce a record of the methods defined in a module.

Our definition of well-foundedness for module systems is based on the well-foundedness of a relation approximating dependencies between modules. This ensures that modules can be sorted topologically. For example, while the above example is unsortable as M_1 and M_2 are circularly dependent, the following example is sortable,

$$
\begin{aligned}
&\{\mathrm{M}_1 = \{\mathrm{M}_{11} = \mathrm{M}_2.\mathrm{M}_{22},\ \mathrm{M}_{12} = \{\ldots\}\},\\
&\ \mathrm{M}_2 = \{\mathrm{M}_{21} = \mathrm{M}_1.\mathrm{M}_{12},\ \mathrm{M}_{22} = \{\ldots\}\}\}
\end{aligned}
$$

as we only have $\mathrm{M}_1.\mathrm{M}_{11}$ depending on $\mathrm{M}_2.\mathrm{M}_{22}$ and $\mathrm{M}_2.\mathrm{M}_{21}$ depending on $\mathrm{M}_1.\mathrm{M}_{12}$, which is not circular. Moreover, we only consider dependencies at the value level. For example, in Figure 1, Subject does not depend on Observer as Observer is used only at the type level in Subject.

In the rest of this section, we formally define the (approximated) dependency relation.

Dependency relation

Our approach is to extract a *dependency relation* from a module system S, then check whether the relation is well-founded or not.

Let S be a module system, the dependency relation of S is a binary relation on flat paths, where a flat path is a path containing no application. The construction of the dependency relation takes two steps: 1) extract a base relation from S; 2) expand the base relation in order to take into account the dependencies that do not explicitly appear in S.

The base relation of S is extracted by the function *dp* given in Figure 4. Given a flat path p and a module expression E, *dp* calculates dependencies assuming that p depends on E. When E is of form $\lambda X{<:}q.E$, it recursively calculates dependencies assuming that p depends on q and E, and X on q. When E is of form $\{[M_i = E_i]_{i=1}^n, \overline{met}\}$, $p.M_i$ depends on E_i. Note that, instead of regarding p as depending on E_i, it employs more precise dependencies. Although this make the dependencies more complex, it gives more freedom for recursion between modules. Coercion and merging are straightforward. Finally, if E is a path q, *dp* approximates functor applications in q by making p depend on all flat paths appearing in q. The function *flats* returns the set of flat paths appearing in a path. For example, $\mathit{flats}(\mathrm{M}_1.\mathrm{M}_2(\mathrm{M}_3(\mathrm{M}_4.\mathrm{M}_5)).\mathrm{M}_6) = \{\mathrm{M}_1.\mathrm{M}_2.\mathrm{M}_6,\ \mathrm{M}_3,\ \mathrm{M}_4.\mathrm{M}_5\}$. It should be pointed out that object creation using path references does not entail dependencies. Our "well-foundedness" concerns recursion at module level, not at object level.

The base relation of S is defined as $dp(\epsilon, S)$. Then the dependency relation of S is defined as the *postfix and transitive closure* of the base relation.

Definition 1 *Let D be a binary relation on flat paths. The postfix and transitive closure of D, denoted as $\tilde{D}$, is the smallest transitive relation which contains D and meets the following condition: if (p,q) is in $\tilde{D}$ and M in Names, then $(p.M, q.M)$ is also in $\tilde{D}$.*

We call postfix closure of D the smallest relation that satisfies only the second condition.

Example 1 *Consider the following module system* S*,*

$$
\begin{aligned}
&\{\mathrm{M}_1 = \{\mathrm{M}_{11} = \{\cdots\}, \mathrm{M}_{12} = \texttt{here}.\mathrm{M}_{13}.\mathrm{N}, \mathrm{M}_{13} = \mathrm{M}_2.\mathrm{M}_{21}\}\\
&\ \mathrm{M}_2 = \{\mathrm{M}_{21} = \{\mathrm{N} = \{\cdots\}, \cdots\}, \mathrm{M}_{22} = \mathrm{M}_1.\mathrm{M}_{11}\}\}
\end{aligned}
$$

The base relation of S *is:*

$$
\begin{aligned}
&\{(\mathrm{M}_1.\mathrm{M}_{12},\ \mathrm{M}_1.\mathrm{M}_{13}.\mathrm{N}), (\mathrm{M}_1.\mathrm{M}_{13},\ \mathrm{M}_2.\mathrm{M}_{21}),\\
&\ (\mathrm{M}_2.\mathrm{M}_{22},\ \mathrm{M}_1.\mathrm{M}_{11})\}.
\end{aligned}
$$

Then the dependency relation is the postfix closure of the following set:

$$
\begin{aligned}
&\{(\mathrm{M}_1.\mathrm{M}_{12},\ \mathrm{M}_1.\mathrm{M}_{13}.\mathrm{N}), (\mathrm{M}_1.\mathrm{M}_{13},\ \mathrm{M}_2.\mathrm{M}_{21}),\\
&\ (\mathrm{M}_2.\mathrm{M}_{22},\ \mathrm{M}_1.\mathrm{M}_{11}), (\mathrm{M}_1.\mathrm{M}_{13}.\mathrm{N},\ \mathrm{M}_2.\mathrm{M}_{21}.\mathrm{N}),\\
&\ (\mathrm{M}_1.\mathrm{M}_{12},\ \mathrm{M}_2.\mathrm{M}_{21}.\mathrm{N})\}.
\end{aligned}
$$

Definition 2 *Let D be a binary relation on flat paths. D is well-founded if and only if D does not contain an infinite descending sequence,* i.e. *there does not exists an infinite sequence $\{p_i\}_{i=1}^{\infty}$ such that, for all i in $[1,\infty)$, (p_i, p_{i+1}) is in D.*

Definition 3 *A module system S is well-founded if and only if the postfix and transitive closure of $dp(\epsilon,S)$ is well-founded. Moreover, we say a program (S,e) is well-founded if and only if S is well-founded.*

Proposition 1 *It is decidable whether a module system S is well-founded or not.*

[N-ROOT]
$nlz(\epsilon,\ \epsilon).$

[N-VAR]
$nlz(X,\ X).$

[N-APP]
$nlz(p(q),\ p'(q'))$
:- $nlz(p,\ p'), nlz(q,\ q').$

[N-ExPV]
$nlz(p.M,\ p'.M)$
:- $nlz(p,\ p'),$
$src(p'.M,\ E),$
$\neg(E \equiv q).$

[N-PV]
$nlz(p.M,\ q)$
:- $nlz(p,\ p'),$
$src(p'.M,\ r),$
$subst(p',\ \theta),$
$nlz(\theta(r),\ q).$

