Fachhochschule für die Wirtschaft Hannover

– FHDW –

LOMF2

Participants:

HFP110 ::= { Thomas Bremer, Jonas Büth, Stefan Christian, Sönke Küper, Benjamin Rogge, Roman Smirnov }

HFP412 ::= { Alexander Bellhäuser, Björn Bodensieck, Malte Kastner, Thimo Koenig, Stefan Pietsch, Stefanie Rademacker, Marius Tempelmeier, Niklas Walter, Christin Weckbrod, Patrick Wolf }

HFP414 ::= { Henrik Herwig, Lisa Leitloff, Timo Lorenz, Marius Schultchen, Florian Selent, Sascha Sternheim, Patrick Stünkel }

Kindly supported by:

Prof. Dr. Michael Löwe

Version:

October 10, 2015

Abstract

This paper describes the second version of the programming language LOMF (Less Overhead, More Fun). The second version (LOMF2) is based on the concepts of category theory. The goal is to develop a strongly typed programming language that is an abstraction of today's commonly used programming languages (e. g. Java, C#, C etc.). Therefore the document gives an introduction into the syntax and semantics of the language. Furthermore the concepts for implementing the integrated development environment (IDE) are described within this document. Against the goal to create a more abstract language in comparison to commonly languages today, this document does not consequently compare LOMF2 with other languages directly. Instead it tries to implement all common constructs of actual languages such as specialization, sums (e. g. abstract types or anonymous ones like `throws A, B, C` in Java), products (e. g. structures, classes etc.), polymorphy (like in object oriented languages) and so on.

At first instance LOMF2 should be a language for teaching programming principles in the bachelor studies in the areas of informatics. So the chapter 2 should also give a short introduction in the syntax and semantics of the programming language. All other chapters deal with implementations of different parts like IDE, compiler, etc.

Contents

1 Introduction

This paper documents the process of LOMF2, which is developed by the master courses HFP110, HFP412 and HFP414 of the FHDW Hannover. Each course realized different parts of this project. Some examples are the following:

- User interface
- Syntax checker
- Type checker
- Evaluation
- Coercer
- Anonymous functions
- Built-in types/functions

To ensure the best development results and to allow the observation of the development process, the LOMF2 project is configured in the FHDW internal Jenkins Continuous Integration Server. Checkstyle[1] and findbugs[2] checks will be applied on every build for higher code quality. The test-coverage will be checked via Cobertura[3] to reflect the test quality as well as the general coverage. The Jenkins server observes the SVN directory and automatically triggers the build process on changes.

The LOMF2 sources and this documentation are open source and available from the FHDW internal SVN system via the address `https://fhdwdev.ha.bib.de/svn/lomf2`. The SVN directory is world-readable, write access can be requested via email.

[1] http://checkstyle.sourceforge.net
[2] http://findbugs.sourceforge.net
[3] http://cobertura.sourceforge.net

2 Language

At first we would like to take a closer look at the language of LOMF2. Therefore we begin introducing the specified syntax and conclude with the semantics in section 2.2.

2.1 Syntax

The syntax of LOMF2 is based on expressions known from mathematical esp. categorial theory constructs[4]. For example `[(A, B) --> C]` defines a signature of a function (morphism) which maps values of the product of the types (objects) `A` and `B` to a value of type `C`.

In LOMF2 it is basicly possible to define *types* and *functions* in expressions which end with a semicolon. Usually each type has a unique identifier (respectively a name) and a defining *constructor expression*. *Specialization* between types is possible in both directions: upwards by using the keyword `specializes` and downwards `generalizes`. Abstract types are marked with the keyword `abstract`. It is possible to define specializations and generalizations between non abstract types, then it is mandantory to specifiy a coercer as a *function expression* that converts between the two types (see below).

The following example for a type hierarchy shows the definition of the natural numbers `Card` with a `Null`-type and the successor type `Succ` which contains the previous number:

```
abstract type Card;
type Null () specializes Card;
type Succ (Card) specializes Card;
```

Beside the named types one can use anonymous types. Here we differentiate between *products* and *sums*. Products are defined between `(` and `)`. For sums { and } are used. Anonymous types are used in signatures and *function expressions*.

[4] A good introduction to category theory can be found in: (R. Goldblatt) Topoi - the Categorial Analysis of Logic, Elsevier 1984

With the specified types it is possible to define functions which operate on these types. Each concrete function has a name as *identfier*, a *signature* and a *function expression*. The signature consists of domain and codomain. In contrast to concrete functions, abstract functions ommit the *function expressions*.

Function expressions are usually serialized calls of functions or the creation of anonymous types. Each call or creation is separated by a whitespace character "␣". A call uses the previous result as input for the next call. For example let `f` be a function with signature `[A --> B]` and `g` a function with signature `[B --> C]`. If the expression `f g` is applied on an `A`-value, firstly `f` will applied on this value and pass its result (which is of type `B`) to `g`[5].

To extract one component of a product a *projection* is used. The syntax is "`.`" followed by a natural number (including zero) which defines the position of the component to be projected in the product e. g. `(A, B) .0` projects the first component `A`. To improve the usability dealing with projections, while the user haven't to remain the order of the components of the product, it is possible to name the components of a product type. The name have to be written in front of the type. The syntax is the name followed by "`:`".It is important, that the components of one product have different names. Projections with the position work also for the product types with named components. Furthermore it is possible to extract the components only with their name. In this case no "`.`" is needed. Additionally it is possible to name the components of an anonymous product. In this case the projections with the name of the components need to have an leading "`.`" before the name. The rest works in the same manner as the product types. But it is important to know, that names in anonymous products are only valid until the next "}". Furthermore LOMF2 supports execution of two or more calls concurrently. This is realized by a *product function expression* like `<f, g>`.

The following example shows the definition of the addition on the previously defined natural numbers.

```
function +[(Null, Card) --> Card] ::= .1;
function +[(Succ, Card) --> Succ] ::= ((.0.0, .1)+) Succ;
```

2.1.1 Grammar

For the context-free grammar, we use the following notation based on Backus-Naur form: (a) Non-terminal symbols are enclosed by `<` and `>`, terminal symbols are written between

[5]Note: The expression requires an input of type `A`

‘ and ’. (b) The symbol | denotes the choice (sum). (c) The suffix * means possible empty list of its prefix and $*^{(‘,’)}$ means possible empty list of items with ‘,’ as delimiter. (d) The symbol ? indicates optional occurrences. (e) If two symbols in a rule's right-hand side are separated by a blank, white-space is allowed or necessary between the two parts. If there is no blank, it indicates that white-space between the parts is not possible.

