Hierarchical Semantics of Modelica

Christoph Höger[1]

[1]Institute of Software Engineering and Theoretical Computer Science, Technische Universität Berlin, Germany
christoph.hoeger@tu-berlin.de

Abstract

We present a definition of syntax and semantics for Modelica's hierarchical lookup. By using a context-independent encoding of the static semantics of free variables, it becomes possible to define the evaluation of references within a calculus based on substitution. Hence, all steps of evaluation have a concrete syntactic representation. We augment the calculus with a terminating evaluation and a semantics-preserving translation to a basic λ-calculus.
Keywords: Semantics, Classes, Compilation

1 Introduction

In current Modelica, there is no way to express the definition of a variable as a purely syntactic property, independent of the context in which it might be used. Its definition is obtained as part of the dynamic semantics of the flattening process. This effectively renders static analysis of models and packages impossible. Furthermore, there is no formal method to obtain its meaning from the a found definition in the context of a simulation model, as the dynamic semantics of hierarchical elements are defined only informally.

This paper attempts to improve this situation by the means of a compositional core calculus of classes, MCL. In this language, we define syntactic elements for the expression of static properties of variables in a class. The semantics of Modelica-style hierarchical classes is integrated within the framework of the classic λ-calculus. This integration is inspired by the treatment of modules by Pierce (2005). For the evaluation, we focus solely on the problems mentioned above. For a discussion of the relation between model elaboration and the λ-calculus, we refer to earlier work(Höger 2016). In a final step, we present a translation that replaces the hierarchical elements with semantically identical non-hierarchical terms. This shows how a hierarchical model can be translated into a simpler functional language.

The rest of this paper is organized as follows: An introduction into Modelica's scoping and hierarchical organization leads to the definition of the hierarchical core calculus of MCL. This is followed by a graphical interpretation of the hierarchical environment and consequently its semantics. The paper concludes with a *transformation* of the hierarchical aspects to more basic elements of the language. This transformation is shown to be faithful in the sense that it preserves the evaluation semantics.

2 Modelica Scoping and Hierarchies

In its simplest form, a Modelica class serves as a container for a sequence of *declarations*. These may introduce constants, parameters, unknowns or declare components that are instances of other classes. The meaning of variables in the right-hand sides of these declarations is somewhat intricate as the example in Listing 1 shows.

The declaration of the constant `x` in class `A` refers to two free variables, `y` and `z`. Class `A` is a child of class `B`, which is in turn a child of `C` in the *class-hierarchy*. Hence it "sees" all declarations[1] of its parent classes. In the classical sense, `B` is part of `A`'s lexical scope. Therefore, `z` is found directly in the surrounding scope. Note that the definition of constant `z` (the literal `21`) is syntactically placed *after* `A`. The scope of a binding is independent from the order of declarations. Variable `y` is not defined inside `B`. The next candidate is `C`, where it is defined as modelicaB.z.

Such a composite name gives access to elements *downwards* the hierarchy. In a first step, `B` is found as before in the scope of `C`. The result of this search is then used to search for `z`, which is defined as `21`. Hence the result of evaluating `C.B.A.x` should yield `42`.

Although this kind of scoping might seem pretty standard, there is a subtle difficulty embedded in this seemingly simple principle. In Modelica, there is no (syntactic) difference between looking up a class (e.g. `B`) and its fields (e.g. `z`). What might seem like an elegant unification, turns out to be a source of major complication in combination with inheritance.

2.1 Inheritance and Modifications

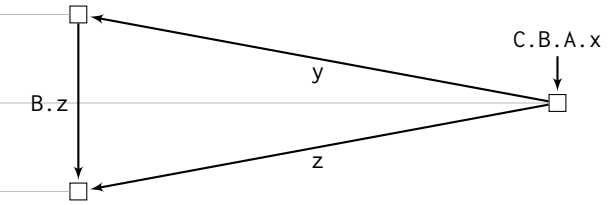
Lexical scoping as it is used above is still a pretty straightforward matter: After all, the environment in which to look for the definition of a variable is determined by the syntactical composition of classes. The complexity rises drastically however, once inheritance (expressed as **`extends`** statements) comes into play:

```
class D
  extends C.B(z=2);
  constant Integer x = A.x;
end D;

class C ... end C;
```

[1]At least the ones with the proper variability

Listing 1. Hierarchical Lookup

```
class C
    constant Integer y = B.z;
    class B
        class A
          constant Integer x = y + z;
        end A;

        constant Integer z = 21;
    end B;
end C;
```



In the class D above, with C left unchanged, what is the value of D.x? Since A is inherited in D from C.B, it is tempting to assume the answer is, again, 42. Instead, the returned value is 23[2].

The reason for this is shown in Listing 2. In the first step, A is found to be inherited from the base class C.B. This lookup succeeds immediately without further involvement of inheritance. Hence, A.x is resolved by looking for x in B. This class contains the same definition of x as before. Accordingly z and y need to be looked up again. Variable z is again looked up in its immediately enclosing scope. This time, this scope is not provided by A, but by the inheriting class D. Therefore the resulting value is 2.

In an interesting twist, y is *not* subject to this modification. Since its lookup passes through C and only then returns to the definition of z the inheritance is discarded. The resulting value is therefore found in the lexical scope of A, and hence yields 21. The overall evaluation yields 23.

This example demonstrates an important fact about Modelica-classes. The site of the definition of a free variable is not a syntactic property of the class. Instead, it depends on the *context* in which this class is used.

2.2 The Principle of Open Recursion

This context is the result of evaluating all relevant super classes. Therefore, the definition of lookup has to be part of the evaluation of classes and vice versa. In Modelica there is no explicit ordering between declarations. Due to the existence of inheritance and because classes are looked up in the same way as other declarations, such an ordering cannot be found without knowledge of the context of the class. Both the construction of the context and the evaluation of class references are recursively linked.

In classical object-oriented languages, this principle is called *open recursion*(Aldrich and Donnelly 2004): Each method has access to a special variable (often called this or self). Methods are always invoked from a concrete object (sometimes called the reciever of a message). This object then becomes the definition of the special variable during evaluation of the method's body (the special variable is *late bound*). Free variables in the method are interpreted by method invocation on the special variable. This principle yields an implementation of *recursion*, since the method itself is an element of the receiving object. It is *open*, since the method might be part of different concrete objects (and invoke different siblings on each). Hence, it is possible to change the behavior of all methods of an object by exchanging only one method. The same concept can be used to explain the lookup inside Modelica's classes, when it is applied not only to one, but possibly many special variables.

```
class D
  constant Integer z = 2;
  extends up(1).C.B;
  constant Integer x = this.A.x;
end D;

class C
  constant Integer y = this.B.z;
  class B
    class A
      constant Integer x = up(2).y + up(1).z;
    end A;

    constant Integer z = 21;
  end B;
end C;
```

In the listing above, references to the context have been codified by two kinds of special variables: this denotes a reference to the immediately enclosing class, while up(i) expresses access to the *i*-th enclosing class (hence up(0) is the same as this, but less readable). The benefit of such a form lies in the fact that it eliminates any free variables and still allows to use the class in different contexts.