[N-CRC]
$nlz(p.M,\ q)$
:- $nlz(p,\ p'.N),$
$src(p'.N,\ r\ \texttt{coerce}\ \{\overline{M}, \overline{m}\}),$
$M \in \{\overline{M}\},$
$subst(p',\ \theta),$
$nlz(\theta(r).M,\ q).$

[N-MRG1]
$nlz(p.M,\ q)$
:- $nlz(p,\ p'.N),$
$src(p'.N,\ r \rightharpoonup r'),$
$subst(p',\ \theta),$
$nlz(\theta(r).M,\ q).$

[N-MRG2]
$nlz(p.M,\ q)$
:- $nlz(p,\ p'.N),$
$src(p'.N,\ r \rightharpoonup r'),$
$subst(p',\ \theta),$
$nlz(\theta(r').M,\ q).$

[N-INF]
$nlz(p.M,\ q)$
:- $nlz(p,\ C),$
$nlz(\Delta(C).M,\ q).$

Figure 5. Normalization of paths

$src(\epsilon,\ S).$
$src(p.M([M_i]_{i=1}^n).N,\ E)$ **:-** $src(p.M,\ \lambda[X_i{:}q_i]_{i=1}^n.\{\cdots, N = E, \cdots\}).$

$subst(\epsilon,\ \mathrm{id}).$
$subst(p.M,\ \theta)$ **:-** $subst(p,\ \theta),\ \ src(p.M,\ \lambda\overline{X}{:}\overline{q}.\{\overline{met}, \overline{D}\}).$
$subst(p(q),\ \theta[X : q])$ **:-** $params(p,\ X :: L),\ subst(p,\ \theta).$

$params(p.M,\ \overline{X})$ **:-** $src(p.M,\ \lambda\overline{X}{:}\overline{q}.\{\overline{met}, \overline{D}\}).$
$params(p.M,\ [])$ **:-** $src(p.M,\ q\ \texttt{coerce}\ \{\overline{m}, \overline{M}\}).$
$params(p.M,\ [])$ **:-** $src(p.M,\ q_1 \rightharpoonup q_2).$
$params(p(q),\ L)$ **:-** $params(p,\ X :: L).$
$params(X,\ []).$

Figure 6. Source predicates

In the following sections, we fix a well-founded program (S, e).

5. Normalization of types

Types are paths or module variables. We judge the equivalence of types by the equality of the modules they refer to. For example, consider the following module system S_1,

$$\begin{array}{rcl} \{\mathrm{M}_1 & = & \{\mathrm{M}_{11} = \{\mathrm{N} = \{\cdots\}\}\}, \\ \mathrm{M}_2 & = & \{\cdots\}, \\ \mathrm{M}_3 & = & \lambda\mathrm{C}{<:}\mathrm{M}_1.\{\mathrm{M}_{31} = \mathrm{C}.\mathrm{M}_{11}\}, \\ \mathrm{M}_4 & = & \mathrm{M}_2 \rightharpoonup \mathrm{M}_1\} \end{array}$$

$\mathrm{M}_1.\mathrm{M}_{11}$, $\mathrm{M}_4.\mathrm{M}_{11}$ and $\mathrm{M}_3(\mathrm{M}_4).\mathrm{M}_{31}$ are equivalent types as they all refer to the module M_{11} contained in module M_1.

In this section, we introduce the notion of normalization of types. We formally define the equivalence of types by the equality of their normal forms.

Normalization is defined using the predicate *nlz* given in Figure 5, and auxiliary predicates in Figure 6. Δ is the finite mapping, which maps module variables to their interfaces. For example, $\Delta(\mathrm{C}) = \mathrm{M}_1$ holds in the above module system S_1. All variables of the module system are assumed to have different names.

We use Horn clauses in Prolog-like syntax to define our predicates and their inference rules. The clause A**:-**$B, C.$ is read as "if B and C hold, then A holds". Another possible notation would be $\frac{B \quad C}{A}$, but we prefer the first one in most cases, as it is more versatile and lets us use explicit names for predicates.

We give a brief account of the predicates in Figure 6. If p is in normal form other than module variables, the module definition of p is looked up in the module system S with the predicate src. For example, $src(\mathrm{M}_1.\mathrm{M}_{11},\ \{\mathrm{N} = \{\cdots\}\})$ holds in S_1, meaning that the module referred to by $\mathrm{M}_1.\mathrm{M}_{11}$ is defined by the module expression $\{\mathrm{N} = \{\cdots\}\}$[1]. Substitutions of types for module variables are constructed from normal forms with the predicate *subst*, where id is the identity substitution. The metavariable θ ranges over the substitutions. When $subst(p,\ \theta)$ holds, we call θ the substitution extracted from p. The predicate *params* denotes the formal parameters of the module referred to by p. For example, $subst(\mathrm{M}_3(\mathrm{M}_1),\ [C \mapsto \mathrm{M}_1])$ and $params(\mathrm{M}_3,\ \mathrm{C})$ hold in S_1.

Definition 4 *A type q is a normal form of a type p if $nlz(p,\ q)$ holds.*

For untyped module systems, some type p may have no normal form or have several different normal forms. Moreover the normalization of p may not terminate. The following two examples show typical cases.

Example 2 *In the following, the normalization of* $\mathrm{M}_1.\mathrm{M}_2$ *does not terminate.*

$$\begin{array}{l}\{\mathrm{M}_1 = \mathrm{M}_2.\mathrm{M}_3,\\ \mathrm{M}_2 = \mathrm{M}_1\}\end{array}$$

Example 3 *In the following, the normalization of* $\mathrm{M}_1.\mathrm{M}_2(\mathrm{M}_1).\mathrm{M}_3$ *does not terminate.*

$$\mathrm{M}_1 = \{\mathrm{M}_2 = \lambda \mathrm{C}{<:}\mathrm{M}_2'.\{\mathrm{M}_3 = \mathrm{C}.\mathrm{M}_2(\mathrm{C}).\mathrm{M}_3\}\}$$

As our type system relies on normalization for judging type equalities, we sometimes need to use normalization on types for not-yet-typed module systems. In order to keep typing decidable, we define a semi-ground normalization that, contrary to the above "direct" normalization, is guaranteed to terminate. Semi-ground normalization meets the following two requirements when S is well-founded.

- Semi-ground normalization of types always terminates[2]. Moreover we have an algorithm to calculate the set of semi-ground normal forms of types.
- If S is well-typed then, both semi-ground normalization and direct normalization always terminate, and they lead to the same normal form.

