this version: May 13, 2011 20:32

The FMathL type system

Peter Schodl
Arnold Neumaier

Fakultät für Mathematik, Universität Wien
Nordbergstr. 15, A-1090 Wien, Austria
WWW: http://www.mat.univie.ac.at/~neum/FMathL.html

Abstract

The FMathL type system is a common generalization of context-free grammars and algebraic data types in programming languages, and consists of a system of declared categories and types. The type declarations themselves are represented via typed objects, making the whole typing self-consistent. Correctness of types can be checked in linear time.
The type system is defined within a framework for representing semantic content, specially designed to naturally represent arbitrary mathematics. Information is represented via semantic units (sems) relating objects, an undefined notion that may be interpreted as elements of the domain of a semantic mapping. Meaning is conveyed by means of type declarations that specify how objects may relate with each other.

Contents

1 Introduction **1**

2 Definition of the framework **2**

3 The type structure **4**

4 Type sheets **5**

5 Description and representation of type declarations in the SM **8**

6 Well-typed records **26**

7 Type declarations and unions as types **27**

1 Introduction

In order to provide a good basis for the semantical analysis of mathematical language expressing arbitrary mathematical content, a system needs to represent mathematics specified in a (perhaps controlled) natural language. Thus a concept of typing is needed that covers

(i) syntactically correct mathematical formulas,
(ii) well-formed sentences built according to a linguistic grammar, and
(iii) structured records in the programming sense.

The typing must be such that, using an appropriate type system, one can define and easily check their well-formedness. In particular, grammatical categories must be representable in the type system. To serve as a foundation in the sense of the FMathL framework [4],

everything is set up in a way that it can reflect itself without the need to augment the basic structure.

The essential formal structure achieving this is the introduction of a **semantic memory** as the abstract representation medium, **categories** as the fundamental structuring concept, and of **types** as a particular form of categories.

Comparison with the type system of XML. Typing in FMathL and typing in XML bear significant similarities, most notably with DTD and Relax NG. (For a description of DTD, Relax NG and other XML schemas, see Lee & Chu [2].) Some of the operators in type declarations have a direct correspondence in the language of DTD and the RelaxNG compact syntax. E.g., `?` in DTD corresponds to `optional`, the pipe `|` corresponds to `oneOf` and parentheses `( )` correspond to `allOf`. A valid XML document corresponds to a well-typed record. However, there are also important differences, since cycles are an important feature of our framework that enables an efficient representation of concrete and abstract grammars, while XML documents are always organized in trees.

Overview. In section 2 we recall the definition of the framework called the **semantic memory**, as defined in [?]. Section 3 defines the **type system** in an abstract way and the type of an object. Section 4 gives the grammar of text documents containing a set of type declarations. Section 5 introduces the requirements we can pose on a record, how to write them down in a text document, and how they are represented in the framework. In Section 6 we define when a record is **well-typed**. In Section 7 we discuss how types (as represented in the semantic memory) can themselves be typed. This closes the reflection cycle and makes the whole typing concept self-consistent.

Acknowledgements. Thanks to Hermann Schichl, Ferenc Domes, Kevin Kofler, and Mike Mowbray for their comments in various discussions of preliminary versions. Support by the Austrian Science Foundation (FWF) under contract number P20631 is gratefully acknowledged.

2 Definition of the framework

We define the abstract data structure we use to represent mathematics.

It can be regarded as a special case of a **semantic network**, introduced by Richens [?] in 1956. This and akin concepts are discussed in detail by Sowa [?]. Also, it is inspired by, and representable in, the semantic web [3]. A standardized and widely used example of a semantic net with the aim to be used in the World Wide Web is the Resource Description Framework (RDF), described by Manola et al. [?] and specified by Lassila et al. [?].