3 MCL

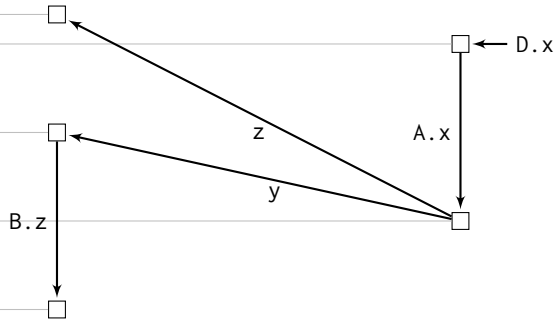
In order to focus the discussion on the hierarchical semantics by the means of such special variables, it is useful to define a minimal calculus and ignore the any feature of Modelica that does not directly contribute to the discussion. To this end we define MCL, a small core calculus that embeds hierarchical term into the classical minimal λ-calculus. Besides the more concise representation, such a reduction allows to express the *complete* domain of discourse. In the following sections, all relevant elements can be expressed in the form of expressions in the core

[2] as discussed in https://trac.modelica.org/Modelica/ticket/2013

DOI 10.3384/ecp17132703

Listing 2. Hierarchical Lookup with Inheritance

```
class D
   extends C.B(z=2);
   constant Integer x = A.x;
end D;
class C
    constant Integer y = B.z;
    class B
        class A
          constant Integer x = y + z;
        end A;

        constant Integer z = 21;
    end B;
end C;
```



language. There is no need to resort to externally (and imprecisely) defined entities like tables, environments or universes of classes.

3.1 Notational Conventions

Languages are defined in a simple BNF-form: Nonterminals (e.g. t, v are expressed with the same small italic letters as meta-variables of the corresponding syntactic sort (e.g. we will use t to denote both the set of terms and a variable from that set). Productions (e.g. $t ::= \lambda x.t \mid x$) map a nonterminal (to the left of the $::=$) to clauses consisting of nonterminal and terminal symbols. Clauses are separated by a $\mid$. Each clause is one possible derivation of the left hand side. Terminal symbols (e.g. , `true`, `if`) are written in a non-proportional font.

Partial functions are univalent relations $\{x \mapsto y\}$. These relations can be augmented using the $\oplus$-operator, borrowed from the specification language Z:

$$p \oplus q \quad \hat{=} \quad \{x \mapsto y \mid x \mapsto y \in p \text{ and } x \notin \mathbf{dom}(q)\} \cup q$$

$[\![t]\!]_\Delta$ denotes the function Δ applied to t. In order to enhance readability, these (recursive) functions are defined using pattern matching on their arguments: $[\![t_1\ t_2]\!]_\Delta$ means the application of Δ to *one* term formed by the juxtaposition of two (possible distinct) terms (i.e. the term representing the application of t_1 to t_2). If multiple arguments are passed to a semantic function, they are separated by commas. Meta variables are bound in the patterns or corresponding *where*-clauses. When necessary, we consider each function as *overloaded* on different syntactic sorts, e.g the function Π can be applied to recursive definitions $\mathcal{F}$ as well as the fields of a class $\overline{F}$.

Sequences are abbreviated by an overline over the name of the contained meta variables, e.g. $\bar{t}$ describes a sequence $t_1 \ldots t_n$. The empty sequence is $\Diamond$. Non-empty sequences are written as pairs of a value and the remaining sequence, separated by a double colon, e.g. $s :: \bar{t}$ describes the sequence $s, t_1, \ldots, t_n$. The operator $\times$ maps a semantic function on a sequence, e.g. $f \times \overline{F}$ yields a sequence where every element is the result of applying f to the corresponding element in $\overline{\Phi}$.

In order to not confuse meta-level equality (e.g. of terms) with its object-level counterpart (e.g. in an equation), we write $a \hat{=} b$ to indicate the former (and $a \hat{\neq} b$ for the opposite).

3.2 Syntax

The syntax of MCL distinguishes between hierarchical terms h and proper terms t (Figure 1). There are five variants of hierarchical terms: The special variables `up`(i) refer to the i-th enclosing class. Literal classes C are a list of fields F bracketed in special curly braces, $\{|\overline{F}|\}$. A class field can either contain a hierarchical class (e.g. a child class) (L `=` h) or a value (l `=` t. We presume that each class-label L can be distinguished from each label l: $L \cap l \hat{=} \emptyset$. A hierarchical node v denotes a class containing the fields $\overline{F}$ as a hierarchical child of the enclosing class denoted by $\overline{\pi}$, written $\{|\overline{F}$ `in` $\overline{\pi}|\}$ (the environment is thus encoded as a list of nodes and each node contains its own environment). Explicit Modifications $\{|h$ `with` $\overline{F}|\}$ override the fields defined in the class described by *hterm* with the fields in $\overline{F}$.

Access to the field L of a class requires an explicit notion of the corresponding super class in a reference R: h_1`.super(`h_2`).`L. Here, h_1 refers to a class extending h_2 which in turn describes the definition-site of the declaration labeled with l. The access reads as: "Get the field labeled with L in the class h_2 extended by h_1". This makes the interface of a class immediately visible (since all inherited fields have to be defined locally as forward references) and is a necessary precondition for a substitution-based semantics. If no super class shall be referenced directly, both parts of a reference are equal. Since this is a common case, we introduce the abbreviation $h.L \hat{=} h$`.super(`h`).`L to enhance the readability.

Terms t consist of the standard elements of the λ-calculus extended with non-strict conditional, and an explicit fixed point operator `fix` (which ranges over multiple,

Hierarchical Terms:

$$
\begin{array}{lcl}
h & ::= & \texttt{up}(i) \mid C \mid v \\
 & \mid & \texttt{\{|}h\ \texttt{with}\ \overline{F}\texttt{|\}} \mid R \\
R & ::= & h\texttt{.super(}h\texttt{).}L \\
v,\pi & ::= & \texttt{\{|}\overline{F}\ \texttt{in}\ \overline{v}\texttt{|\}} \\
C & ::= & \texttt{\{|}\overline{F}\texttt{|\}} \\
F & ::= & L\ \texttt{=}\ h \mid l\ \texttt{=}\ t
\end{array}
$$

Core Terms:

$$
\begin{array}{lcl}
t & ::= & x \mid v \mid t\ t \mid t \circ t \mid r \mid \texttt{if}\ t\ \texttt{then}\ t\ \texttt{else}\ t \\
r & ::= & h\texttt{.super(}h\texttt{).}l
\end{array}
$$

Values:

$$
\begin{array}{lcl}
v & ::= & b \mid \lambda x.t \mid \texttt{fix}\ x\ \texttt{in}\ \mathcal{F} \mid \texttt{true} \mid \texttt{false} \mid \mathbb{Z} \mid \mathbb{Q} \\
\mathcal{F} & ::= & \overline{x\ \texttt{=}\ \lambda y.t}
\end{array}
$$

Figure 1. MCL basic syntax

mutually recursive functions in $\mathcal{F}$). Values $v \subseteq t$ are the evaluated normal forms. Builtin primitives b are booleans, rational numbers, integers and strings. The corresponding binary operators are summarized in $\circ$.