[1] *src* (and other predicates we will define in the following sections) should also take the module system we are considering as parameter, but we omit it throughout this paper, supposing a fixed well-founded module system.

[S-VAR]

$$\frac{nlz(\Delta(X),\ \tau)}{X \leq^0 \tau}$$

[S-MRG]

$$\frac{src(p.M,\ q_1 \rightharpoonup q_2) \quad subst(p,\ \theta) \quad nlz(\theta(q_i),\ \tau_i)}{p.M \leq^0 \tau_i \ \text{ for } i = 1, 2}$$

Figure 7. Subtype relation

Using semi-ground normalization, we can decide the typability of a module system in 3 steps.

1. Check for well-foundedness of the dependency relation (we already know this is decidable.)
2. Check the typing using semi-ground normalization in place of direct normalization (normalization is no longer a cause of undecidability.)
3. This typing is also valid with direct normalization (nothing to do.)

The formal definition of semi-ground normalization and the statements of these properties are found in Appendix A.

Basically, semi-ground normalization uses the corresponding interfaces instead of functor arguments when accessing inner modules of variables (hence it is ground.) Remaining variables are substituted with arguments only at the end of this process, once all accesses are solved (hence it is only semi-ground.) Our restriction on functor arguments, which is detailed in Section 6, makes it a valid normalization strategy.

6. Type system

In this section, we define our type system. As defined in Section 3, types are paths or module variables. Let us begin by defining the subtype relation over types. As we judge the equivalence of types by the equality of their normal forms, we first define the subtype relation on normal forms then extend it to any types.

The subtype relation $\leq^0$ on normal forms is the smallest reflective and transitive relation containing the rules given in Figure 7. Subtyping basically arises from merging**[S-MRG]**. **[S-VAR]** denotes that X is a subtype of a normal form of its interface $\Delta(X)$. There are no variance rules associated with parametric types, meaning that parametric types are invariant.

Then, the subtype relation is naturally extended to any types.

Definition 5 (subtype relation) *τ_1 is a subtype of τ_2, denoted $\tau_1 \leq \tau_2$, if there are types τ_1', τ_2' such that $nlz(\tau_1,\ \tau_1')$, $nlz(\tau_2,\ \tau_2')$ and $\tau_1' \leq^0 \tau_2'$ hold.*

[2] Note that our well-foundedness criterion for a module system S is only a sufficient condition for this termination. The converse does not hold.

Module definition typing

$$\frac{\epsilon \vdash \overline{met} \diamond \quad \epsilon \vdash \overline{D} \diamond}{\vdash \{\overline{met}, \overline{D}\} \diamond} \textbf{[T-ROOT]} \qquad \frac{E \not\equiv \lambda \overline{X} <: \overline{q}.\{\overline{met},\ \overline{D}\} \quad \vdash E \diamond}{p \vdash M = E \diamond} \textbf{[T-ExBM]}$$

$$\frac{\vdash q_1 \diamond \ \ldots \ \vdash q_n \diamond \quad p.M([X_i]_{i=1}^n) \vdash \overline{met} \diamond \quad p.M([X_i]_{i=1}^n) \vdash \overline{D} \diamond}{p \vdash M = \lambda [X_i <: q_i]_{i=1}^n.\{\overline{met},\ \overline{D}\} \diamond} \textbf{[T-BM]}$$

Module expression typing

$$\frac{valid(p)}{\vdash p \diamond} \qquad \frac{\begin{array}{c}\vdash p_1 \diamond \quad \vdash p_2 \diamond \\ mergeable(p_1, p_2)\end{array}}{\vdash p_1 \rightharpoonup p_2 \diamond} \qquad \frac{\begin{array}{c}\vdash p \diamond \\ coercible(p,\ \{\overline{m}, \overline{M}\})\end{array}}{\vdash p\ \texttt{coerce}\ \{\overline{m}, \overline{M}\} \diamond}$$

Method typing

$$\frac{\begin{array}{c}\vdash \tau \diamond \quad \vdash \tau' \diamond \\ \texttt{this} : p,\ x : \tau' \vdash e : \tau\end{array}}{p \vdash \tau\ m(\tau'\ x)\{e\} \diamond} \qquad \frac{\vdash \tau \diamond \quad \vdash \tau' \diamond}{p \vdash \texttt{abstract}\ \tau\ m(\tau'\ x) \diamond} \qquad \frac{\vdash \tau \diamond \quad \vdash \tau' \diamond}{p \vdash \texttt{required}\ \tau\ m(\tau'\ x) \diamond}$$

Expression typing

$$\frac{\Gamma \vdash e : \tau' \quad \tau' \leq \tau \quad \vdash \tau \diamond}{\Gamma \vdash e : \tau} \textbf{[T-SUB]} \qquad \Gamma \vdash x : \Gamma(x) \textbf{[T-VAR]}$$

$$\frac{\vdash p \diamond \quad nlz(p,\ p') \quad sig(p', \mathcal{A}, \mathcal{R}, \mathcal{I}, b) \quad N(\mathcal{A}) \cup N(\mathcal{R}) \subseteq N(\mathcal{I})}{\Gamma \vdash \texttt{new}\ p : p} \textbf{[T-NEW]}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : p \quad nlz(p,\ p') \\ sig(p', \mathcal{A}, \mathcal{R}, \mathcal{I}, b) \ \ (m,\ \tau',\ \tau) \in \mathcal{A} \cup \mathcal{R} \cup \mathcal{I} \ \ \Gamma \vdash e' : \tau'\end{array}}{\Gamma \vdash e.m(e') : \tau} \textbf{[T-MTD]}$$

Figure 8. Typing rules

Figure 8 provides the typing rules for *Room*. They use auxiliary predicates to be found in Figure 9 to 13.

Before examining these rules, we explain the predicate *sig* (Figure 9), which is frequently used. This predicate is meant to give information about the *method signature*s of a module. A method signature is a tuple (m, τ, τ') where m is a method name and τ, τ' are types. The metavariables $\mathcal{A}$, $\mathcal{R}, \mathcal{I}$ range over sets of method signatures, and b ranges over false or true. When $sig(p, \mathcal{A}, \mathcal{R}, \mathcal{I}, b)$ holds, $\mathcal{A}$, $\mathcal{R}$ and $\mathcal{I}$ give respectively the sets of abstract methods, required methods, and implemented methods, provided by module p. We give details on the use of b later. Note that, since a concrete module variable may only be instantiated by modules implementing all required methods, in **[Sig-VAR]** required methods are added to the set of implemented methods.