2.1 The semantic memory

There is an unlimited number of **objects**, but only finitely many of them are represented explicitly in stored memory. Objects can be compared for equality, which is an equivalence relation. On the metalevel, we refer to objects by strings not beginning with a hash (#); different objects are referred to by different strings. `Empty` is an object. **Object variables** are variables in the usual sense, ranging over the set of objects. We refer to object variables via a string beginning with a hash (#) followed by some alphanumeric string. For example, in the statement

#`name.type` = `String` for every object #`name` representing a string,

`type` and `Name` are specific objects, and #`name` is a variable in the same sense as x is a variable in

$$x^2 \text{ is even for every even integer } x.$$

Usually, we will use suggestive strings for variables, e.g., we use #`handle` or #`h` for an object that is intended to be a handle.

A **semantic mapping** (abbreviated SM) assigns to any two objects #`h` and #`f` a unique object #`h`.#`f` such that

if #`f` = `Empty` or #`h` = `Empty` then #`f`.#`h` = `Empty`.

A **semantic unit** (short **sem**) is an equation of the form #`h`.#`f` = #`e` with nonempty #`h`, #`f`, and #`e`; we call #`h` the **handle**, #`f` the **field**, and #`e` the **entry** of the sem. The **constituents** of an object #`a` are the sems in which #`a` is the handle.

Semantic mappings are used to store mathematics, but to be able to alter the data we need a dynamical framework. The semantic mapping that changes over time (formally, a semantic mapping valued function of time) is called the **semantic memory**.

A **position** is a pair (#`h`/#`f`) consisting of two objects #`h` and #`f`. We call #`h` the **handle**, #`f` the **field** and #`h`.#`f` the **entry** of (#`h`/#`f`). This position is called **occupied** if #`h`.#`f` is not `Empty`.

We say that the sem #`d`.#`e`=#`f` **follows** the sem #`a`.#`b` = #`c` if #`d` = #`c`. Using a left-associative notation, we then write #`a`.#`b`.#`e` = #`f`; thus #`a`.#`b`.#`e` stands for (#`a`.#`b`).#`e`. This notation naturally extends to more dots.

A short-hand notation for k repetitions ($k = 0, 1, 2, \ldots$) of a field: $\#\texttt{a}.\underbrace{\#\texttt{b}.\cdots.\#\texttt{b}}_{k \text{ times}}.\#\texttt{e}$ is written as $\#\texttt{a}.\#\texttt{b}^k.\#\texttt{e}$.

A **path of sems** starting at #`h` and ending at #`e` is a sequence of sems such that the first sem has the handle #`h`, each later sem follows the previous one, and the last sem has entry #`e`, and no sem has the field `type`. An object #`e` is **reachable** from a handle #`h` if there is some path of sems starting at #`h` and ending in #`e`. A sem is **reachable** from a handle #`h` if there is some path of sems starting at #`h` that contains that sem. A position is **reachable** from a handle #`h` if the handle of that position is an object reachable from #`h`.

If the set of sems reachable from an object #`h` is finite, then the set of sems reachable from #`h` defines the **record** with handle #`h`.

Clearly, a SM allows one to construct arbitrarily complex records. In contrast to records in programming languages such as Pascal, records in a SM may contain cycles. Indeed, backreferences are an important part of the design of the type system; for example, they allow labelled context-free grammars to be defined as type systems.

2.2 Illustration by semantic graphs

For graphical illustration of a semantic mapping, we will interpret a sem #`a`.#`b`=#`c` as an edge with label #`b` from node #`a` to node #`c` of a directed labeled graph, called a **semantic graph**. Objects may, but need not have **external values**, i.e., data of arbitrary form, associated with the object, but stored outside the semantic memory. We refer to the value of an object #`obj` by `VALUE`(#`obj`). In a semantic graph, objects that have an external value are printed as a box containing that value. For better readability we use dashed edges for edges labeled with `type`, since these constituents have importance for the typing, and bold edges for edges labeled with `next`, since this makes linked lists more readable. Different nodes of the semantic graph may represent the same object. For example, the information $\frac{12}{4} = 3$ may be represented as a list of sems as fiven in Figure 1, or equivalently as the semantic graph in Figure 2.

```
$380.type=Binary          $370.type=Fraction       VALUE($244) = 12
$380.lhs=$370             $370.num=$244            VALUE($246) = 3
$380.rhs=$246             $370.denom=$248          VALUE($248) = 4
$380.relation=Equal       $244.type=Integer
$246.type=Integer         $248.type=Integer
```