A value field can be accessed from a class using a notation similar to the class-selection: A reference $r \triangleq h_1\texttt{.super(}h_2\texttt{).}l$ refers to the field labeled with l in class h_2 extended by h_1. Again we allow the convenient abbreviation $h\texttt{.}l \triangleq h\texttt{.super(}h\texttt{).}l$

Capture-avoiding substitution of variables by a partial function p is written as $[p]t$. The usual conditions for freshness of bound variables have to apply to the codomain of the partial function. Substitutions do not pass over references, i.e. $[p]r \triangleq r$. We write $[\![t]\!]_{\text{fv}}$ to denote the set of free variables in a term. Variables are bound by abstraction and the mutually recursive functions (plus their arguments) of a fixed-point. In all other cases, the set of free variables is the union of the free variables of all sub terms.

A context is a term with a "hole" into which another term is plugged. This hole is expressed as a special variable $\blacksquare$. By convention $\blacksquare$ is never bound in any term. Plugging a term t into a context s is then obtained via substitution $[\blacksquare \mapsto t]s$.

4 The Hierarchical Environment

The semantics of hierarchical terms can be seen as the reduction to a normal form of evaluated classes. We will motivate this normal form by a somewhat informal interpretation of the process. For reasons that will become clear in a moment, call an evaluated class a *node* (expressed by the syntactic sort v). Nodes are created as the combination of a literal class C with an environment.

The environment *inside* a node has one additional entry, mapping 0 to the node itself. All other entries link back to the original environment (just one level higher). In a certain sense, this definition forms an inverted view of the syntax tree, as each node gives access to an ordered set of children (which may be its parents in the syntax tree). Environments are forests of such trees and literal classes are node labels. (Hence the name *node* for the elements of this structure.)

Evaluation of hierarchical terms can be defined by the resolution of special variables and the three mutually recursive operations, *selection*, *search* and *evaluation*. During evaluation, a special variable i is *resolved* in a given environment $\mathcal{E}$ to the i-th entry of the environment.

$$
\begin{array}{ll}
[\![\mathcal{E}, \texttt{up}(n)]\!]_{eval} & \triangleq\ \mathcal{E}(n) \\
[\![\mathcal{E}, h\texttt{.}L]\!]_{eval} & \triangleq\ [\![[\![\mathcal{E}, h]\!]_{eval}, L]\!]_{select} \\
[\![\mathcal{E}, C]\!]_{eval} & \triangleq\ \texttt{\{|}C\ \texttt{in}\ \mathcal{E}\texttt{|\}} \\
[\![\mathcal{E}, t]\!]_{eval} & \triangleq\ \ldots
\end{array}
$$

Composite names (e.g. `up(2).z`) are evaluated from left to right by *selecting* the label. Evaluating a literal class with a given environment yields a context by appending that literal class to the current environment. We ignore the evaluation of proper terms for now.

$$
\begin{array}{ll}
[\![v, L]\!]_{select} & \triangleq\ [\![[\![v']\!]_{env} \oplus \{0 \mapsto v\}, h]\!]_{eval} \\
\text{when} & [\![v, L]\!]_{search} \triangleq v', h
\end{array}
$$

In order to *select* a field from a class, its definition has to be found in the class itself or in a super class. The resulting term is then evaluated under a new environment (obtained via *env* from v'). By setting the 0-th environment entry to the receiver, the special variable `this` is given a new meaning. If the definition is found in the receiver itself, i.e. $v \triangleq v'$, the change has no effect.

$$
[\![v, L]\!]_{search} \triangleq \begin{cases} v, h & \text{if}\quad L\ \texttt{=}\ h \in [\![v]\!]_{class} \\ v', h' & \text{if}\quad v\ \text{extends}\ h_S \\ & \qquad [\![h, [\![v]\!]_{env}]\!]_{eval} \triangleq v_S \\ & \qquad [\![v_S, L]\!]_{search} \triangleq v', h' \end{cases}
$$

A definition is *searched* recursively: If the field is a literal child, its right hand side is searched. Otherwise, the super classes of the context are evaluated and search continues there.

4.1 Graphical Interpretation

As an example, consider the classes `C` and `D` from above and the evaluation of `D.x`. Since all classes in that example have a unique name, this name is used in abbreviations as

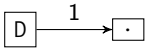
DOI 10.3384/ecp17132703

a subscript, i.e. v_C is the obtained from evaluating class `C`, $\mathcal{E}_C$ is its environment and C_C denotes its literal class.

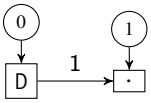
To support the interpretation of class values as nodes in a tree, they are drawn as directed graphs. Each node can be interpreted as such a directed graph with edge labels to indicate the ordering of its enclosing classes. A node label indicates the literal class. The order of nodes in an environment is drawn as graphs with two kinds of vertices. Each entry is represented by a circular node that is labeled with a natural number and has exactly one outgoing edge to the corresponding element of the environment. The top-level environment is drawn as follows:

$$\mathcal{E}_{root} \triangleq \quad \text{  } \tag{1}$$

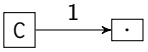
Creating new nodes from a literal classes inside an environment is achieved by adding a new node (labeled with the literal class), replacing the special nodes with edges from that node to their respective target and increasing their label by one:

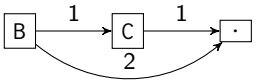
$$\texttt{\{|}C_D\ \texttt{in}\ \mathcal{E}_{root}\texttt{|\}} \triangleq v_D \triangleq \quad \text{  } \tag{2}$$

A node implies an environment on its immediate children (function *env*). This environment is obtained by mapping the node itself to 0 and adding an entry for each outgoing edge:

$$[\![v_D]\!]_{env} \triangleq \mathcal{E}_D \triangleq \quad \text{  } \tag{3}$$

Classes `C` and `B` are evaluated in a similar style to `D` in 2:

$$v_C \triangleq \texttt{\{|}C_C\ \texttt{in}\ \mathcal{E}_{root}\texttt{|\}} \triangleq \quad \text{  } \tag{4}$$

$$v_B \triangleq \texttt{\{|}C_B\ \texttt{in}\ \mathcal{E}_C\texttt{|\}} \triangleq \quad \text{  } \tag{5}$$

Using these classes, a simple evaluation can be processed as follows:

$$\begin{aligned}[\![v_C, \texttt{y}]\!]_{select} &\triangleq [\![\mathcal{E}_C, \texttt{up(0).B.z}]\!]_{eval} \\ &\triangleq [\![[\![v_C, \texttt{B}]\!]_{select}, \texttt{z}]\!]_{select} \\ &\triangleq [\![v_B, \texttt{z}]\!]_{select} \triangleq [\![\mathcal{E}_B, 21]\!]_{eval} \triangleq 21\end{aligned} \tag{6}$$

Similarly, the super class of `D` can be evaluated:

$$\begin{aligned}[\![\mathcal{E}_D, \texttt{up(1).C.B}]\!]_{eval} &\triangleq [\![[\![v_{root}, \texttt{C}]\!]_{select}, \texttt{B}]\!]_{select} \\ &\triangleq [\![v_C, \texttt{B}]\!]_{select} \triangleq v_B\end{aligned} \tag{7}$$