The type judgment $p \vdash M = E \diamond$ states that "the module definition $M = E$ is well-typed in the context p", and $\vdash E \diamond$ states that "the module expression E is well-typed". The type judgment $p \vdash met \diamond$ is read similarly.

A type environment Γ is a finite mapping from program variables to types. The type judgment $\Gamma \vdash e : \tau$ states that "the program expression e has type τ in the type environment Γ".

Module definition typing

A module system S is well-typed when each component of S is well-typed in context ϵ.

If a module is defined by a basic module, its module definition is well-typed if each component defined in the basic module is well-typed in the context of this module.

Otherwise the module definition is well-typed if the module expression defining it is well-typed.

Module expression typing

A module expression p is well-typed when p is *valid*. The formal definition of the validity of paths is given in Figure 10. Roughly, $valid(p)$ checks that p has a normal form, and any application contained in p *match*es the corresponding interface.

match is formally defined in Figure 11. We distinguish the matching of virtual module variables from that of concrete modules variables. When q is the interface of a virtual module variable **[Mat-V]**, then p matches q provided p is a subtype of q. When q is the interface of a concrete module variable **[Mat-C]**, the condition is stricter. Since a concrete module variable may be used in expressions such as "`new` C" or "$C.M$", we must check that all required methods are implemented, and that the identity condition on inner mod-

[Sig-BM]

$$
\begin{array}{ll}
 & sig(\ p, \{[(m_{1i},\ \theta(\tau'_{1i}),\ \theta(\tau_{1i}))]_{i=1}^{n_1}\}, \\
 & \qquad \{[(m_{2i},\ \theta(\tau'_{2i}),\ \theta(\tau_{2i}))]_{i=1}^{n_2}\}, \\
 & \qquad \{[(m_{3i},\ \theta(\tau'_{3i}),\ \theta(\tau_{3i}))]_{i=1}^{n_3}\}, \text{false}) \\
\textbf{:-} & p \equiv p_1.M([q_i]_{i=1}^{n}), subst(p,\ \theta) \\
 & src(p_1.M, \lambda[X_i{<:}q'_i]_{i=1}^{n}.\{ \\
 & \quad [\texttt{abstract}\ \tau_{1i}\ m_{1i}(\tau'_{1i}\ x_{1i})]_{i=1}^{n_1}, \\
 & \quad [\texttt{required}\ \tau_{2i}\ m_{2i}(\tau'_{2i}\ x_{2i})]_{i=1}^{n_2}, \\
 & \quad [\tau_{3i}\ m_{3i}(\tau'_{3i}\ x_{3i})\{e_i\}]_{i=1}^{n_3},\ \overline{D}\}).
\end{array}
$$

[Sig-MRG]

$$
\begin{array}{ll}
 & sig(p.M, \mathcal{A}_1 \cup \mathcal{A}_2, \mathcal{R}_1 \cup \mathcal{R}_2, \mathcal{I}_1 \cup \mathcal{I}_2, b_1 \vee b_2) \\
\textbf{:-} & src(p.M,\ q_1 \rightharpoonup q_2), subst(p,\ \theta), \\
 & nlz(\theta(q_1),\ q'_1), sig(q'_1, \mathcal{A}_1, \mathcal{R}_1, \mathcal{I}_1, b_1), \\
 & nlz(\theta(q_2),\ q'_2), sig(q'_2, \mathcal{A}_2, \mathcal{R}_2, \mathcal{I}_2, b_2).
\end{array}
$$

[Sig-CRC]

$$
\begin{array}{ll}
 & sig(p.M, \mathcal{A}\mid_{\{\overline{m}\}}, \mathcal{R}\mid_{\{\overline{m}\}}, \mathcal{I}\mid_{\{\overline{m}\}}, \text{false}) \\
\textbf{:-} & src(p.M,\ q\ \texttt{coerce}\ \{\overline{m}, \overline{N}\}), subst(p,\ \theta), \\
 & nlz(\theta(q),\ q'), sig(q', \mathcal{A}, \mathcal{R}, \mathcal{I}, b).
\end{array}
$$

where $\mathcal{M}\mid_{\{\overline{m}\}} = \{(m, \tau', \tau) \in \mathcal{M} \mid m \in \{\overline{m}\}\}$.

[Sig-VAR]

$$
\begin{array}{ll}
 & sig(C, \mathcal{A}, \mathcal{R}, \mathcal{I} \cup \mathcal{R}, \text{true}) \\
\textbf{:-} & nlz(\Delta(C),\ q), \\
 & sig(q, \mathcal{A}, \mathcal{R}, \mathcal{I}, b).
\end{array}
$$

Figure 9. Method signature lookup

$$
\begin{array}{lll}
valid(\epsilon). & & \\
valid(X). & & \\
valid(p.M) & \textbf{:-} & valid(p),\ nlz(p.M,\ q). \\
valid(p(q)) & \textbf{:-} & valid(p), valid(q), nlz(p,\ p'), nlz(q,\ q'), \\
 & & params(p',\ X :: L), subst(p',\ \theta), \\
 & & match(q',\ (X,\ \theta(\Delta(X)))).
\end{array}
$$

Figure 10. Validity of paths

[Mat-V]

$$
match(p,\ (V,\ q)) \quad \textbf{:-} \quad p \leq q.
$$

[Mat-C]

$$
\begin{array}{lll}
match(p,\ (C,\ q)) & \textbf{:-} & p \leq q, \\
 & & \forall M(Nlz(p.M) = Nlz(q.M)), \\
 & & sig(p, \mathcal{A}_1, \mathcal{R}_1, \mathcal{I}_1, b_1), \\
 & & nlz(q,\ q'), sig(q', \mathcal{A}_2, \mathcal{R}_2, \mathcal{I}_2, b_2), \\
 & & N(\mathcal{A}_1) \backslash N(\mathcal{I}_1) \subseteq N(\mathcal{A}_2) \backslash N(\mathcal{I}_2), \\
 & & N(\mathcal{R}_1) \subseteq N(\mathcal{I}_1).
\end{array}
$$