Table 2.1: Grammar syntax

#	Semantics	Syntax
(00)	Program	(<T> | <F>)*
(01)	T(ype)	<AT> | <CT>
(02)	A(bstract)T(ype)	‘abstract’ ‘type’ <I_t> <TO>* ‘;’
(03)	T(ype)O(rder)	<S> | <G>
(04)	S(pecialization)	‘specializes’ <I_t>
(05)	G(eneralization)	‘generalizes’ <I_t>
(06)	C(oncrete)T(ype)	‘type’ I_t<FPTE> <TOC>* ‘;’
(07)	F(lat)P(roduct)T(ype)E(xpression)	‘(’ (<I_p> ‘:’)? <I_t>$*^{(‘,’)}$ ‘)’
(08)	T(ype)O(rder)C(oercer)	<SC> | <GC>
(09)	S(pecialization)C(oercer)	<S> ‘::=’ <FE>
(10)	G(eneralization)C(oercer)	<G> ‘::=’ <FE>
(11)	F(unction)	<AF> | <CF>
(12)	C(oncrete)F(unction)	‘function’ <I_f> <FTE> ‘::=’ <FE> ‘;’
(13)	A(bstract) F(unction)	‘abstract function’ <I_f> <FTE> ‘;’
(14)	F(unction)T(ype)E	‘[’ <TE> ‘-->’ <CTE> ‘]’
(15)	T(ype)E(xpression)	<I^t> | <FTE> | <LTE>
(16)	L(ist)T(ype)E(xpression)	<PTE> | <STE>
(17)	P(roduct)T(ype)E	‘(’ (<I_p> ‘:’)? <TE>$*^{(‘,’)}$ ‘)’
(18)	S(um)T(ype)E	‘{’ <TE>$*^{(‘,’)}$ ‘}’
(19)	C(odomain)T(ype)E	<I^t> | <CLTE>
(20)	C(odomain)L(ist)T(ype)E	<CPTE> | <CSTE>
(21)	C(odomain)P(roduct)T(ype)E	‘(’ <CTE>$*^{(‘,’)}$ ‘)’
(22)	C(odomain)S(um)T(ype)E	‘{’ <CTE>$*^{(‘,’)}$ ‘}’

A simple example of a type definition can be found in figure 2.1. The same for a function definition is shown in figure 2.2. The figures refer to the code examples which are described in section 2.1.

Table 2.2: Grammar syntax (continued)

#	Semantics	Syntax
(23)	F(unction)E(xpression)	<CFE> | <PFE> | <SFE> | <SerFE> | <AFE> | <ANFE>
(24)	C(onstructor)F(unction)E	<SCFE> | <TCFE> | <L>
(25)	S(imple)C(onstructor)FE	'(' (<I_p> ':')? <FE>*(',') ')'
(26)	T(yped)C(onstructor)FE	Not syntactically available
(27)	P(roduct)F(unction)E	'<' <FE>*(',') '>'
(28)	S(um)F(unction)E	'{' <FE>*(',') '}'
(29)	Ser(ial)F(unction)E	<FE>*('␣')
(30)	A(tomic)F(unction)E	<I^f> | <I> | <TI> | <FM> | <TFM> | <Pro> | <TPro>
(31)	AN(onymous)F(unction)E(xpression)	'[' <TE> '::=' <FE> ']'
(32)	I(dentity)	'.'
(33)	T(yped)I(dentity)	<I> '-' <I^t>
(34)	F(inal)M(ap)	'!'
(35)	T(yped)F(inal)M(ap)	<FM> '-' <I^t>
(36)	Pro(jection)	'.' (<N> | <I_p>)
(37)	L(iteral)	<N> | <S>
(38)	N(umber)	Natural number $\geq$ zero
(39)	S(tring)	'"' character* '"'
(40)	T(yped)Pro(jection)	<Pro> '-' <I^t>
(41)	I_p	Projection name declaration
(42)	I_t	Type name declaration
(43)	I^t	<UT> | <BIT>
(44)	U(ser)T(ype)	User defined type name application
(45)	B(uilt)I(n)T(ype)	See Built-In type chapter for available types
(46)	I_f	Function name declaration
(47)	I^f	<UF> | <BIF>
(48)	U(ser)F(unction)	User defined function name application
(49)	B(uilt)I(n)F(unction)	See Built-In function chapter for available functions

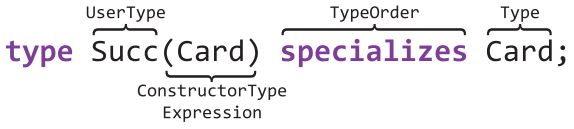

Figure 2.1: Relation between syntax and types (type definition)

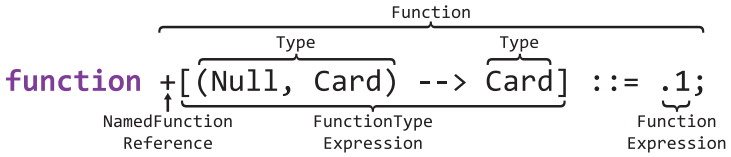

Figure 2.2: Relation between syntax and types (function definition)

2.1.2 Keywords

As mentioned before LOMF2 uses the following keywords which should not be used as any type- or function-identifier: `abstract`, `function`, `generalizes`, `specializes` and `type` as well as the symbols `!`, `.` and `-`.

2.2 Semantics

As LOMF2 is a categorial programming language, most features correspond to a categorial concept. Therefore a LOMF2 program can be thought of as a category. In the following subsections the already mentioned features will be semantically described.

2.2.1 Types

LOMF2 is a strongly typed programming language. Types represent the objects of the program's category. An empty program does only contain the `empty product` and a few built-in types as objects which will be introduced later on. Using these types, user defined named types can be constructed by declaring products and sums which correspond to the categorial concepts of products and sums. In this section product based types are introduced.

The (short) LOMF2-program: `type A();` defines a type `A` that can be constructed out of an empty product. This type can be compared to a class not owning any attributes in an object-oriented programming language (like C++ or Java).

One can declare n-ary products containing any named type contained in the program. For each dimension of a product, there exists a projection to extract the corresponding component out of the product. Instead of using names to identify projections, they are numbered starting with zero. For example `type P(A, B, C);` defines a product of the types `A`, `B` and `C` named `P`. Implicitly there are three projections `.0`,`.1` and `.2` defined such that `.0` applied on an element of `P` extracts an `A` element, `.1` an `B` element and so on.

The definition of a type implicitly creates a *constructor function* which maps a product to the defined type. Given the example from above there is a *constructor function* `P` that maps a product of an `A`, a `B` and a `C` element to a product element `P`. Again this is similar to a constructor of a class in an object-oriented language that has an argument for each attribute the class owns.

Named sum based types are used to model inheritance.

2.2.2 Type Hierarchy

The type hierarchy or inheritance as used in other object-oriented programming languages (like C++ or Java) finds its semantics in the categorial sum. As an example we define a type `Card` to represent the natural numbers including zero:

```
abstract type Card;
type Null() specializes Card;
type Succ(Card) specializes Card;
```

A named sum can be defined by a specialization or a generalization. The abstract type `Card` is a named sumtype. Each element of this type is either a `Null` or a `Succ` element. It's also possible to specify the named sum `Card` the other way round[6]:

```
abstract type Card generalizes Null generalizes Succ;
type Null();
type Succ(Card);
```

An abstract type can be compared to an *interface* in an object-oriented language. It is also possible to specify inheritance between concrete types. A concrete type `A` generalizing another concrete type `B` implies that there is a way to model each `B` element as an `A` element. To specify how this adaption (injection) works, *coercers* are used in LOMF2. As an example we define a type `Integer` which consists of two natural numbers modeling a positive and a negativ part. Each `Card` element can be converted into an `Integer` element by adding a zero as negative part.

```
type Integer(Card, Card) generalizes Card ::= (.,!Null)Integer;
```

[6]Note: Multiple inheritance is fully supported in LOMF2.

Each type can be thought of as a sum containing all of the type's concrete subtypes (including the type itself if it is concrete).

Besides the named types, the described category implicitly contains every finite product and sum of objects as an anonymous type[7]. This allows us to define functions having anonymous types as domain or codomain. Every *constructor function* is a function from an anonymous product to a named type.