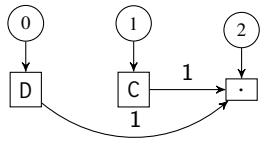
With these observations it is now possible to understand the evaluation of `D.x`. In a first step, the field is found in the node itself:

$$[\![v_D, \texttt{x}]\!]_{search} \triangleq v_D, \texttt{this.A.x} \tag{8}$$

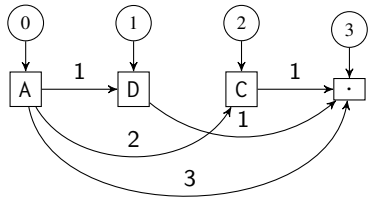
There is no field `A` in C_D. The search continues in the super class, v_B. Here, the field is found:

$$\begin{aligned}[\![v_D, \texttt{A}]\!]_{search} &\triangleq [\![v_B, \texttt{A}]\!]_{search} \quad \text{by (7)} \\ &\triangleq v_B, C_A\end{aligned} \tag{9}$$

C_A is found in the super class v_B. The evaluation environment is constructed from both the super class as well as the extending class:

$$\mathcal{E}_S \triangleq [\![v_B]\!]_{env} \oplus \{0 \mapsto v_D\} \triangleq \quad \text{  } \tag{10}$$

Evaluation of the literal class then appends C_A to the nodes of the environment. This node in turn has its own environment as usual:

$$\mathcal{E}_{AS} \triangleq [\![v_{AS}]\!]_{env} \triangleq \quad \text{  } \tag{11}$$

Resolving `up(2)` and `up(1)`in $\mathcal{E}_{AS}$ yields v_C and v_D, respectively. Hence, `y` and `z` can be selected:

$$\begin{aligned}[\![\mathcal{E}_{AS}, \texttt{up(2).y}]\!]_{eval} &\triangleq [\![v_C, \texttt{y}]\!]_{select} \quad \text{by (11)} \\ &\triangleq 21 \quad \text{by (6)}\end{aligned} \tag{12}$$

$$\begin{aligned}[\![\mathcal{E}_{AS}, \texttt{up(1).z}]\!]_{eval} &\triangleq [\![v_D, \texttt{z}]\!]_{select} \quad \text{by (11)} \\ &\triangleq [\![\mathcal{E}_D, 2]\!]_{eval} \triangleq 2\end{aligned} \tag{13}$$

This demonstrates two important aspects: First, the modification in C_D has affected the evaluation as intended. And second, the inheritance is "forgotten", when the lookup passes the correspoding lexical scope. The evaluation of `y` takes place in v_C without further changes. In conclusion, the result is 23, as expected:

$$\begin{aligned}[\![v_D, \texttt{x}]\!]_{select} &\triangleq [\![\mathcal{E}_D, \texttt{this.A.x}]\!]_{eval} && \text{by (8)} \\ &\triangleq [\![[\![v_D, \texttt{A}]\!]_{select}, \texttt{x}]\!]_{select} && \text{by (3)} \\ &\triangleq [\![[\![\mathcal{E}_S, C_A]\!]_{eval}]\!]_{select} && \text{by (9)} \\ &\triangleq [\![\mathcal{E}_{AS}, \texttt{up(2).y} + \texttt{up(1).z}]\!]_{eval} && \text{by } C_A \\ &\triangleq [\![\mathcal{E}_{AS}, 21 + 2]\!]_{eval} \triangleq 23 \quad \square && \text{by (12,13)}\end{aligned}$$

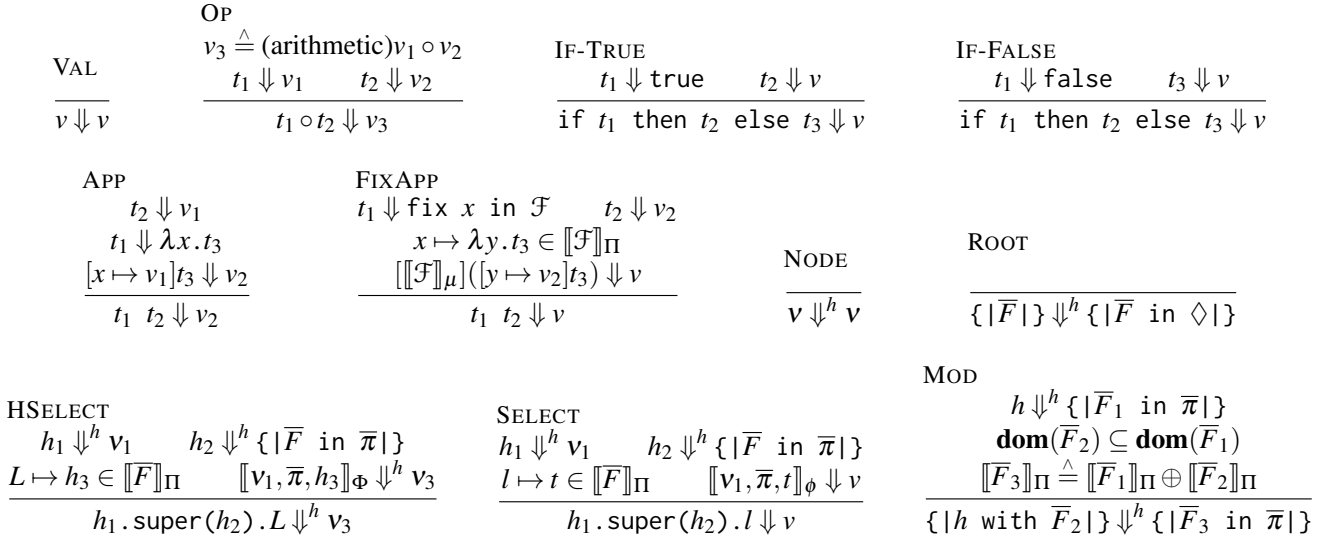


Figure 2. Evaluation semantics

4.2 Dynamic Semantics

Figure 2 depicts the rules of the dynamic semantics of MCL in big-step or natural(Kahn 1987) style. A term t evaluates to a value v, iff both are related by a reduction relation $t \Downarrow v$. Erroneous terms are identified by not being related to some value. All elements of $\Downarrow$ are defined inductively by inference rules.