Figure 11. Conditions for matching

[Mrg-FF]

$$
\begin{array}{ll}
 & mergeable(p_1, p_2) \\
\textbf{:-} & \forall M(Nlz(p_1.M) = \emptyset \vee Nlz(p_2.M) = \emptyset), \\
 & nlz(p_1,\ p'_1), sig(p'_1, \mathcal{A}_1, \mathcal{R}_1, \mathcal{I}_1, \text{false}), \\
 & nlz(p_2,\ p'_2),\ sig(p'_2, \mathcal{A}_2, \mathcal{R}_2, \mathcal{I}_2, \text{false}), \\
 & \forall m((m,\ \tau_1,\ \tau'_1) \in \mathcal{A}_1 \cup \mathcal{R}_1 \cup \mathcal{I}_1 \\
 & \quad \wedge (m,\ \tau_2,\ \tau'_2) \in \mathcal{A}_2 \cup \mathcal{R}_2 \cup \mathcal{I}_2 \\
 & \quad \Rightarrow\ nlz(\tau_1,\ \tau) \wedge nlz(\tau_2,\ \tau) \\
 & \qquad \wedge nlz(\tau'_1,\ \tau') \wedge nlz(\tau'_2,\ \tau')).
\end{array}
$$

[Mrg-FT]

$$
\begin{array}{ll}
 & mergeable(p_1, p_2) \\
\textbf{:-} & \forall M(Nlz(p_1.M) = \emptyset \vee Nlz(p_2.M) = \emptyset), \\
 & nlz(p_1,\ p'_1),\ sig(p'_1, \mathcal{A}_1, \mathcal{R}_1, \mathcal{I}_1, \text{false}), \\
 & nlz(p_2,\ p'_2),\ sig(p'_2, \mathcal{A}_2, \mathcal{R}_2, \mathcal{I}_2, \text{true}), \\
 & \forall m((m,\ \tau_1,\ \tau'_1) \in \mathcal{A}_1 \cup \mathcal{R}_1 \cup \mathcal{I}_1 \\
 & \quad \Rightarrow (m,\ \tau_2,\ \tau'_2) \in \mathcal{A}_2 \cup \mathcal{R}_2 \cup \mathcal{I}_2 \\
 & \qquad \wedge nlz(\tau_1,\ \tau) \wedge nlz(\tau_2,\ \tau) \\
 & \qquad \wedge nlz(\tau'_1,\ \tau') \wedge nlz(\tau'_2,\ \tau')).
\end{array}
$$

[Mrg-SYM]

$$
mergeable(p_1, p_2) \quad \textbf{:-} \quad mergeable(p_2, p_1).
$$

Figure 12. Mergeability

ules is satisfied. The former translates to the two following requirements: all required methods of q are implemented in p, and the abstract methods of p are a subset of the abstract methods of q. Here $N(\mathcal{M}) = \{m \mid (m, \tau, \tau') \in \mathcal{M}\}$ extracts method names from method signatures.

The latter is done by checking that for every module name M, either both p and q have a submodule $p.M$, defined identically, or they both lack it. For this we use the set of normal forms of p, defined as $Nlz(p) = \{q \mid nlz(p,\ q)\}$[3]. When this condition is satisfied for all modules, direct normalization and semi-ground normalization coincide. This restriction means that we should pass functor arguments as several flat modules rather than one module containing all of them.

A module expression $p_1 \rightharpoonup p_2$ is well-typed when both p_1 and p_2 are well-typed and the following two conditions are satisfied: 1) p_1 and p_2 do not contain modules with the same names, 2) if both p_1 and p_2 contain identically named methods, then these methods have the same signatures. We formally define these conditions in Figure 12.

We must pay particular attention to modules derived from non-coerced concrete module variables, as they might have more methods than described in their interfaces. The 5th argument of *sig* is used for that purpose. It is set to true for modules derived from non-coerced module variables, false otherwise. Rule **[Mrg-FT]** states that when one of the modules inherits from a non-coerced module variable, this mod-

[3] As direct normalization does not always terminate, $Nlz(p)$ works as an oracle in typing derivations. However, semi-ground normalization always terminates, and we have an algorithm that calculates the set of semi-ground normal forms of paths.

$$
\begin{array}{ll}
 & coercible(p,\ \{\overline{m}, \overline{M}\}) \\
\text{:-} & \forall M(M \in \{\overline{M}\} \Rightarrow nlz(p.M,\ q)), \\
 & nlz(p,\ p'), sig(p', \mathcal{A}, \mathcal{R}, \mathcal{I}, b), \\
 & N(\mathcal{A}) \cup N(\mathcal{R}) \subseteq \{\overline{m}\} \cup N(\mathcal{I}), \\
 & \{\overline{m}\} \subseteq N(\mathcal{A}) \cup N(\mathcal{R}) \cup N(\mathcal{I}).
\end{array}
$$

Figure 13. Coercibility

ule should have signatures for all the methods in the other module. This way we make sure that the typing is consistent. For the same reason, we cannot merge two modules, both derived from non-coerced module variables.

A module expression p `coerce` $\{\overline{m}, \overline{M}\}$ is well-typed when p is well-typed and the following three conditions are satisfied: 1) for all M in $\{\overline{M}\}$, p contains a module named M, 2) for all m in $\{\overline{m}\}$, p contains a method named m, 3) all the not-yet-implemented methods of p are contained in $\{\overline{m}\}$. The last condition is needed to avoid hiding unimplemented methods, as hidden methods cannot be overridden. The formal definition of these conditions is given in Figure 13.

Program expression typing

The typing rules for program expressions are classical. Hence, we only give a brief account.

The rule **[T-SUB]** is the subsumption rule. The rule for program variables **[T-VAR]** is as usual. The rule for object creation **[T-NEW]** checks that all methods are implemented. The rule for method invocation **[T-MTD]** first checks that e has type p, and p has a method m with signature (m, τ', τ). Then, it checks that e' has type τ'. If all of these conditions are satisfied, then $e.m(e')$ has type τ.

Definition 6 *The module system S is well-typed if and only if $\vdash\ S \diamond$ holds. Moreover, the program (S, e) has type τ, denoted as $\vdash (S, e) : \tau$ if and only if S is well-typed, e does not contain module variables, and $\vdash e : \tau$ holds.*

In Appendix A, we establish the result that, if we use semi-ground normalization instead of direct normalization, then type checking of a well-founded module system is decidable. Moreover, type checking with direct normalization and semi-ground normalization are equivalent.

7. Operational semantics

In this section, we provide the operational semantics for *Room*. The purpose of the semantics is to reduce a program expression to a *value*. A value is a reference $\mathrm{obj}(\ell, w)$ to an object, where ℓ is a *location*, which is an element of an infinite enumerable set *Loc*, and w is a *method dictionary*, which is a finite mapping over method names.