Figure 1: A list of sems and values

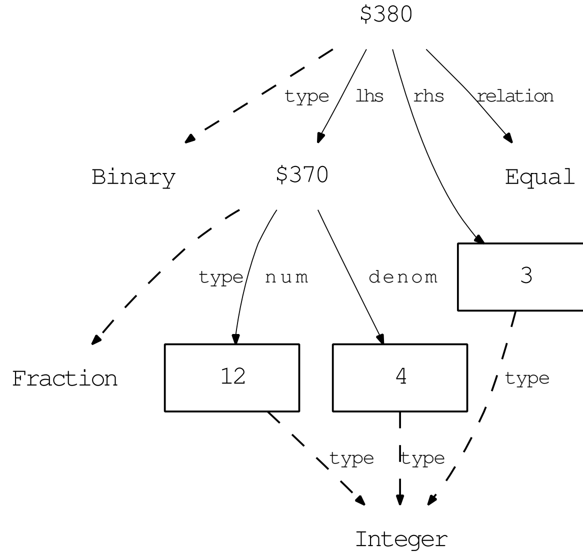


Figure 2: A semantic graph

3 The type structure

Information in the SM is organized in records. When using a record, or passing it to some algorithm, we need information about the structure of this record, as we do not want to examine the whole graph every time a record is used. For this reason we define a procedure to determine that a given record is **well-typed** of a certain type, or **ill-typed**. These assignments are always made with respect to a particular type system.

A **type system** is a set of objects which are called the **categories** of that type system. The object `Empty` is never a category.

In the following, until mentioned otherwise, we always consider an arbitrary but fixed type system and its associated order relations. The set of categories in a type system is ordered by an irreflexive partial order relation $<$. If for the categories #`C1` and #`C2` the relation #`C1` $<$ #`C2` holds, we say that #`C1` is a **subtype** of #`C2`.

We define the relation $<<$ to be the reflexive and transitive closure of the relation $<$, i.e., #`C1` $<<$ #`C2` if either #`C1` $=$ #`C2` or there exist categories $\#\mathtt{c}_1, \ldots, \#\mathtt{c}_n$ such that #`C1` $<$ $\#\mathtt{c}_1 < \ldots < \#\mathtt{c}_n <$ #`C2`. If #`C1` $<<$ #`C2` we say that #`C2` **contains** #`C1`.

A category is called a **type** if it is minimal in the ordering $<<$, and a **union** otherwise. Each type is either the **default type** `Object`, an **atomic type**, or a **proper type**.

Objects of an atomic type have no constituents; they are used as objects with a fixed semantic meaning. Objects of a proper type always have a field `type` whose entry is this type. Proper types are used to pose requirements on the other constituents of the objects of this type.

3.1 The type of an object and matching

Every object #`obj` has a **type**, defined by the following rules:

(i) If #`obj.type` is a proper type, then the type of #`obj` is #`obj.type`.
(ii) If #`obj.type` = `Empty`, and #`obj` is an atomic type, then the type of #`obj` is #`obj`.
(iii) Otherwise, the type of #`obj` is `Object`.

The fact that the type of an atomic type is object itself is the reason why we use the word "atomic type" and "atomic object" (or just "atomic") synonymously.

We say that an object #`obj` **matches** a category #`C`, in symbols: **m**(#`obj`/#`C`) if either #`C` = `Object`, or #`T` << #`C` for #`T` the type of #`obj`. Note that since `Empty` is not a category, no object matches `Empty`. Note also that which type matches which category depends on the type system used. Thus in an implementation, the type system appears as an extra argument.

For the basic type structure as presented here, the naming convention is to use names with an upper case initial for categories (and hence for types), but names with a lower case initial for fields unless they are also names of categories. Noninitial letters are capitalized if they represent the first letter of an independent word in the name. (This is sometimes called "camel case" or "medial capitals".) Users who define their own type systems are of course not bound to this convention.

4 Type sheets

Categories can be defined by text called a **type declaration**. A document that contains one or more type declarations is called a **type sheet**.