In order to simplify the notation of sequential constructs, it is useful to define a mapping between concrete syntax and partial functions. Each sequence can be seen as a partial function, mapping its left-hand elements to the corresponding right-hand side. This conversion is implemented with the function Π.

$$
\begin{aligned}
[\![\mathcal{F}]\!]_\Pi &\triangleq \{x_1 \mapsto \lambda y_1 . t_1\} \oplus \ldots \oplus \{x_n \mapsto \lambda y_n . t_n\} \\
\text{where} \quad \mathcal{F} &\triangleq x_1 \texttt{ = } \lambda y_1 . t_1, \ \ldots, x_n \texttt{ = } \lambda y_n . t_n \\
[\![\Diamond]\!]_\Pi &\triangleq \emptyset \\
[\![l \texttt{ = } t :: \overline{F}]\!]_\Pi &\triangleq [\![\overline{F}]\!]_\Pi \oplus \{l \mapsto t\} \\
[\![L \texttt{ = } h :: \overline{F}]\!]_\Pi &\triangleq [\![\overline{F}]\!]_\Pi \oplus \{L \mapsto h\} \\
[\![\mathcal{F}]\!]_\mu &\triangleq \{x_i \mapsto \texttt{fix } \mathcal{F} \texttt{ in } x_i \mid x_i \in \textbf{dom}([\![\mathcal{F}]\!]_\Pi)\}
\end{aligned}
$$

Rule APP is the standard application via substitution. OP implements binary operators on builtin primitives; it is actually a family of rules with one element for each builtin operator. Rules IF-FALSE and IF-TRUE implement non-strict conditionals.

Mutually recursive functions $\mathcal{F} \triangleq x_i = \lambda y_i . t_i$ are implemented via the explicit fixed point term `fix` x `in` $\mathcal{F}$, the special function μ and rule FIXAPP. In order to evaluate a recursive function, first the argument has to be evaluated. This argument is then substituted into the body of the function, followed by a substitution of the group itself, as defined by μ. Due to the nature of natural semantics, divergence cannot be distinguished from a stuck term.

The hierarchical semantics of MCL is embedded into the proper evaluation (but not vice-versa). In a certain sense, classes play the role of modules. Evaluation of a hierarchical term h to a hierarchical class $v \triangleq \{|\overline{F} \texttt{ in } \overline{\pi}|\}$ with parents $\pi_1 \ldots \pi_n$ is written $h \Downarrow^h v$. Rule SELECT augments the evaluation relation $\Downarrow$. Hierarchical nodes are already in normal form (rule NODE). An empty literal class evaluates to a root node (rule ROOT).

4.2.1 Selections and Inheritance

Selecting a child class via HSELECT or SELECT relies on the search of the corresponding definition. This is implemented by a partial function from labels to hierarchical terms. Depending on the context, either a class label L or a value label l is looked up. Notably, this definition of the search operation is not recursive. It relies on the encoding of inherited fields as references from `this` to the corresponding super class.

MCL does not allow for unqualified inheritance of names: Instead of a single **`extends`** statement, all inherited fields have to be explicitly present in the base class. The definition then forwards to the super class with the second argument of the reference:

```
A = this.super(up(1).Y).A;
a = this.super(up(1).Z).a;
```

In the example above, class `A` and the value `a` are inherited from classes `Y` and `Z`, which are found in the outer scope. The delegation is resolved by either HSELECT or SELECT. Multiple levels of inheritance are then expressed by a chain of such delegations. This style decouples the set of inherited elements from the definitions in the super class and allows for a more selective approach (e.g. it becomes possible to express the resolution of multiple inheritance of fields with the same name).

DOI 10.3384/ecp17132703

$$
\begin{array}{lll}
[\![\_,\_,v]\!]_\Phi & \triangleq v & [\![v,\overline{\pi},\{|h \texttt{ with } \overline{F}|\}]\!]_\Phi \triangleq \{|[\![v,\overline{\pi},h]\!]_\Phi \texttt{ with } f \times \overline{F}|\} \\
[\![v,\overline{\pi},\texttt{up}(0)]\!]_\Phi & \triangleq v & \quad \text{where} \quad [\![L = h]\!]_f \triangleq L = [\![v,\overline{\pi},h]\!]_\Phi \\
[\![\_,\overline{\pi},\texttt{up}(n+1)]\!]_\Phi & \triangleq \pi_{n+1} & [\![l = t]\!]_f \triangleq l = [\![v,\overline{\pi},t]\!]_\phi \\
[\![\_,\Diamond,\texttt{up}(n+1)]\!]_\Phi & \triangleq \texttt{up}(n+1) & [\![v,\overline{\pi},h_1.\texttt{super}(h_2).L]\!]_\Phi \triangleq [\![v,\overline{\pi},h_1]\!]_\Phi.\texttt{super}([\![v,\overline{\pi},h_2]\!]_\Phi).L \\
[\![v,\overline{\pi},\{|\overline{F}|\}]\!]_\Phi & \triangleq \{|\overline{F} \texttt{ in } v :: \overline{\pi}|\} & [\![v,\overline{\pi},h_1.\texttt{super}(h_2).l]\!]_\phi \triangleq [\![v,\overline{\pi},h_1]\!]_\Phi.\texttt{super}([\![v,\overline{\pi},h_2]\!]_\Phi).l
\end{array}
$$

$$
[\![v,\overline{\pi},x]\!]_\phi \triangleq x \qquad [\![v,\overline{\pi},\lambda x.t]\!]_\phi \triangleq \lambda x.[\![v,\overline{\pi},t]\!]_\phi \qquad [\![v,\overline{\pi},t_1\ t_2]\!]_\phi \triangleq [\![v,\overline{\pi},t_1]\!]_\phi\ [\![v,\overline{\pi},t_2]\!]_\phi \quad \ldots
$$

Figure 3. The Fold Function

In order to adhere to the principle of open recursion between fields of a class, the special variables in a found term are resolved using the fold operators $\Phi : h \times \overline{v} \times h \to h$ and $\phi : h \times \overline{v} \times t \to t$ before evaluation. These mutually recursive functions (Figure 3) take three arguments: An evaluated class v represents the tip of the environment (i.e. the *self*-instance), $\overline{\pi}$ is the list of hierarchical parents of the super class containing the definition of the current term, and the third argument is the input that is being folded. Φ expects and returns an hierarchical term h while ϕ works on plain terms t.

In the case of plain terms, the result of folding is simple: ϕ is applied on the sub terms or returns its input unchanged if the argument is primitive. Value references are processed by folding the hierarchical sub terms with Φ.

Folding hierarchical terms resolves the special variables `up(`i`)` (thus implementing the environment directly via substitution): The special variable `up(0)` (the `this`-variable) is replaced with the *self*-instance (the first entry of the environment). Other special variables are looked up accordingly. If the environment is empty, the result is left unresolved. References and modified classes are folded by folding their corresponding sub terms. In the case of modifications this ensures that a modified field is evaluated in the context of the modification site (and not in the context of the modified class). Literal classes are turned into nodes by storing the environment alongside their fields. Contrary to modifications, their fields are not subject to further folding. This ensures that child classes retain their own context, when a field is selected from that child class. Nodes are left unchanged by the fold function.

4.3 Modifications and Redeclarations

MCL supports both redeclarations and modifications. The former are implemented via overriding of inherited methods (thus, there all fields are considered replaceable). Modifications differ from overriding in their scope — modifications live outside of the modified class. Each modified field must exist in the modified class. It is not possible to *add* a field via a modification. The modification of a class h with a sequence of fields $\overline{F}$ results in a class containing the modified fields merged with the result of the evaluation (rule MOD). Merging is implemented by lifting the fields into partial functions, augmenting the original function with the new fields and lowering the result into a sequence of fields.