Values refer to objects. An object in *Room* is a collection of labeled components $[m_1 = \zeta_1, \ldots m_n = \zeta_n]$ where m_i is a method name and ζ_i is a *closure*. A closure is a 4-tuple (p, w, x, e): p is a path, meant for an evaluation context, w is a method dictionary, x is a program variable, meant

$$
\frac{\begin{array}{c} subst(p,\ \theta) \quad nlz(\theta(q),\ q') \\ elb(q',\ \{[(m_i, \zeta_i)]_{i=1}^n\}) \quad \ell \notin dom(\kappa) \end{array}}{\begin{array}{c} \iota;\ \kappa;\ p \models \texttt{new}\ q\ \Downarrow (\mathrm{obj}(\ell, \mathrm{id}),\ \iota,\ \kappa') \\ \text{where } \kappa' \equiv \kappa[\ell \mapsto \{[m_i = \zeta_i]_{i=1}^n\}] \end{array}}
$$

$$
\frac{\begin{array}{c} \iota;\ \kappa;\ p \models e\ \Downarrow (\mathrm{obj}(\ell, w_0),\ \iota_0,\ \kappa_0) \\ \kappa_0(\ell).w_0(m) = (p_1,\ w_1,\ x,\ e'') \\ \iota_0;\ \kappa_0;\ p \models e'\ \Downarrow (v_1,\ \iota_1,\ \kappa_1) \\ \texttt{this} : \mathrm{obj}(\ell, w_1), x : v_1;\ \kappa_1;\ p_1 \models e''\ \Downarrow (v_2,\ \iota_2,\ \kappa_2) \end{array}}{\iota;\ \kappa;\ p \models e.m(e')\ \Downarrow (v_2,\ \iota_1,\ \kappa_2)}
$$

$$
\iota;\ \kappa;\ p \models x\ \Downarrow (\iota(x),\ \iota,\ \kappa)
$$

Figure 14. Operational semantics

for a formal parameter, e is a program expression, meant for a method body. We take into account hiding of methods caused by coercion by adding method dictionaries to closures. Any method invocation on self, which is expressed as `this`$.m$, is done by looking up its actual name in the method dictionary.

Given an *object store* κ, which is a finite mapping from locations to objects, a value $\mathrm{obj}(\ell, w)$ refers to an object stored in the location ℓ of κ, denoted $\kappa(\ell)$, and any method invocation on $\mathrm{obj}(\ell, w)$ is done consulting w.

An *execution state* is a couple (ι, κ): ι is a finite mapping from program variables to values, κ is an object store.

Our operational semantics is given in terms of a reduction relation $\Downarrow$. We write $\iota;\ \kappa;\ p \models e\ \Downarrow (v,\ \iota',\ \kappa')$ to mean that in the context p with the execution state (ι, κ), e is evaluated to the value v and the execution state transits to (ι', κ'). The rules for the semantics are given in Figure 14 with an auxiliary predicate in Figure 15.

The first rule of the semantics describes object creation. In order to evaluate `new` q in context p, the module $\theta(q)$ should be *elaborated*, where θ is the substitution extracted from p. Elaboration is defined by means of the predicate *elb* given in Figure 15. It traverses, with allowance for method hiding, all modules which contribute to the module referred to by a path, in order to collect all the methods constituting the module. $elb(p,\ \{(m_1, \zeta_1), \ldots, (m_1, \zeta_n)\})$ means that the module p has methods m_i with closures ζ_i. If the elaboration of $\theta(q)$ resolves, the result is added to the object store κ. The second rule describes method invocation. In order to evaluate $e.m(e')$, we should first calculate the result of e, check that the result refers to an object which has the target method m seen through the method dictionary, calculate the result of e', and then evaluate the invocated method's body. The third rule implements access to variables. Note that, runtime elaboration is not needed actually, as we statically know which paths should be elaborated.

[Elb-BM]

$$
\begin{array}{ll}
 & elb(p,\ \{[(m_i,\ (p,\ \mathrm{id},\ x_i,\ e_i))]_{i=1}^{n})\}) \\
\text{:-} & p \equiv p_1.M([q_i]_{i=1}^{n'}), \\
 & src(\underline{p_1.M,\ \lambda[X_i{<:}r_i]_{i=1}^{n'}}.\{ \\
 & \quad \underline{\texttt{abstract}\ \tau\ m(\tau\ x)}, \\
 & \quad \underline{\texttt{required}\ \tau\ m(\tau\ x)}, \\
 & \quad [\tau_{1i}\ m_i(\tau_{2i}\ \ x_i)\{e_i\}]_{i=1}^{n},\ \overline{D}\}).
\end{array}
$$

[Elb-CRC]

$$
\begin{array}{ll}
 & elb(p.M,\ \{[(w(m_i),\ (p_i,\ w \circ w_i,\ x_i,\ e_i))]_{i=1}^{n})\}) \\
\text{:-} & src(p.M,\ q\ \texttt{coerce}\ \{\overline{m}, \overline{M}\}), subst(p,\ \theta), \\
 & nlz(\theta(q),\ q'), elb(q',\ \{[(m_i,\ (p_i,\ w_i,\ x_i,\ e_i))]_{i=1}^{n})\}).
\end{array}
$$

where w is a mapping which renames method names in $\{[m_i]_{i=1}^{n}\} \setminus \{\overline{m}\}$ to fresh names.

[Elb-MRG]

$$
\begin{array}{ll}
 & elb(p.M,\ \mathcal{M}) \\
\text{:-} & src(p.M,\ q_1 \rightharpoonup q_2), subst(p,\ \theta) \\
 & nlz(\theta(q_1),\ q_1'), nlz(\theta(q_2),\ q_2'), \\
 & elb(q_1',\ \mathcal{M}_1), elb(q_2',\ \mathcal{M}_2) \\
 & \mathcal{M} = \mathcal{M}_1 \cup (\mathcal{M}_2 \mid_{N(\mathcal{M}_2) \setminus N(\mathcal{M}_1)}).
\end{array}
$$

where $N(\mathcal{M}) = \{m \mid (m, \zeta) \in \mathcal{M}\}$.

Figure 15. Elaboration

The following proposition states that the type system guarantees the module elaboration.

Proposition 2 *If the module system S is well-founded and well-typed, and $\vdash\ p \diamond$ holds, then the elaboration of p is always successful.*

As we have an algorithm that checks whether S is well-founded or not, and a decidable type system(see Appendix A), we can statically guarantee all the elaboration needed during evaluation.

Type Soundness

Our type soundness states that if a program has a type, then either it reduces to a value of the same type, or its evaluation does not terminate. In the following of this section, we assume that the program (S, e) is well-founded and well-typed.

To reason about type soundness, we extend program expression typing to account for the context in which the expression is type checked, and define a judgment for value typing. We write $V(p)$ to denote the set of module variables contained in p.