2.2.3 Functions

As mentioned above *types* can be thought of as objects in a category. *Functions* bring morphisms into the categorial programming language LOMF2 as they are maps between types. A *concrete function* consists of a name, a signature and a function expression which defines its behaviour. An *abstract function* consists of a name and a signature with an abstract domain[8] only. At runtime it has to be implemented by at least one concrete function, whose function expression will be evaluated.

Signatures

The function's signature defines the domain and codomain of the morphism. The domain specifies the type of the argument on which the function can be applied. The codomain specifies the type of the result of an application of the function on an element typed in the domain.

It is possible to define several functions with the same name, if they differ in domains. The domains do not necessarily have to be in the same type hierachy. The following rule must hold for all functions f,f' with the same name: If the domain of f' is subtype of the domain of f, than the codomain of f' must be subtype of the codomain of f.

Function Expressions

The function expressions are responsible for the behavior of a function. Basically a function expression is the composition of functions (resp. morphism). The composition of two morphisms $g \circ f$ is denoted in LOMF2 syntax by: $f\ g$. Function expressions are strictly

[7] This implies that sums can be contained in a product and vice versa.
[8] The domain of a signature is abstract, if at least one type of the domain is abstract.

evaluated from left to right. The LOMF2 syntax does not allow manipulating the evaluation sequence through brackets, as they are only used to define products. Therefore the associative law for functional composition holds. A function can be applied on an element of its domain's type and returns an element of its codomain's type.

Since categories contain an identity morphism for each object, these are also available in LOMF2. The dot-operator (".") stands for the identity of the given argument. If an identity is applied on an element, the element itself will be the result of the application.

The exclamation mark ("!") denotes the final morphism to the final object[9], which is the empty product. The final morphism is used if a function without parameters should be called. We used the final map above in the coercer function from `Card` to `Integer` to call the constructor function `Null`.

2.2.4 Anonymous Types

As described above LOMF2 supports anonymous sums and products.

We already came across *anonymous products* while defining named types. As mentioned above every type definition `type A(B,C);` implies a constructor function from the anonymous product `(B,C)` to `A`.

The same notation can be used to create elements of the anonymous product's type. Assume that `f` and `g` are functions. Then the domain of `(f, g)` is the intersection of the codomains of `f` and `g`. The codomain of this "constructor" is the anonymous product of the codomains of `f` and `g`.

A special function expression is the product map, which maps a product to another product by calling a function on each projection of the product and implicitly constructing a new product out of the elements returned by both functions. The product maps use the "<", ">"-syntax. Let `m` be a function having the anonymous product `(A,B)` as codomain and `f` and `g` be two functions with domain `A` resp. `B`. Then the following expressions are equivalent:

```
(context) m <f,g>
(context) m (.0 f, .1 g)
```

[9]Note: In a LOMF2 program the initial object (empty sum) is not available.

So that `<f,g> = (.0 f, .1 g)`. Building products in both ways can be executed concurrently.

Anonymous sums can be used in signatures or in function expressions. An anonymous sum uses the "{", "}"-syntax. E.g. to declare that a function `f` either expects a parameter of type `A` or `B`, one can simply write: `function f[{A,B}-->C] ::= ...`

In function expression the anonymous sums can be used for case distinction by using typed identities or typed projections. As an example again we look at the natural numbers `Card` defined above. Assume that the context's codomain is `Card`.

```
function f[Card --> Card] ::= !Null;
function g[Card --> Card] ::= !(Null)Succ;


(context) {.-Succ g, .-Null f}
```

Then the expression will return one if the context is zero and return zero if not.

Giving following code the equations $\{A, B\} = C$ and $(A, B) \neq D$ hold.

```
type A ...
type B ...
abstract type C generalizes A generalizes B;
type D(A, B);
```

2.2.5 Built-in Types and Functions

LOMF2 contains three built-in types and a set of built-in functions operating on these.

Bool

LOMF2 contains a minimum implementation of a boolean type, because it is needed either as parameter or as result for built-in functions. The implementation is equivalent to the following LOMF2-code:

```
abstract type Bool;
type True() specializes Bool;
type False() specializes Bool;
```

Nat

The built-in type `Nat` is isomorphic to the type `Card` introduced above.

```
abstract type Nat;
type Zero() specializes Nat;
type NextNat(Nat) specializes Nat;
```

The main benefit in using the type `Nat` is, that instead of building terms for larger numbers, numeric literals can be used so that both of the following expressions are equal:

```
((Zero)NextNat)NextNat;
2;
```

The projection .0 is only applicable on natural numbers greater than zero and reduces them by one.

Multiplication and addition (called by `*` and `+`) of natural numbers use the binary product of natural numbers as domain and the natural numbers as co-domain.

```
function +[(Nat, Nat) --> Nat]
function *[(Nat, Nat) --> Nat]
```

The function less or equal (called by `leq`) compares two natural numbers (domain) and returns `True` (co-domain) iff the first argument is less or equal than the second argument.

```
function leq[(Nat, Nat) --> Bool]
```

String

The `String` type takes a special role in LOMF2. It is the only type that is neither abstract nor constructable. Elements of the type String can only be created using `String` literals enclosed in “ ” or built-in functions. Therefore no projections are applicable on `Strings`.

Concatenation of two strings is also denoted by `+` but differs to addition of natural numbers in the domain (product of two strings) and co-domain (string).

```
function +[(String, String) --> String]
```

Splitting of a string is implemented by `split`. The domain is represented by the binary product of string and natural number. The second argument marks the position where the first argument shall be split. The co-domain is the binary string product which contains both parts of the split string as result.

```
function split[(String, Nat) --> (String, String)]
```

The length of a string can be determined by function `size`.

```
function size[String --> Nat]
```

The function `contains` compares two strings (domain) and returns `True` (co-domain) iff the second argument is part of the first argument.

```
function contains[(String, String) --> Bool]
```

The function less or equal (called by `leq`) compares two strings (domain) and returns `True` (co-domain) iff the first argument is lexicographical less or equal than the second argument.

```
function leq[(String, String) --> Bool]
```

@pply Function

LOMF2 contains one special function for applying functions that are provided as parameter. This function is called *apply* and is represented by the `@` operator.

```
function @["<T> ([T -> X], T)" --> T]
```

The domain (`"<T> ([T -> X], T)"`) of this function is the sum of all products consisting of a function and the corresponding argument for it. The codomain of the function depends on the provided parameters i. e. on the codomain of the function that has to be provided as the first factor in the parameter. This behavior is also known from *generic functions* in the Java programming language.

3 Model

In this chapter we describe essential modules of the LOMF2 datamodel. Intention of this section is to give a basic overview of the model, such that subsequent working groups are able to extend the existing structure in a comfortable way.

3.1 Type

The LOMF2 type system differs between built-in and user types (see figure 3.1).

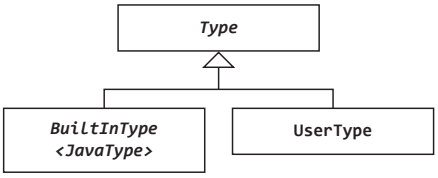

Figure 3.1: Type hierarchy

Built-in Types are types, which are created and provided at runtime by LOMF2 such as `BoolType` (≙ Boolean), `NatType` (≙ natural numbers with zero), `NonZeroNatType` (≙ natural numbers greater than zero), `ZeroNatType` (≙ zero) and `StringType` (≙ String). These names refer to class names. The relationship between class names and type names at runtime is depicted in table 3.1.