5 Translation of References

The specification of Modelica require the evaluation of names only when necessary; i.e. the lookup of classes, functions, types and variables is always driven by the attempt to flatten a particular class. We take a slightly different stance, and demand that all references can be looked up strictly. The goal is to replace all references (inside a certain term) with their definitions (and transitively all references in them). The resulting term is then free of any hierarchical references and can be evaluated as usual. This technique allows to consider lookup and flattening as completely separate parts of the semantics (and gives reason to consider the former as part of the static semantics).

5.1 Evaluation of Hierarchical Terms

The definition of $\Downarrow^h$ is algorithmic, a naive implementation will however not always terminate due to the open recursion. In particular, evaluating the subterms of a class reference might require evaluation of the same class reference:

```
{| class A = this; x = this.A.x |}.x
```

In the example above, a naive interpreter will repeatedly attempt to evaluate the class reference `this.A.x`. This is not a particularity of MCL, as the following example shows:

```
class A
    model B extends C; end B;
    model C extends B; end C;
    B.Foo b;
end A;
```

This simple model cannot be flattened (as there is no class definition for the component `b`). Yet, the attempt drives the leading free implementations OpenModelica (in version 1.11.0) and JModelica (version 1.17) into an endless loop, eventually ended by a stack overflow. In a realistically sized model, the user can only speculate what causes such a crash and, should the relevant loop be optimized to a tail-recursive implementation, might not even encounter a crash but a “frozen” implementation.

Thus it is necessary to restrict the computation of references in a way that *guarantees* termination and retains a valid result for a meaningful subset of the terminating nodes. It is hardly constructive to reject *all* recursive relations between classes, as for instance recursive functions would fall under the same rule (functions are specialized classes in Modelica). Instead, the restriction should only prevent divergence during lookup. This can be achieved by only attempting to evaluate an hierarchical reference once for any given environment.

We assume that each literal class is labeled with a unique identifier from a set $\mathbb{L} \subseteq \mathbb{N}$. We write $\overline{F}_i$ to indicate a class with id i. The *syntactic depth* of a literal class is the number of syntactically visible enclosing classes. It is easy to see that this number is invariant during evaluation (otherwise, special variables might be invalidated). Each node is only valid when it has *precisely* the correct amount of enclosing classes for its literal class. A node that fulfills this requirement is called *context correct*. The set of context correct nodes is not finite, though. Consider two distinct literal classes $\overline{F}_1$ and $\overline{F}_2$ with depths 0 and 1, then $\{|\overline{F}_2 \texttt{ in } \{|\overline{F}_1 \texttt{ in } \Diamond|\}|\}$ is context correct. But so is $\{|\overline{F}_2 \texttt{ in } \{|\overline{F}_2 \texttt{ in } \{|\overline{F}_1 \texttt{ in } \Diamond|\}|\}|\}$ and so on. Obviously, hierarchies with the repeated occurance of the same literal class are problematic. It is thus necessary to find a syntactic criterion to rule out such strange loops.

The directed graphs used in Section 4 can be formalized as directed multigraphs (V,E) with vertices $V \subseteq \mathbb{L}$ represented by the labels of literal classes and edges as triples of one outgoing and one incoming vertex together with a natural number $E \subseteq \mathbb{L} \times \mathbb{L} \times \mathbb{N}$. The identity of an edge is defined by its source, its destination and its number. The usual terms from (multi) graph theory (reachability, cycles, etc.) apply.

Definition 1 (Graph Representation of Nodes and Environments)**.** *The directed multigraph of a node is the vertex labeled with the literal class of the node linked to the graphs of all parents by ordered edges. The graph of an environment (i.e. a list of nodes) is the union of the graphs of each node (where the union of graphs is the union of their components).*

$$
\begin{aligned}
[\![\{|\overline{F}_u \texttt{ in } \overline{\pi}|\}]\!]_{gr} &\triangleq (\{u\} \cup V, P \cup E) \\
\text{where} \qquad P &\triangleq \{(u, p_i, i) \mid p_i \triangleq [\![\pi_i]\!]_L\} \\
(V,E) &\triangleq \bigcup_{i \in 1 \ldots |\overline{\pi}|} [\![\pi_i]\!]_{gr}
\end{aligned}
$$

The set of possible results of this transformation is finite, when both the set of labels and edges are finite. Both conditions are trivially fulfilled by graphs created from context correct nodes, since each node is in itself a finite structure, the syntactic depth of each node is limited by the syntactic structure of the source program, and each source program is labeled by a finite set of labels.

Multigraphs that do not contain any cycles and have a distinguished root node can be unambiguously transformed into a node, when the outgoing edges of a each node are labeled consecutively with the numbers ranging from 1 to the depth of the corresponding literal class. Such a graph is said to be context correct. This transformation is bijective.

Definition 2 (Admissible Lookups)**.** *A node is admissible, iff its graph representation is a context correct, rooted multigraph. An environment $\overline{\pi}$ is admissable, iff all contained nodes are admissable and the lookup of a label L in an environment is admissible iff the environment is admissible:*

$$
\begin{aligned}
\nu \text{ admissible} &\iff [\![\nu]\!]_{gr} \text{is rooted and context correct} \\
\overline{\pi} \text{ admissible} &\iff \forall i \in 1 \ldots |\overline{\pi}| \;\; \pi_i \text{ admissible} \\
\langle \nu :: \overline{\pi} \cdot L \rangle \text{ admissible} &\iff \nu \text{ admissible} \wedge \overline{\pi} \text{ admissible}
\end{aligned}
$$

If all nodes are rejected that do not meet these simple criteria, the set of admissible nodes is finite. This allows to evaluate any hierarchical term without in a finite amount of steps (by checking for repetitions). As a side effect, all strange loops (i.e. classes that contain themselves) are ruled out, but classes that merely *refer* to each other are still allowed.

Lemma 1 (Finiteness of Admissible Lookups)**.** *For any given finite labeling of literal classes, and a finite maximal depth of classes the set of admissible lookups is finite. admissable nodes is finite.*

Proof. Admissible nodes are finite due to their injective mapping to a context correct, rooted multigraph over the (finite) labeled vertices. Admissible (finite) environments and lookups are products of finite sets. □

Evaluation of hierarchical terms can be implemented in a terminating, total function $\mathcal{H}$. This function follows the definition of $\Downarrow^h$ by construction. The sole difference lies in the "memory" G, a set of admissible lookups. No lookup is ever repeated, hence the function terminates.