The first line of a type sheet declares the name of the type system to be specified. Every of the following lines either creates a new category (via the name of the category followed by a colon), or specifies the category, by a keyword possibly followed by further specifications.

4.1 Proper types

A type declaration of a proper type has the following structure:

(i) the name of the proper type (followed by a colon)
(ii) then optional other proper types (each followed by a `+`) to inherit requirements from
(iii) then lines of requirements starting with certain keywords listed in Table 1 followed by a greater sign (`>`) and together with some arguments.

In (ii), the final `+` is missing if no lines of the form (iii) follow.

Example. We give a simple type declaration of some proper type `Norm`, to get acquainted with the syntax and the meaning of a type declaration. The type declaration

```
Norm:
allOf> entry=Expression
optional> index=Expression
```

expresses that any object #`obj` with #`obj.type` = `Norm` is required to have a constituent #`obj.entry` = #`e` with #`e` matching type `Expression`, and optionally it may have a sem #`obj.index` = #`i` with #`i` also matching type `Expression`.

operator	arguments	usage
allOf	list of equations	restricts entry of certain fields
oneOf	list of equations	restricts entry of certain fields
someOf	list of equations	restricts entry of certain fields
optional	list of equations	restricts entry of certain fields
fixed	list of equations	restricts entry of certain fields
only	list of equations	restricts entry of certain fields
someOfType	list of equations	restricts entry of certain fields
itself	list of names	restricts entry of certain fields
array	list of equations	restricts entry of certain fields
index	list of equations	requires to index each instance
template	one name	assigns a template
nothingElse	none	forbids further fields

Table 1: Keywords in declarations of proper types

4.2 Unions and atomics

Since a type declaration for a union may also declare a number of objects as atomic types, they are treated together in this subsection.

A type declaration of a union has the following structure:

(i) the name of the union type (followed by a colon)

(ii) followed by lines starting with certain keywords listed in Table 2 possibly followed by further specifications.

operator	arguments	usage
nothing	none	defines an atomic type
union	list of names	defines a union
atomic	list of names	defines a union of atomic types
complete	none	closes a union
index	list of equations	requires to index each instance

Table 2: Keywords in declarations of unions and atomics

4.3 Inheritance

Inheritance adds the specifications from an existing type declaration to a newly defined type declaration.

For example, if we want a proper type that uses all specifications of the type `Norm` as defined above, but adds an optional comment, the type declaration

```
NormWithComment:
allOf> entry=Expression
optional> index=Expression
          comment=String
```

is equivalent to the shorter version

```
NormWithComment: Norm +
optional> comment=String
```

that uses inheritance. Similarly, we can also define the **intersection** of two types. Given the type declaration

```
Comment:
optional> comment=String
```

we can equivalently define

```
NormWithComment: Norm + Comment
```

The same is possible for unions: if we assume a union `Document` that was defined by the type declaration

```
Document:
union> LatexDocument, PlainText
```

and now we want to add `SpreadSheet` as a further subtype, we write

```
Document: Document +
union> SpreadSheet
```

4.4 Templates

Assume a type declaration of a declared type #`D` containing the line `template> #C`. In this case, #`T` is called the **template** of #`D`. All the requirements form #`T` apply to #`D`, and additionally the requirements for #`D`.

There are several differences to inheritance:

- The SM stores the fact that the template of #`D` is #`T`, while inheritance is visible only on the type sheet level but (without closer analysis) not in the SM.
- Only proper types can have templates, while inheritance is also defined for unions.
- The proper type and the template may pose requirements on the same constituent.
- A proper type has at most one template of, while inheritance from multiple proper types is possible.

Templates are important for efficient programming with records. Indeed, graph walkers may handle all types with the same template using a single program rather than one for each such type; see [1]. If a type declaration does not specify a template, then the type is assumed to be its own template.

4.5 A grammar for type sheets

A txt document containing a number of type declarations is called a **type sheet**. The first line of a type sheet contains the type system it defines or enlarges. Every line in a type sheet beginning with an exclemation mark (`!`) is a comment.