`BuiltInType`s may be constructable like `ZeroNatType` and `NonZeroNatType` or not constructable like `String`. Constructable types are concrete and posses a *ConstructorTypeExpression* and a set of *Projection*s. `String`s are not constructable and can only be created by using *literals* and *functions*. *Type*s which are not constructable do not provide *ConstructorTypeExpression* and *Projection*s are not applicable on them.

User Types are types which are defined by the user in LOMF2 syntax. A *UserType* being concrete implies that it is constructable.

Table 3.1: Constructable built-in types

Built-In type classname	Constructor
ZeroNatType	Zero
NonZeroNatType	NextNat
FalseType	False
TrueType	True

3.1.1 Type Expressions

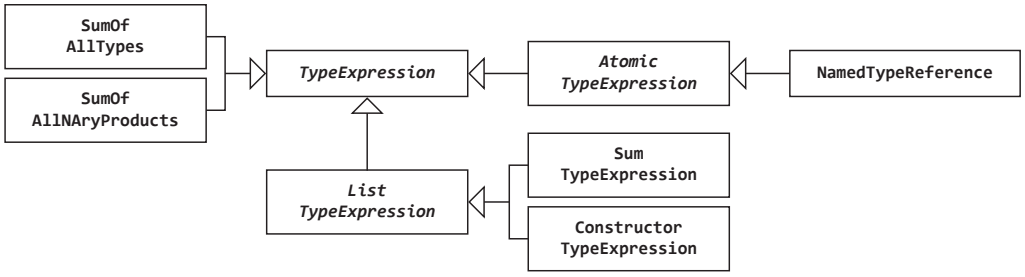

Figure 3.2: Type expression hierarchy

There are several expressions in order to declare the notion of a type: The *TypeExpression*. All classes, that express some kind of type inherit from this class. There are two key operations[10], that every *TypeExpression* has to implement.

- $isSuperTypeOf$: $TypeExpression \times TypeExpression \times AddToSumStrategy \rightarrow Boolean$. Invoking this operation on a *TypeExpression* given another *TypeExpression* and a *AddToSumStrategy* will return true, iff the object on which the operation is invoked on is a supertype of the given *TypeExpression*. The *AddToSumStrategy* specifies how abstract or non constructable types are handled and is explained in the following.
- $concretize : TypeExpression \times AddToSumStrategy \rightarrow TypeExpression$. This operation will return a normalized representation of the given *TypeExpression*. The *AddToSumStrategy* specifies which subtypes will be added to the resulting *SumTypeExpression*. Available strategies are:
  - *AddAllTypes* adds all *NamedTypeReference*s to the resulting sum.
  - *AddOnlyConcreteTypes* adds only *NamedTypeReference* referencing concrete types to the result.

[10] Note: There are several other necessary operations e. g. visitor implementations, which are not important in this context.

- *AddOnlyMostGeneralConcreteTypes* adds a *NamedTypeReference* to the result, if the referenced type is concrete and no concrete supertype is already contained. If a *NamedTypeReference* is added all already contained subtypes are removed from the result.
- *AddOnlyConstructableTypes* adds only *NamedTypeReference* referencing a type that posseses a constructor. Constructable types are a subset of all concrete types which contain all concrete *UserType*s and all *BuiltInType*s having a constructor.

Atomic Type Expressions

Atomic type expressions are described as expressions, that do not contain more fine-grained parts. Up to this point, there only is the *NamedTypeReference*, which represents a reference to an explicitly defined type in a model. This expression is used to declare domains of certain function expression, where the domain is explicitly defined (typed final map, typed identity, typed projection).

List Type Expressions

These type of expressions are made up of list-like structures, containing an ordered list of other *TypeExpressions*.

First, there is a *ConstructorTypeExpression*, which represents the type of a categorical product. E.g. this this expression is used while declaring types and defining function signatures.

Another list-like expression is the *SumTypeExpression*. It has the semantics of a categorical co-product. It is widely used when calculating the domain and co-domain of functions.

Artifical Type expressions

There are certain types that are introduced internally to the model. They cannot be used explicitly in a model.

The sum of all type expressions (*SumOfAllTypeExpressions*), represents the co-product of all types available in the model which is a super type of an arbitrary other type in the model.

There is the sum of all n-ary products (*SumOfAllNaryProducts*). This type is used to declare the domain of an untyped projection. Since there is no explicit type given, the domain has to be assumed as all products, which have at least as many projections as the untyped projection suggests - thus the sum of all n-ary products.

3.2 Functions

This chapter introduces the classes used to model *user functions* and *build-in functions* and explaines how functions are grouped to *operations*.

3.2.1 Function Expressions

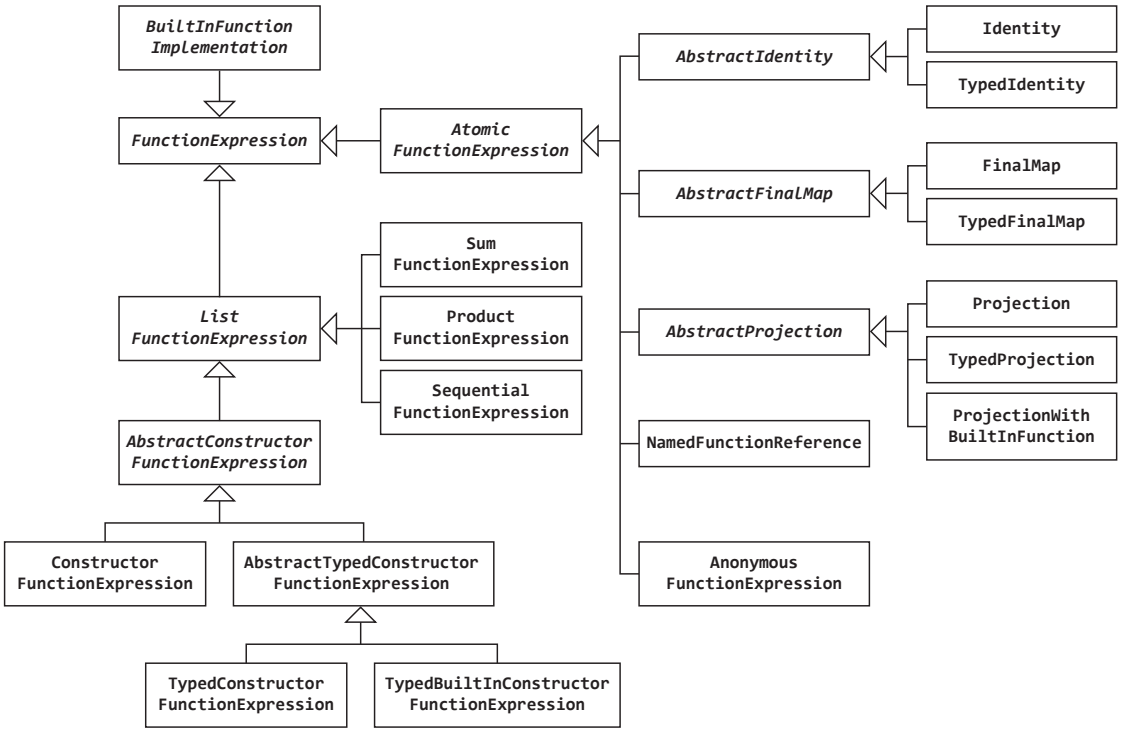

Figure 3.3: Function expressions hierarchy

The operational aspect in a model is described by *function expressions*. We find function expressions as the implementations of operations and as coercers between types in a hierarchy.