$$
\begin{aligned}
[\![G, \nu]\!]_{\mathcal{H}} &\triangleq \nu \\
[\![G, \{|\overline{F}|\}]\!]_{\mathcal{H}} &\triangleq \{|\overline{F} \texttt{ in } \Diamond|\} \\
[\![G, \{|h \texttt{ with } \overline{F_2}|\}]\!]_{\mathcal{H}} &\triangleq \{|\overline{F}_3 \texttt{ in } \overline{\pi}|\} \\
\quad \text{if} \quad & [\![G,h]\!]_{\mathcal{H}} \triangleq \{|\overline{F_1} \texttt{ in } \overline{\pi}|\} \\
& \mathbf{dom}(\overline{F}_2) \subseteq \mathbf{dom}(\overline{F}_1) \\
& [\![\overline{F}_3]\!]_{\Pi} \triangleq [\![\overline{F}_1]\!]_{\Pi} \oplus [\![\overline{F}_2]\!]_{\Pi} \\
[\![G, h_1\texttt{.super(}h_2\texttt{).}L]\!]_{\mathcal{H}} &\triangleq [\![G', [\![\nu_1, \overline{\pi}, h_L]\!]_{\Phi}]\!]_{\mathcal{H}} \\
\quad \text{if} \quad & [\![G,h_1]\!]_{\mathcal{H}} \triangleq \nu_1 \\
& [\![G,h_2]\!]_{\mathcal{H}} \triangleq \{|\overline{F} \texttt{ in } \overline{\pi}|\} \\
& L \mapsto h_L \in [\![\overline{F}]\!]_{\Pi} \\
& \langle \nu_1 :: \overline{\pi} \cdot L \rangle \text{ admissible} \notin G \\
& G' \triangleq G \cup \{\langle \nu_1 :: \overline{\pi} \cdot L \rangle\} \\
[\![G,h]\!]_{\mathcal{H}} &\triangleq \lightning \qquad \text{in any other case}
\end{aligned}
$$

DOI 10.3384/ecp17132703

5.2 Lookup of References

The definition of $\mathcal{H}$ immediately yields an algorithm for the lookup of references, $\mathcal{L}$. A lookup is successful if both the super class and the base class can be evaluated successfully, the resulting environment is admissable and it contains a matching element. The result of a successful lookup maps the environment and looked up label to the folded result term. An error is indicated by the mark $\lightning$.

$$
\begin{aligned}
[\![h_1\,.\,\texttt{super(}h_2\texttt{)}\,.\,l]\!]_{\mathcal{L}} \quad &\triangleq\ \langle v_1 :: \overline{\pi} \cdot l\rangle \mapsto [\![v_1, \overline{\pi}, t]\!]_{\phi} \\
\text{if} \quad & [\![\emptyset, h_1]\!]_{\mathcal{H}} \triangleq v_1 \\
& [\![\emptyset, h_2]\!]_{\mathcal{H}} \triangleq \texttt{\{|}\overline{F}\ \texttt{in}\ \overline{\pi}\texttt{|\}} \\
& l \mapsto t \in [\![\overline{F}]\!]_{\Pi}
\end{aligned}
$$

$$
[\![r]\!]_{\mathcal{L}} \quad \triangleq\ \lightning \qquad \text{otherwise}
$$

This allows to to lookup *all* references in a term, including those references that occur transitively as the result of a successful lookup. Such an exhaustive search is achieved by repeated applications of a one-step search function $\mathcal{G}$ to an intermediate result set R:

$$
[\![\lightning]\!]_{\mathcal{G}} \quad \triangleq\ \lightning
$$

$$
[\![R]\!]_{\mathcal{G}} \quad \triangleq\ \begin{cases} \lightning & \text{if } \exists [\cdot \mapsto r]s \in \mathbf{img}(R) \text{ s.t. } [\![r]\!]_{\mathcal{L}} \triangleq \lightning \\ R \cup \{\, [\![r]\!]_{\mathcal{L}} \mid [\cdot \mapsto r]s \in \mathbf{img}(R) \,\} & \text{otherwise} \end{cases}
$$

The exhaustive search terminates, when a fixed point is reached. This is guaranteed due to the finite set of admissible environments. $\mathcal{G}$ is also *inflationary*. This guarantees the existence of the conditional fixed point starting from a set R (see Pepper and Hofstedt 2006, chapter 10).

Lemma 2 (Fixed Point of $\mathcal{G}$)**.** *The ascending Kleene chain of $\mathcal{G}$ has a least fixed point.*

Proof. The partial functions (and error marker) obtained by $\mathcal{G}$ form a complete partial order (*cpo*) under the subset relation, i.e. $R_1 \leq R_2 \Longleftrightarrow R_1 \subseteq R_2$ with $\lightning$ as top element, i.e. $R \leq \lightning$, because the set of admissible environments (the domain of each R) is finite and adding a top element to a cpo yields a cpo. Function $\mathcal{G}$ is also Scott-continuous (the least upper bound of any chain is the set-union in the absence of errors and the error otherwise). □

5.3 Transformation

A reference must be evaluated in order to look up its corresponding definition. This does not introduce any errors, if the underlying search result is indeed a fixed point, though (as all contained references have already been evaluated at least one). The definition of a reference might (after several steps of lookup) depend on the reference itself. This implicit recursion has to be transformed into a proper fixed point. In order to do so, all definitions have to be regarded as functions, since MCL does not allow for any other recursive definitions. This is easily achieved by wrapping them into a "thunk" (a function taking an unused argument).

Function $\mathcal{C}$ maps each found definition to a structurally similar term where all references are replaced with their corresponding name. It is assumed that the lookup result is arbitrarily ordered.

Definition 3 (Transformation)**.** *The transformation replaces all references with a recursive function that is obtained by the lookup closure of its definition. The closure replaces each references with the name of its definition.*

$$
\begin{aligned}
[\![R, t]\!]_{\mathcal{C}} \quad &\triangleq\ t \qquad \text{if } r \notin t \\
[\![R, [\cdot \mapsto r]t]\!]_{\mathcal{C}} \quad &\triangleq\ [\cdot \mapsto x_i\ \ 0][\![R, t]\!]_{\mathcal{C}} \\
\text{where} \quad & R \triangleq \{ L_1 \mapsto t_1, \ldots, L_n \mapsto t_n \} \\
& [\![r]\!]_{\mathcal{L}} \triangleq L_i \mapsto t_i \\
& \{x_1 \ldots x_n\} \text{ fresh in } \mathbf{img}(R) \\
[\![R]\!]_{\mathcal{C}} \quad &\triangleq\ \{ x_i \mapsto \lambda y\,.\,t_i \mid t_i \in \mathbf{img}(R), y \notin [\![t_i]\!]_{fv} \}
\end{aligned}
$$

$$
\begin{aligned}
[\![t]\!]_{\gamma} \quad &\triangleq\ t \qquad \text{if } r \notin t \\
[\![[\cdot \mapsto r]s]\!]_{\gamma} \quad &\triangleq\ [\cdot \mapsto (\texttt{fix}\ x_r\ \texttt{in}\ \mathcal{F}_R)\ 0][\![s]\!]_{\gamma} \\
\text{if} \quad & [\![\mathcal{F}_R]\!]_{\Pi} \triangleq [\![R]\!]_{\mathcal{C}} \\
\text{and} \quad & [\![R]\!]_{\mathcal{G}} \triangleq R \\
\text{and} \quad & R \triangleq [\![\{ \langle v :: \overline{\pi} \cdot l\rangle \mapsto t \}]\!]_{\mathcal{G}^n} \\
\text{and} \quad & [\![r]\!]_{\mathcal{L}} \triangleq \langle v_1 :: \overline{\pi}_1 \cdot l_1\rangle \mapsto t_1 \\
[\![[\cdot \mapsto r]s]\!]_{\gamma} \quad &\triangleq\ \lightning \qquad \text{otherwise}
\end{aligned}
$$