The following context-free grammar defines type sheets as the sentences derivable from the starting symbol TYPESHEET. The token COMMENT is an arbitrary string beginning with an exclamation mark (`!`) and not containing a newline (`\n`).

TYPESHEET → HEADER BODY
HEADER → NAME ::
BODY → LINE | BODY \n LINE
LINE → UNION | DECLARED | \n | COMMENT \n
UNION → UNIONHEADER \n UNIONLINES
UNIONHEADER → NAME :
UNIONLINES → `union>` NAMESEQ | `atomic>` NAMESEQ | `complete>` | `index>` EQUATIONLINES
DECLARED → DECHEADER \n DECLINES
DECHEADER → NAME : | NAME : NAMESUM + | NAME : NAMESUM
DECLINES → DECKEYWORD > EQUATIONLINES | `itself>` NAMELINES | `template>` NAME | `nothingElse>` | `nothing>`
DECKEYWORD → `allOf` | `oneOf` | `someOf` | `optional` | `someOfType` | `fixed` | `array` | `index` | `only`
EQUATIONSEQ → EQUATION | EQUATIONSEQ , EQUATION
EQUATIONLINES → EQUATION | EQUATIONSEQ \n EQUATION
EQUATION → NAME = NAME
NAMESEQ → NAME | NAMESEQ , NAME
NAMELINES → NAME | NAMESEQ \n NAME
NAMESUM → NAME | NAMESEQ + NAME
NAME → A-z | NAME A-z | NAME 0-9

Each line contains a production with a nonterminal on the left side of the arrow (→), and a disjunction of strings of terminals and/or nonterminals on the right side, separated by a pipe (|). All words in capital letters are nonterminals, A-z and 0-9 denote the letters and digits respectively, `\n` denotes the "newline" character, and all other nonblank characters – in particular, `>`, `:`, `+`, `=`, `(`, `)` and `,` denote themselves.

An **annotated type sheet** has a more complex syntax, allowing for comments and rendering information. Annotated type sheets will be dicussed in detail elsewhere.

4.6 Consistency of type sheets

Assume that the type declaration of proper type #`D` specifies a template #`T`. A program that reads the type sheet must perform the following consistency checks:

- All atomic types declared in a type sheet have to be declared as subtypes of a union `Atomic`.
- If a union declares atomic types, these must not already exist.
- The order relation $<$ must be irreflective.
- When using a template, the added requirements are actually restrictions of the template.
- In a type declaration the left-hand sides following a particular operator (including those types declarations to inherit from) have to be unique.

If any of these requirements is violated, the type sheet reader issues an error.

5 Description and representation of type declarations in the SM

We will now describe informally which requirements each keyword in a type declaration poses on the object that has this type. We also define how type declarations and type systems are represented in the semantic memory.

5.1 Type declarations of proper types

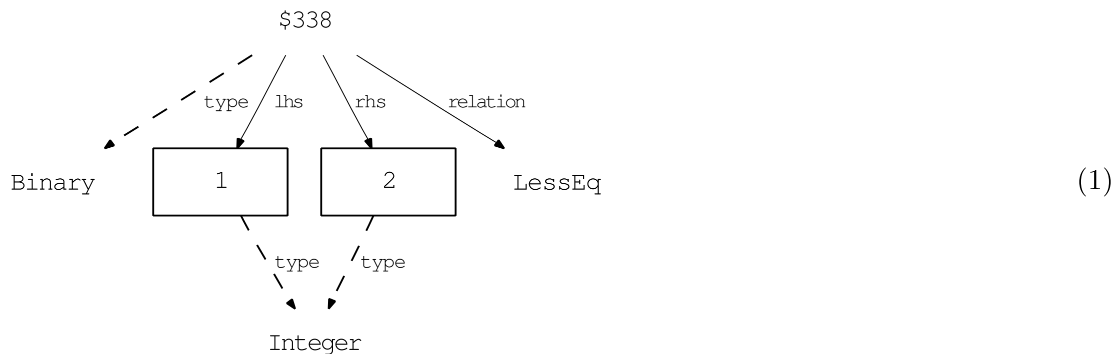
5.1.1 allOf

Via `allOf`, we require an object to have all of a collection of fields with entries of a certain kind.