The type judgment $p; \Gamma \vdash e : \tau$ states that the program expression e has type τ in context p under the type environment Γ.

Definition 7 *$p; \Gamma \vdash e : \tau$ holds if $\Gamma \vdash \theta(e) : \tau$ holds, where θ is the substitution extracted from p.*

The judgment $\kappa \vdash v : \tau$ asserts that the value v has type τ under the object store κ. It checks that the object referred to by v has signatures for all the methods the module referred to by τ has.

Definition 8 *$\kappa \vdash v : \tau$ holds if $v \equiv \mathrm{obj}(\ell, w)$, $nlz(\tau,\ \tau')$, $sig(\tau', \mathcal{A}, \mathcal{R}, \mathcal{I}, b)$, and for all $(m, \tau_1', \tau_1) \in \mathcal{A} \cup \mathcal{R} \cup \mathcal{I}$, $\kappa(\ell).w(m) = (p, w', x, e)$ and $p; \texttt{this} : p, x : \tau_1' \vdash e : \tau_1$*

The following theorem states type soundness formally.

Theorem 1 *If the well-founded program (S, e) has type τ, then either the evaluation of e does not terminate, or $\emptyset;\ \emptyset;\ \epsilon \models e\ \Downarrow (v,\ \iota',\ \kappa')$ and $\kappa' \vdash v : \tau$ hold.*

Our type system also ensures the progress property, *i.e.* well-typed terms do not get stuck due to the absence of applicable rules. This result is shown in [11], by introducing run-time errors which distinguish incorrect termination.

8. Related Work

In this section, we examine related work. We first take up languages and calculi which have mechanisms for both ML-style modules and classes, then compare our approach to existing approaches to recursive modules in terms of well-foundedness of the recursion.

νObj [12] is a calculus for objects and classes. It identifies objects with modules, and classes with functors. Most mechanisms of ML-modules and classes are supported in νObj, including higher-order functors, which are missing in *Room*. On the one hand, our subtype relation is weaker than that of νObj, which is a reason why we have a decidable type system, and they do not. On the other hand, their support for mutual recursion is less flexible than ours, while we retain decidability.

Objective Caml [9] and Moscow ML [14] are real languages, that support recursion between modules. As their type systems do not guarantee well-foundedness of the recursion, run-time errors might occur because of cycles in module import dependency graphs.

Moby [6] and Loom [4] have both of modules and classes, however they lack inter-module recursion, which is the main motivation for *Room*.

Mixin modules (hereafter, "mixins") are modules equipped with class mechanisms such as mutual recursion or overriding. Ancona&Zucca notably developed a calculus for mixins [2], and, based on it, constructed a module system, called JAVAMOD [1], on top of a Java like language. In JAVAMOD, they face the problem of cycles in the inheritance hierarchy. Yet, as the modules of JavaMod are not hierarchical, the problem is much simpler and easily solved. Hirschowitz&Leroy investigated a mixin calculus in a call-by-value settings [8], which requires them to statically reject

ill-founded recursion between mixins. They employ a different approach from ours, which we review in detail below. They do not consider nested structures and type members.

Boudol [3], Hirschowitz&Leroy [8] and Dreyer[5] have investigated type systems for well-founded recursion. They track recursively used variables while checking that they are protected under lambda abstraction. On the one hand, we can access to components of a module before the evaluation of the module is yet completed, which is illegal in their systems. On the other hand, their modules can recursively refer to themselves inside their own definition if the reference is protected under a lambda abstraction, which is illegal in *Room* regardless of whether there is a lambda abstraction or not. For example, the following module system:

$$\begin{array}{rcl}\{\mathrm{M} & = & \{\mathrm{M}_1 = \mathrm{N.N}_2,\ \mathrm{M}_2 = \{\cdots\}\}, \\ \mathrm{N} & = & \{\mathrm{N}_1 = \mathrm{M.M}_2,\ \mathrm{N}_2 = \{\cdots\}\}\}\end{array}$$

is accepted in *Room*, but rejected in theirs. On the other hand, our definition of well-foundedness excludes the module system $\mathrm{M} = \lambda X{<:}\mathrm{N}.\{\mathrm{M}_1 = \mathrm{M}\}$, which is legal in their systems. Module systems of the former form are needed to support mutual recursion as seen in object-oriented programming. However, the absence of the latter form means that we have no way to define modules as fixpoints of functors.

9. Conclusion

In this paper, we presented an object-oriented module calculus, which unifies classes, nested structures, simple functors, and their types. The unification eases the simultaneous use of ML-style module and class mechanisms.

Mutual recursion is fundamental to classes, yet, it might allow undesirable modules which have circular dependencies or are inconsistent, when we introduce it into an ML-style module setting. We defined a decidable type system, which ensures the absence of such ill-founded modules. Decidability is reached by first eliminating ill-found module systems by verifying their dependency relation on flat paths, and then checking types with a variant of normalization guaranteed to terminate when this relation is well-founded.

There are two points we would like to improve in *Room*. First, it would be nice to make it more liberal with recursion. *Room* is flexible enough for mutual recursion, yet it lacks the ability to define modules as fixpoints of functors. A possible approach would be to introduce two kinds of functor applications, one for virtual module variables, the other for concrete module variables. This approach seems to work well, but would make our calculus more verbose. We are still looking for a better solution. Second, the condition on inner modules of functor arguments seems to be overly restrictive: actual values of concrete module variables must have exactly the same inner modules as their corresponding interfaces. This is not an essential restriction, as one can always pass inner modules as independent parameters, but we would like to relax it, to make our calculus more general.

References

[1] D. Ancona and E. Zucca. True modules for Java-like languages. In *Proc. ECOOP'01*, number 2072 in Springer LNCS, pages 354–380, 2001.

[2] D. Ancona and E. Zucca. A Calculus of Module Systems. *Journal of Functional Programming*, 12(2):91–132, 2002.

[3] Gerard Boudol. The recursive record semantics of objects revisited. *Journal of Functional Programming*, 14:263–315, 2004.

[4] K. Bruce, L. Petersen, and J. Vanderwaart. Modules in LOOM: Classes are not enough. http://www.cs.williams.edu/kim, 1998.

[5] Derek Dreyer. A type system for well-founded recursion. In *Proc. POPL'04*, 2004.

[6] Kathleen Fisher and John H. Reppy. The Design of a Class Mechanism for Moby. In *Proc. PLDI'99*, pages 37–49, 1999.