We differentiate between “atomic” and “list-like” function expressions, where the list-like expressions consist of other function expressions (see figure 3.3).

Atomic Function Expressions

There are different types of atomic function expressions.

A *final mapping* is a function, which maps a type to the empty product. With this mapping, we can 'forget' a type and apply constructors, that take the empty product as domain after the final map. There is a typed and an untyped version of the final map, where the typed final map explicitly specifies the domain of the final map.

To bring the categorical concept of *identities* into the model, there are two identity expressions (typed and untyped). The typed version takes as domain the specified type and has as co-domain exactly this type. The untyped version is polymorphic so that the co-domain is determined by the context the untyped identity is apllied to.

To access elements in a product type, there are *projection* expressions. Again, there are typed and untyped versions, where the typed version specifies the domain of the projection. A projection specifies the element which is accessed by a numerical index. The domain of a projection is either the specified type or – in case of an untyped projection – the sum of all products which have at least as many parts as specified in the projection plus one (since we start counting at zero). The co-domain is the type of the accessed element. If projections are applicable on a *BuiltInType*, it is equipt with a *BuiltInFunction* that is used in the evaluation to determine the result of the projection on an element typed in this *BuiltInType*.

To invoke a function, there are *named function references*. These kind of expressions denote the invocation of an operation which is specified by the model. The domain as well as the co-domain of these references are the sum of all domains or co-domains from all implementations of these operations.

AnonymousFunctionExpressions are functions which can be declared anonymously within a *FunctionExpression*. These anonymous function expressions consist of a function type expression and a function expression. The semantic is similar to the command pattern in object-oriented languages. The anonymous function expression can be passed as a parameter to another function with applicable domain and executed via the built-in apply function (@). Anonymous function expressions can be nested arbitrarily (but it is recommended not to nest anonymous function expressions, since these constructs downgrade the readibility of your LOMF2 code.

List-like Function Expressions

In order to be able to construct an element of a type, there is a *constructor function expression*. There is an untyped and two typed versions. If a *BuiltInType* is constructable it is equipt with an *BuiltInConstructorFunctionExpression*. Analogous to projections on elements of `BuiltInTypes` a *BuiltInFunction* is used to evaluate the constructor-application.

Since the application of an *constructor function expression* is semantically the application of all expression parts to the domain, the domain is the intersection of all domains of the expression parts. The co-domain is either the specified type in case of the typed version or a constructor type expression made up from all co-domains of the expression parts.

To apply different functions on parts of a product simultaniously, there is the *product function expression*. Its domain is a constructor type expression containing the domain of each part of the expression, whereas the co-domain is a constructor type expression containing all co-domains of the parts.

To execute functions sequentially, there is the *sequential function expression*. The domain of this expression is the domain of the first expression part and the co-domain is the co-domain of the last part.

With the *sum function expression* a conditionally execution of functions is possible. The domain of such an expression is a concretized sum type expression containing all domains of the expression parts. The co-domain is a sum type expression from all co-domains of the expression parts.

Built-in Functions

To improve reusability in LOMF2 models we use *built-in functions* for recurrent functions which use *built-in types* in their implementations. Instead of writing equivalent functions in different models we can reuse those built-in functions, which are published to the model by the *BuiltInTypeAndFunctionGeneratorTask* (section 5.2).

This version[11] of LOMF2 contains eight different *built-in functions* that have been introduced in Section 2.2.5. Section 6.2 describes how to implement additional *built-in functions*.

[11] *October 10, 2015*

3.2.2 Operations

Functions having the same name are grouped to operations in order to objectify and therefore describe the resulting set of functions. An operation consequently functions as an entry point to all equally named functions. This allows for a rather simple "jump-table" approach. Each function will be registered with its respective operation and will stay in or change to "uncached" state while the operation set of internal functions is being modified.

If a function call requires an updated version of the "jump-table" it uses the existing table (operation is in cached state) or triggers a recalculation of a new table by using the algorithm below (operation is in uncached state).

3.3 Error Model

The LOMF2 error model consists of two parts. The first part is the ModelError type hierarchy. All subtypes in this hierarchy describe a semantic error within the model (e.g. a missing type of a symmetric hierarchy is represented by its own subtype). Each type defines an error message that will be displayed in the GUI, a CodeContainable that consists of a subset of highlightable tokens in the model and a ModelErrorType. The ModelErrorType can be one of Error, Warning, Info or Style. The types are used to allow more precise flow-control within the checker architecture (see chapter 5.2). Also they are applied for transportation of information in situations where the occurrence of an error does not affect further checks.

The second part of the LOMF2 Error Model is the ModelException hierarchy. These types are, as the name suggests, subtypes of the Java Exception and are used for technical errors that occur during check phase. For example the getCodomain method of a SequentialFunctionExpression can throw a SequenceDomainCodomainMissmatchException if the types of domain and co-domain do not match. These exceptions will be wrapped by the checker in a corresponding ModelError for further handling. Like the ModelError, the ModelException includes a CodeContainable for its position description in the source code to highlight the position of the occurrence.

4 Evaluator

This section describes the fundamental evaluation-mechanism for LOMF2 functions. The following figure 4.1 shows the customized model-structure. To keep the additional components seperate from the original `Model`, an abstract class `AbstractModel` has been extracted. A model that is defined in the model tab of the application is parsed to a `Model` object and called *typeModel*. A model defined in the test tab of the application is parsed to a `EvaluableModel` object and uses the typeModel to evaluate its expressions.

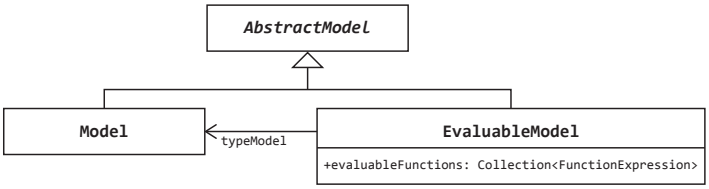

Figure 4.1: Evaluable Modell

All abstract operations in `EvaluableModel` will be implemented by delegation-pattern using the concrete model (association `typeModel` see figure 4.1).

The `EvaluableModel` was extended by the following list of evaluable functions:

```
+evaluableFunctions:  Collection<FunctionExpression>
```

Additionally a model is required that contains all evaluated values. The following figure (4.2) shows the defined model.

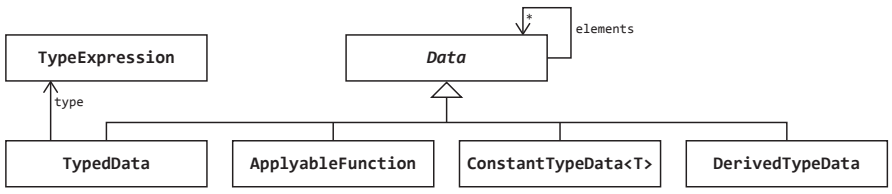

Figure 4.2: Evaluation Datamodel

The association `Elements` represents the possible projections of an evaluated expression.[12]

The class `Data` provides an abstraction for the following, four concrete 'value-types':

TypeData Provides a typed and evaluated value (e. g. `Null`).[13]

ApplyableFunction Represents a not yet executed function, which is the result of an `AnonymousFunctionExpression` in the evaluation.