5.4 Example

Our running example can be encoded in MCL as v_{root}`.D.x`. For this term, the exhaustive lookup yields the result:

$$
\begin{aligned}
R \triangleq \{\ & \\
& \langle v_D :: \mathcal{E}_{root} \cdot \texttt{x}\rangle \mapsto [\![v_D, \mathcal{E}_{root}, \texttt{this.A.x}]\!]_{\phi}, \\
& \langle v_A :: \mathcal{E}_S \cdot \texttt{x}\rangle \mapsto [\![v_A, \mathcal{E}_S, \texttt{up(2).y} + \texttt{up(1).z}]\!]_{\phi}, \\
& \langle v_C :: \mathcal{E}_{root} \cdot \texttt{y}\rangle \mapsto [\![v_C, \mathcal{E}_{root}, \texttt{this.B.z}]\!]_{\phi}, \\
& \langle v_D :: \mathcal{E}_{root} \cdot \texttt{z}\rangle \mapsto [\![v_D, \mathcal{E}_{root}, 2]\!]_{\phi}, \\
& \langle v_B :: \mathcal{E}_C \cdot \texttt{z}\rangle \mapsto [\![v_B, \mathcal{E}_c, 21]\!]_{\phi} \}
\end{aligned}
$$

After closing this complete result, the translation yields:

```
(x0 in fix
   x0 = y. (x1 0) ;
   x1 = y. (x2 0) + (x3 0) ;
   x2 = y. (x4 0) ;
   x3 = y. 2 ;
   x4 = y. 23 ;
) 0
```

This term then evaluates to `23`, as expected.

5.5 Correctness

The correctness of γ depends on the closure of a term by a complete lookup result. The most important step is an observation about a symmetry between the evaluation of a term containing hierarchical references and that of a fixed point constructed from the lookup result R of that term: Both evaluations only differ in the presence or absence of rule SELECT, which is replaced by specific instances of FIXAPP (with a group of recursive definitions generated from R).

Lemma 3 (Correctness of $\mathcal{C}$)**.** *The closure of a complete lookup result is equivalent to the lookup of references. When R is complete, i.e. $[\![R]\!]_{\mathcal{G}} \stackrel{\wedge}{=} R \stackrel{\wedge}{\neq} \lightning$, a term t appears on the image of R, i.e. $\langle v :: \overline{\pi} \cdot l \rangle \mapsto [\cdot \mapsto t]s \in R$, and t evaluates with n applications of rule* SELECT*, i.e. $[\![v, \overline{\pi}, t]\!]_\phi \Downarrow_{n-\text{SELECT}}$ Then the application of R as a fixed point evaluates to an equivalent result n applications of rule* FIXAPP *to R:*

$$[[\![R]\!]_{\mu \circ \mathcal{C}}][\![R, t]\!]_{\mathcal{C}} \Downarrow_{n-\text{FIXAPP}-R} [\![R, v]\!]_{\mathcal{C}}$$

Proof. By natural induction over n. The base step ($n = 0$) follows by a straightforward induction over $\Downarrow$, since SELECT is not applied in the derivation. The inductive step also requires a nested induction over $\Downarrow$. In the case of $t \stackrel{\wedge}{=} r$, the completeness of R is used to apply one step of rule FIXAPP (and thus the outer induction hypothesis). □

The correctness of the overall transformation is defined as the preservation of the semantics of the transformed term: When a term evaluates and the transformation yields no error, then the transformed term yields a value that is equal to the transformation of the original result.

Theorem 4 (Correctness of γ)**.** *The transformation γ preserves the semantics of terms.*

$$t \Downarrow v \wedge [\![t]\!]_\gamma \stackrel{\wedge}{=} s \Longrightarrow s \Downarrow [\![v]\!]_\gamma$$

Proof. By induction over t. The fundamental case is $t \stackrel{\wedge}{=} r$. By construction of γ, $[\![r]\!]_{\mathcal{L}} \stackrel{\wedge}{=} \langle v :: \overline{\pi} \cdot l \rangle \mapsto [\![v, \overline{\pi}, s]\!]_\phi \in R$. Inversion of the evaluation yields $[\![v, \overline{\pi}, s]\!]_\phi \Downarrow v$. The conclusion then follows via rule FIXAPP and Lemma 3. □

6 Discussion

We have given a definition of an explicit, context-independent syntax and semantics for the lookup of names in Modelica classes. Classes (hierarchical terms) can be translated by a terminating evaluation of all references. This translation maintains the original semantics.

6.1 Related Work

The semantics of Modelica has been subject to surprisingly little research. The work of Kågedal (1998), has a much broader scope. It does however not discuss open recursion nor redeclarations and is considerably outdated when it comes to modern Modelica. Satabin et al. (2015) use a style comparable to ours, but favor a global environment (called class table) over our substitution based approach. Interestingly, they also notice the difficulty to separate the *static* semantics of a model from its dynamics, but solve this problem by restricting their input language. In particular, no short class definitions or redeclarations are considered. It is also somewhat unclear if their approach allows for the late binding of modifications. Despite these differences, the presented techique may solve the open question of how to obtain the values for our special variables in the first place.

6.2 Conclusion

The definition of Modelica's hierarchical elements by special variables allows to express their static semantics. Treating classes and their interactions like modules with open recursion allows for a precise reasoning of the outcome of redeclarations and modifications. Last but not least, the difficulties that come with the uniform treatment of classes and components are now obvious and might have an influence on the design of future versions of Modelica. The *correct* translation of hierarchical references in a *terminating* process while maintaining the semantics of inheritance, modifications and redeclarations is a feature that, to our knowledge, has not been solved before. It allows a clear separation between the static and dynamic semantics of names in Modelica.

References

Aldrich, Jonathan and Kevin Donnelly (2004). "Selective open recursion: Modular reasoning about components and inheritance". In: *SAVCBS 2004 Specification and Verification of Component-Based Systems*, p. 26.

Höger, Christoph (2016). "Modeling with monads: extensible modeling semantics as syntactic sugar". In: *Proceedings of the 7th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*. ACM, pp. 15–24.

Kågedal, David (1998). "A Natural Semantics specification for the equation-based modeling language Modelica". In: *LiTH-IDA-Ex-98/48, Linköping University, Sweden*.

Kahn, Gilles (1987). "Natural semantics". In: *Annual Symposium on Theoretical Aspects of Computer Science*. Springer, pp. 22–39.

Pepper, Peter and Petra Hofstedt (2006). *Funktionale Programmierung – Sprachdesign und Programmiertechnik*. Springer.

Pierce, Benjamin C., ed. (2005). *Advanced Topis in Types and Programming Languages*. MIT Press.

Satabin, Lucas et al. (2015). "Towards a formalized Modelica subset". In: *Proceedings of the 11th International Modelica Conference, Versailles, France, September 21-23, 2015*. 118. Linköping University Electronic Press, pp. 637–646.