[7] R. Garcia, J. Järvi, A. Lumsdaine, J. Siek, and J. Willcock. A comparative study of language support for generic programming. In *Proc. OOPSLA'03*, pages 115–134, 2003.

[8] Tom Hirschowitz and Xavier Leroy. Mixin modules in a call-by-value setting. In *Proc. ESOP'02*, number 2305 in Springer LNCS, pages 6–20, 2002.

[9] X. Leroy, D. Doligez, J. Garrigue, and J. Vouillon. The Objective Caml system. Software and documentation available on the Web, http://caml.inria.fr /.

[10] Xavier Leroy. Manifest types, modules, and separate compilation. In *POPL'94*, pages 109–122. ACM Press, 1994.

[11] Keiko Nakata, Akira Ito, and Jacques Garrigue. Recursive object-oriented modules. Extended version. http://www.kurims.kyoto-u.ac.jp/~keiko/.

[12] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In *Proc. ECOOP'03*, 2003.

[13] Benjamin C. Pierce. Bounded quantification is undecidable. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 427–459. The MIT Press, Cambridge, MA, 1994.

[14] S. Romanenko, C. Russo, N. Kokholm, and P. Sestoft. Moscow ML. Software and documentation available on the Web, http://www.dina.dk/ sestoft/mosml.html.

A. Appendix

In this section, we define semi-ground normalization and establish technical results on it.

The intuition of semi-ground normalization is to look at interfaces instead of actual values when pulling out inner modules from module variables. This works well because our type system ensures that the inner modules of actual values coincide with the inner modules of their corresponding interfaces.

Formally, semi-ground normalization is defined by the predicate *gnlz* given in Figure 16, and the function η given in Figure 17, where $\tilde{\Delta}$ replaces a variable by its interface until it obtains an absolute path (*i.e.* not starting by a variable).

[P-ROOT]

$$gnlz(\epsilon, \epsilon).$$

[P-VAR]

$$gnlz(X, X).$$

[P-APP]

$$\begin{array}{ll} & gnlz(p(q), p'(q')) \\ \textbf{:-} & gnlz(p, p'), gnlz(q, q'). \end{array}$$

[P-ExPATH]

$$\begin{array}{ll} & gnlz(p.M, p'.M) \\ \textbf{:-} & gnlz(p, p'), \\ & src(p'.M,\ E), \\ & E \equiv C \vee \neg(E \equiv q). \end{array}$$

[P-PATH]

$$\begin{array}{ll} & gnlz(p.M, q) \\ \textbf{:-} & gnlz(p, p'), \\ & src(p'.M,\ r), \\ & \neg(r \equiv C), \\ & subst(p',\ \theta), \\ & gnlz(\theta(r), q). \end{array}$$

[P-CRC]

$$\begin{array}{ll} & gnlz(p.M, q) \\ \textbf{:-} & gnlz(p, p'.N), \\ & src(p'.N,\ r\ \texttt{coerce}\ \{\overline{M}, \overline{m}\}), \\ & M \in \{\overline{M}\}, \\ & subst(p',\ \theta), \\ & gnlz(\theta(\tilde{\Delta}(r)).M, q). \end{array}$$

[P-MRG1]

$$\begin{array}{ll} & gnlz(p.M, q) \\ \textbf{:-} & gnlz(p, p'.N), \\ & src(p'.N,\ r \rightharpoonup r'), \\ & subst(p',\ \theta), \\ & gnlz(\theta(\tilde{\Delta}(r)).M, q). \end{array}$$

[P-MRG2]

$$\begin{array}{ll} & gnlz(p.M, q) \\ \textbf{:-} & gnlz(p, p'.N), \\ & src(p'.N,\ r \rightharpoonup r'), \\ & subst(p',\ \theta), \\ & gnlz(\theta(\tilde{\Delta}(r')).M, q). \end{array}$$

[P-PAR]

$$\begin{array}{ll} & gnlz(p.M, q) \\ \textbf{:-} & gnlz(p, p'.N), \\ & src(p'.N,\ C), \\ & subst(p',\ \theta), \\ & gnlz(\theta(\tilde{\Delta}(C)).M, q). \end{array}$$

[P-INF]

$$gnlz(C.M, q) \quad \textbf{:-} \quad gnlz(\Delta(C).M, q).$$

Figure 16. Ground-normalization

$$\begin{array}{lcl} \eta(X) & = & X \\ \eta(p.M) & = & \begin{cases} \eta(\theta(C)) & \text{if } src(p.M,\ C) \text{ and } subst(p,\ \theta) \text{ hold} \\ \eta(p).M & \text{otherwise} \end{cases} \\ \eta(p(q)) & = & \eta(p)(\eta(q)) \end{array}$$

Figure 17. Variable normalization

It is defined as follows:

$$\tilde{\Delta}(p) = \begin{cases} \tilde{\Delta}(\Delta(X)) & (p \equiv X) \\ \tilde{\Delta}(\Delta(C)(\overline{q}).r) & (p \equiv C(\overline{q}).r) \\ p & (\text{otherwise}) \end{cases}$$

Definition 9 *A path q is a semi-ground normal form of p if $gnlz(p, q')$ and $\eta(q') = q$ hold.*

We use subscript W to denote type judgments with semi-ground normalization, *e.g.* $\vdash_W\ S \diamond$ denotes that S is well-typed when type checked with semi-ground normalization.

Theorem 2 *Let (S, e) be a well-founded program, it is decidable whether $\vdash_W\ (S, e) : \tau$ holds or not. Moreover, $\vdash_W\ (S, e) : \tau$ holds if and only if $\vdash (S, e) : \tau$ holds.*

Above theorem is a direct result from the following proposition.

Proposition 3 *Let (S, e) be a well-founded program, then*

- *semi-ground normalization of paths always terminates.*
- *the set of semi-ground normal forms of any path is finite, and we have an algorithm that calculates this set.*
- *it is decidable whether $\vdash_W\ S \diamond$ holds or not.*
- *for any path p, it is decidable whether $\vdash_W\ p \diamond$ holds or not.*
- *if $\vdash S \diamond$ then $\vdash_W\ S \diamond$, and vice versa.*
- *if $\vdash S \diamond$, then $\vdash p \diamond$ holds if and only if $\vdash_W\ p \diamond$ holds.*
- *if $\vdash\ S \diamond$ and $\vdash\ p \diamond$, then the normal form of p coincides with the semi-ground normal form of p.*
- *if $\vdash\ S \diamond$ and $\vdash\ p \diamond$, then the elaboration of p is always successful.*

For reasons of space, we refer the proof for the proposition to the extended version [11].