ConstantTypeData This type represents an evaluated constant value, which is typed in a `BuiltInType`.

DerivedTypeData Provides an evaluated sum or product, in which the concrete type is not determined explicitly[14].

4.1 Evaluation Algorithms

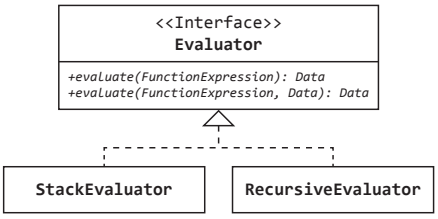

Figure 4.3: Evaluation-Strategies

The interface `Evaluator` was created for the evaluation and provides the necessary operations. Entrypoint for evaluation is the following operation:

```
+evaluate(FunctionExpression):  Data
```

There are two concrete evaluators: `StackEvaluator`[15] and `RecursiveEvaluator`. LOMF2 uses the `RecursiveEvaluator` which evaluates a `FunctionExpression` recursively, while the `StackEvaluator` does this by using a stack.

The basic evaluation of expressions will be described in the following overview (see table 4.1 below). The general evaluation works as follows:

[12] E. g. the type `Succ(Null)` contains one element of type `TypedData` (Null). A sum contains an element for each summand.
[13] Note: At least one concrete type has to be assigned.
[14] Like the name `DerivedTypeData` suggest.
[15] This evaluator was created to gain the ability of developing an LOMF2-internal debugger.

$$\texttt{V A} \rightsquigarrow \texttt{A(V)}$$

The $\rightsquigarrow$ suggest a replacement process. The expression on the left side of an arrow, will replaced by the right side. For example in case of a product, the description in table 4.1 means, that the expression V will applied on every element of the product.

Most of the following cases require, that the evaluation remember at least parts of the V to use them for further evaluations. To accomplish that requirement, the evaluation works as a stack (or, as described before, recursively).

Table 4.1: Evaluation

Semantics	**Syntax**
Projection	V.i $\rightsquigarrow$ V(i)
Operation	V op $\rightsquigarrow$ V $method^{Op}(v)$
Identity	V. $\rightsquigarrow$ V
Terminal	V! $\rightsquigarrow$ ()
Sum	V $\{s_1, \ldots, s_n\}$ $\rightsquigarrow$ V $\{s_1, \ldots, s_n\}$(v)
Product	V $(f1, \ldots, f_n)$ $\rightsquigarrow$ (V $f1, \ldots, V f_n$)
Constant	V constant $\rightsquigarrow$ $V_{constant}$

5 Infrastructure (IDE)

5.1 Scanner, Parser & Checker

The IDE infrastructure of LOMF2 consists of tree main parts. The first two parts is an simple Scanner-Parser construct. The third part is the checker infrastructure. Here all checks that are performed on the source code are modeled as checks that are executed in an predefined order. This is descriped in detail in the following chapters.

5.2 Checker Infrastructure

5.2.1 Service

The model checker service provides a facility for parallel execution of model checker tasks. Tasks can statically be registered with the service as each tasks will get called only once sometime during the execution. In order to prepare the registered tasks order, the service requires the tasks to provide there direct predecessors. Each task will therefore be executed as early as possible and as late as necessary. On execution, the service executes each set of parallel independent tasks asynchronously at waits for all tasks to call back on their completion. Therefore the service also acts as a point of synchronisation between each phase of parallel tasks execution and can be used to introduce changes to the model itself.

5.2.2 Tasks

There are different tasks to perform the checking on the model, generate elements and set references. The following figure 5.1 shows the complete sequence in which they are executed. In this section a selection of tasks is described in detail.

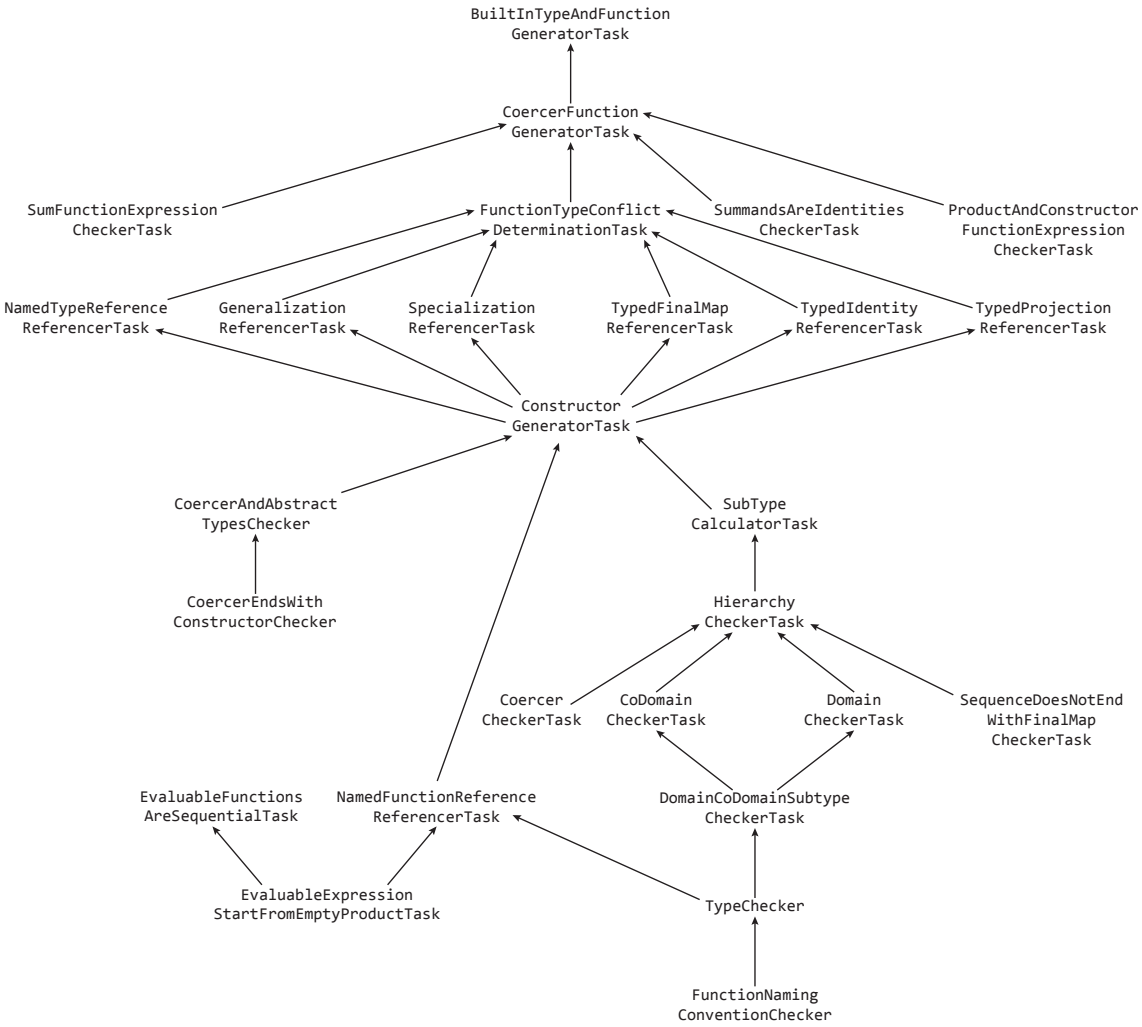

Figure 5.1: Sequence of predecessors

BuiltInTypeAndFunctionGeneratorTask This checker adds the built-in types like String, Bool, Nat, ... and built-in functions like apply (@), addition on type Nat, ... to the model.