Example. Consider a category `binary`, which we want to use to represent binary relations, e.g., in the representation of

$$1 \leq 2,$$

given in the following semantic graph:



(1)

We require constituents with fields both `lhs` and `rhs` and entries of type `Integer`, and we require a constituent with field `relation` and an entry which matches the type `RelationAtomic` (assuming `LessEq` << `RelationAtomic`).

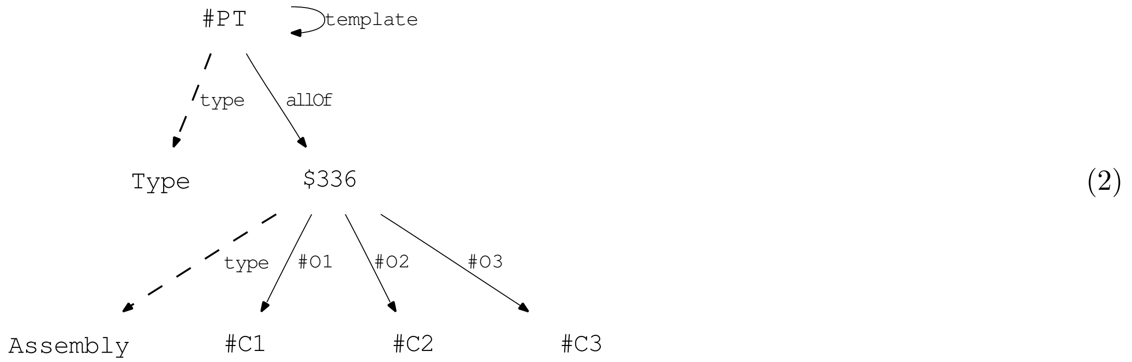
We can express these restrictions via the following type declaration:

```
LessEq:
allOf> lhs=Integer
       rhs=Integer
       relation=RelationAtomic
```

Representation in the SM. Consider a proper type #`PT` using `allOf`:

```
Test(TypeSystem)::

#PT:
allOf> #O1=#C1
       #O2=#C2
       #O3=#C3 ! etc.
```

This is stored in the SM as the following semantic graph:



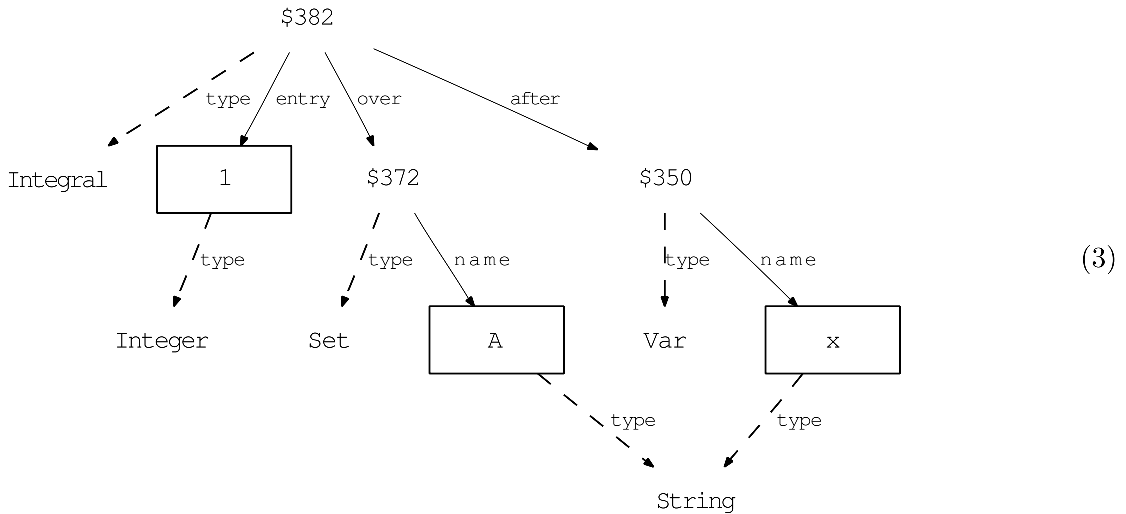
(2)

5.1.2 oneOf

Via `oneOf`, we require an object to have exactly one of a collection of fields with entries of a certain kind.