CoercerFunctionGeneratorTask This task adds a coercer as a function to the model, such that, it can be used while evaluation and checking.

The predecessor of this task is the *BaseTypeGeneratorTask*.

FunctionTypeConflictDeterminationTask The *FunctionTypeConflictDeterminationTask* determines all *Functions* and *Types* which are in conflict with each other. For example:

`type A();` and `function A[...] ::= ...` are in conflict, because they have the same name (i. e. *A*).

ProductAndConstructorFunctionExpressionCheckerTask The *ProductAndConstructorFunctionExpressionCheckerTask* checks a *ProductFunctionExpression* and *ConstructorFunctionExpression* contains at least one part: `(.0)`, `(.0, .1)`, `<.0>`, `<.0, .1>` etc. An empty *ProductFunctionExpression* or *ConstructorFunctionExpression* is not allowed.

SumFunctionExpressionCheckerTask The *SumFunctionExpressionCheckerTask* checks, that a *SumFunctionExpression* contains at least one part. If the *SumFunctionExpression* does not contain any part, an Error (i. e. EmptySumError) will be created. In the case that the *SumFunctionExpression* contains only one part, an style-error (i. e. SumContainsOnePartError) will be created.

SummandsAreIdentitiesCheckerTask The *SummandsAreIdentitiesCheckerTask* checks, that all summands in a implicit sum (i. e. *SumFunctionExpression*) start with an identity expression. Following expressions are allowed for example:

- `{.-A, .-B}`
- `{<.-A,.-C>,.-B}`
- `{.-A, .-B, .}`
- etc.

Not more than one summand in an implicit sum can start with an untyped identity.

SpecializationReferencerTask The *SpecializationReferencerTask* sets a link (for the association *refersTo*) to the specialized type in *Specialization*. If the model does not contain the type, which have to be referenced, an error (i. e. MissingTypeError) will be created.

The predecessor of this task is the *FunctionTypeConflictDeterminationTask*.

GeneralizationReferencerTask The *GeneralizationReferencerTask* sets a link (for the association *refersTo*) to the generalized type in *Generalization*. If the model does not contain the type, which have to be referenced, an error (i. e. MissingTypeError) will be created.

The predecessor of this task is the *FunctionTypeConflictDeterminationTask*.

TypedProjectionReferencerTask The *TypedProjectionReferencerTask* sets the referenced type for a *TypedProjection*. If the model does not contain the type, which have to be referenced, an error (i. e. MissingTypeError) will be created.

The predecessor of this task is the *FunctionTypeConflictDeterminationTask*.

TypedFinalMapReferencerTask The *TypedFinalMapReferencerTask* sets the referenced type for a *TypedFinalMap*. If the model does not contain the type, which have to be referenced, an error (i. e. MissingTypeError) will be created.

The predecessor of this task is the *FunctionTypeConflictDeterminationTask*.

NamedTypeReferenceReferencerTask The *NamedTypeReferenceReferencerTask* sets the referenced type for a *NamedTypeReference*. If the model does not contain the type, which have to be referenced, an error (i. e. MissingTypeError) will be created.

The predecessor of this task is the *FunctionTypeConflictDeterminationTask*.

TypedIdentityReferencerTask The *TypedIdentityReferencerTask* sets the referenced type for a *TypedIdentity*. If the model does not contain the type, which have to be referenced, an error (i. e. MissingTypeError) will be created.

The predecessor of this task is the *FunctionTypeConflictDeterminationTask*.

ConstructorGeneratorTask This *ConstructorGeneratorTask* adds a typed constructor (i. e. Function) for each concrete *UserType* and for each constructable *BuiltInType* within the model.

The predecessor of this task are: *SpecializationReferencerTask*, *GeneralizationReferencerTask*, *TypedProjectionReferencerTask*, *TypedFinalMapReferencerTask*, *NamedTypeReferenceReferencerTask* and *TypedIdentityReferencerTask*.

CoercerAndAbstractTypesChecker This task containts two main-goals. In general coercers from and to abstract types are not allowed. This means in detail, that an abstract generalization cannot have a coercer and additionally a specialization to an abstract class cannot have a coercer, too.

SubTypeCalculatorTask The *SubTypeCalculatorTask* calculates the subtypes for every *Type* within the model.

The predecessor of this task is the *ConstructorGeneratorTask*.

NamedFunctionReferenceReferencerTask The *NamedFunctionReferenceReferencerTask* sets all *Function*-references within the given model. If *Function*-references refer to a missing *Function* a *MissingFunctionError* is added to the model.

The predecessor of this task is the *ConstructorGeneratorTask*.

EvaluableExpressionStartFromEmptyProductTask Each evaluable function has to start with `!`, so that the evaluation can start with the empty product.

CoercerEndsWithConstructorChecker At the moment, it is not possible to define a arbitrary `FunctionExpression` as coercer, only `ConstructorExpressions` are allowed.

HierarchyCheckerTask The *HierarchyCheckerTask* checks that the subtype relation is a hierarchy (antisymmetric).

The predecessor of this task is the *SubTypeCalculatorTask*.

DomainCheckerTask The*DomainCheckerTask* checks the domain of the assigned *FunctionTypeExpression*. The following combinations are not allowed in a domain of a function:

- {}
- `(A, ())`
- {`A, ()`}
- etc.

A sum with only one summand and a product with only one part leads to a style-error in the model. It is possible that the domain is an empty product (e. g. `f[() --> ...]`).

The predecessor of this task is the *HierarchyCheckerTask*.

CoDomainCheckerTask The *CoDomainCheckerTask* checks the codomain of the assigned *FunctionTypeExpression*, not allowed are:

- {}
- `()`

The predecessor of this task is the *HierarchyCheckerTask*.

SequenceDoesNotEndWithFinalMapCheckerTask determines that every *SequentialFunctionExpression* does not end with a *FinalMap* oder *TypedFinalMap*.

The predecessor of this task is the *HierarchyCheckerTask*.

CoercerCheckerTask The *CoercerCheckerTask* checks the coercer definitions such that:

If t' is abstract, the (specialization representing) monomorphism is always the set inclusion. If t' is concrete, each concrete type $t \leq t'$ needs an explicit coercer specification into t' (completeness of concrete coercers):

If a concrete type specialises another concrete type directly, the coercer must be provided in the appropriate syntactical clause. Transitivity is given by composition of coercers. (There might be serveral paths. It is not checked if these paths represent the same morphism.)

If an abstract type t specialises a concrete type t', a direct coercer is not required and syntactically impossible. Nevertheless, all concrete (direct or indirect) subtypes t'' of t need a coercer into t'. These coercers can be specified in the definition of t' (if the subtype exists prior to the definition of t') or in the definition of t'' (if t'' becomes a subtype of t after the definition of t').

This checks requieres that each *TypeOrder* between two concrete types has a coercer.

The predecessor of this task are: *HierarchyCheckerTask* and *SubTypeCalculatorTask*.

DomainCoDomainSubtypeCheckerTask The *DomainCoDomainSubtypeCheckerTask* determines the following property holds for each pair of methods f1 and f2 for the same function f: `domain(f1) <=(abstract) domain(f2) implies codomain(f1) <=(abstract) codomain(f2)`.

The predecessor of this task are: *DomainCheckerTask* and *CoDomainCheckerTask*.

TypeChecker The *TypeChecker* determines whether the domain of the method is greater or equal than the domain of the function (the method belongs to) and whether the codomain of the function is greater or equal than the codomain of the method.

The predecessors of this task are *DomainCoDomainSubtypeCheckerTask* and *NamedFunctionReferenceReferencerTask*

5.3 GUI

The LOMF2 GUI (see figure 5.2) is an editor for LOMF2 source code. It is devided into two parts (tabs).

The "Model" tab is for typing source code and checking its correct syntax. This can be done by clicking the "Check" button at the bottom of the editor. If errors occur during the check, they will be displayed in the error table below the source code and underlined in the source code. By clicking an item of the table the view jumps to the referenced position.

The "Testing" tab is for executing the typed source code of the model tab. Built-in types and functions allow out of the box operations like calculating with natural numbers (see section 3.1). This can be done by clicking the "Evaluate" button at the bottom of the editor.

The editor also allows basic file operations like load and store. The model and testing tab will be saved in two files (`filename.model` and `filename.test`).

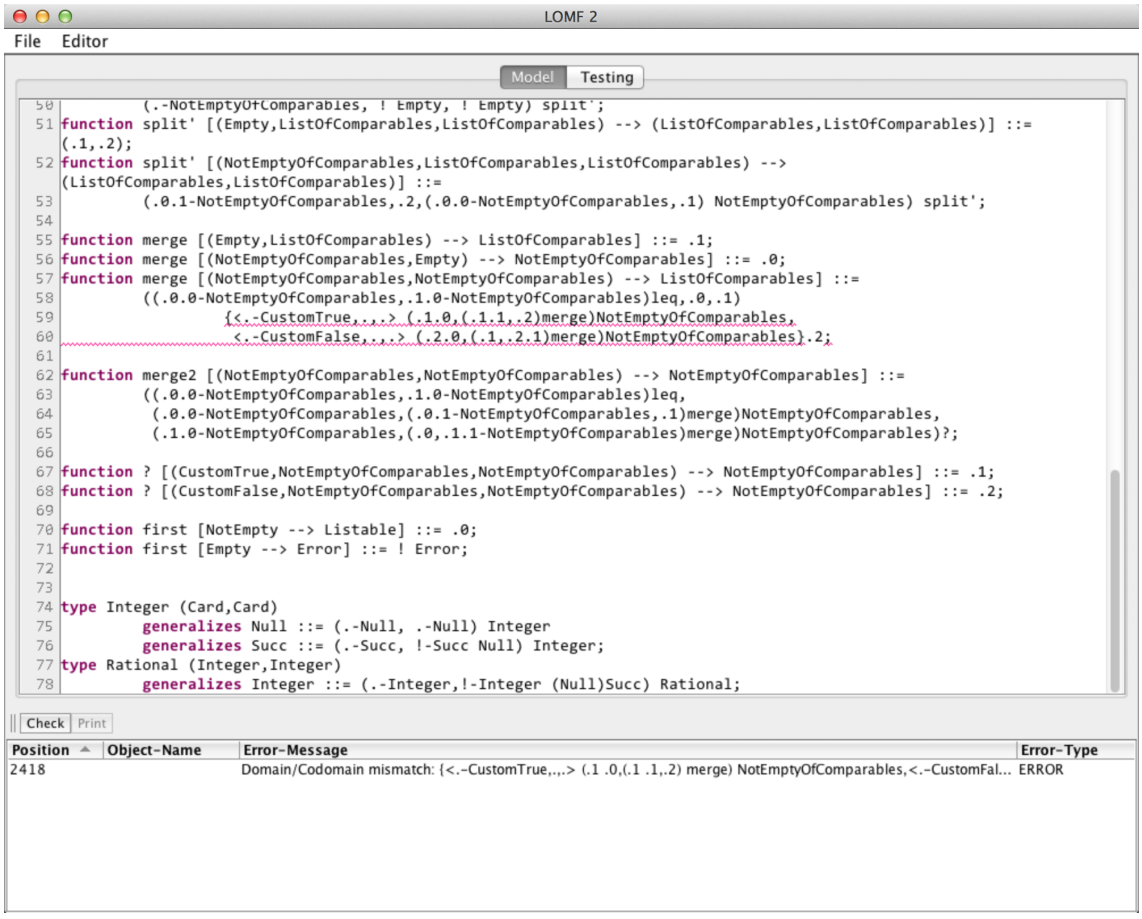

Figure 5.2: The LOMF2 GUI

6 How-to

The following sections contains step-by-step instructions to extend the LOMF2 system with additional functionality (for instance built-in types and functions).

6.1 Add New Built-in Type

To add a new *BuiltInType* **T** to the system, follow these steps:

6.1.1 Initial Steps

- Add a string-constant `TYPENAME_<T>` in the class `BuiltInType<JavaType>`.
- Create a new subclass in `de.fhdw.lomf.model.type` which extends `BuiltInType<JavaType>` and is *Singleton*.
- Modify the `constructor` of the class to add all specialization-relationships to the `this.hierarchies`-set.
- Specify if the new type is abstract or concrete in the method of the operation `isAbstract`.

6.1.2 Further Steps for Abstract Types

If **T** is abstract:

- Implement `hasConstructor` to return `false`.
- Implement `fetchConstructorDomain` to return `null`.
- Implement `getConstructorMethod` to return `null`.

Example: `de.fhdw.lomf.model.type.NatType`

6.1.3 Further Steps for Constructable Concrete Types

If **T** is concrete and constructable:

- Implement `hasConstructor` to return `true`.
- Implement `fetchConstructorDomain` to return the domain of the constructor.
- Implement `getConstructorMethod` to return a `TypedBuiltInConstructorFunctionExpression` that contains `ProjectionWithBuiltInFunction`s as parts and a `BuiltInFunctionImplementation` as evaluation-method.

Example: `de.fhdw.lomf.model.type.NonZeroNatType`

The `ProjectionWithBuiltInFunction`s will be used in the evaluation to determine the algorithm to project on the type. The `BuiltInFunctionImplementation` will be used in the evaluation to determine the result of a constructor-application on the constructor's domain.

6.1.4 Further Steps for Non-Constructable Concrete Types

If **T** is concrete and not constructable:

- Implement `hasConstructor` to return `false`.
- Implement `fetchConstructorDomain` to return `null`.
- Implement `getConstructorMethod` to return `null`.

Example: `de.fhdw.lomf.model.type.String`

6.1.5 Add `BuiltInType` to Model

To finally add a *BuiltInType* to the model, it has to be added to the static set of *BuiltInTypes* `BUILT_IN_TYPES` in the `BuiltInTypeAndFunctionGeneratorTask` in package `de.fhdw.lomf.text.checker`.

6.2 Add New Built-in Function

- Create a new subclass in `de.fhdw.lomf.model.type.functions` which extends `BuiltInFunctionImplementation`.
- Implementation of relevant methods. The `execute` method of the new subclass contains the logic of the new built-in function. The class `NatPlus` can exemplify this by the addition of natural numbers.
- The new subclass has to be registered in LOMF2 by adding it with `addFunction` to the `performOperation` method in `BuiltInTypeAndFunctionGeneratorTask`.

7 Outlook

The following topics could be part of future development by master courses:

- Naming concept.
- Generics.
- Parallelized evaluation.