Example. An integral must have either a field `over` or a field `fromTo`, but not both. The following semantic graph gives the representation of

$$\int_A 1 \, \mathrm{d}x.$$



(3)

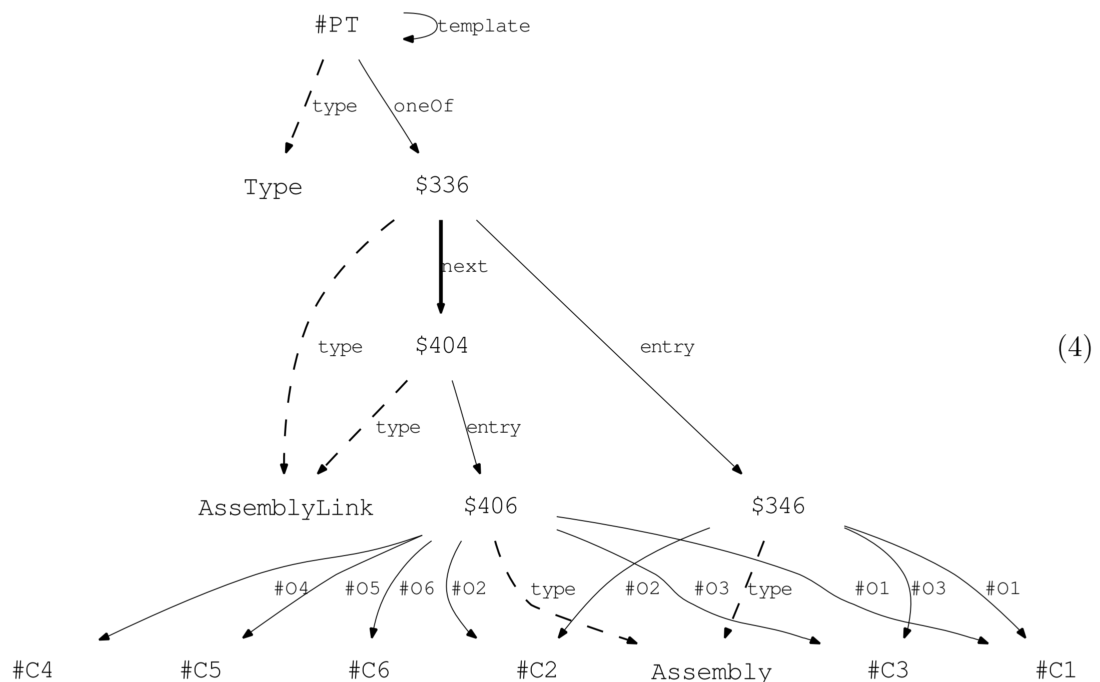
The restrictions we want to express are that the entry of the sem with field `over` must be a set, and the entry of the sem with field `fromTo` must be an expression. For this, we use the quantifier `oneOf` in the type declaration of `Integral`. We assume `Integer` $<<$ `Expression`.

```
Integral:
oneOf> fromTo=Expression
       over=Set
allOf> entry=Expression
       after=Var
```

Representation in the SM. Consider a proper type #`PT` using `oneOf`:

```
Test(TypeSystem)::

#PT:
oneOf > #O1=#C1
        #O2=#C2
        #O3=#C3 ! etc.
oneOf > #O4=#C4
        #O5=#C5
        #O6=#C6 ! etc.
```

This is stored in the SM as the following semantic graph:



(4)

5.1.3 someOf

Via `someOf`, we require an object to have at least one constituent with a field from a collection of fields with entries of a certain kind.

Example. The type declaration `Index` requires a subscript, a superscript, or a subscript or superscript on the left side, i.e, at least one of the positions **m**(#`obj/sub`), **m**(#`obj/sup`), **m**(#`obj/lsub`) and **m**(#`obj/lsup`) to be occupied by an expression. We express thiese requirements in the type declaration:

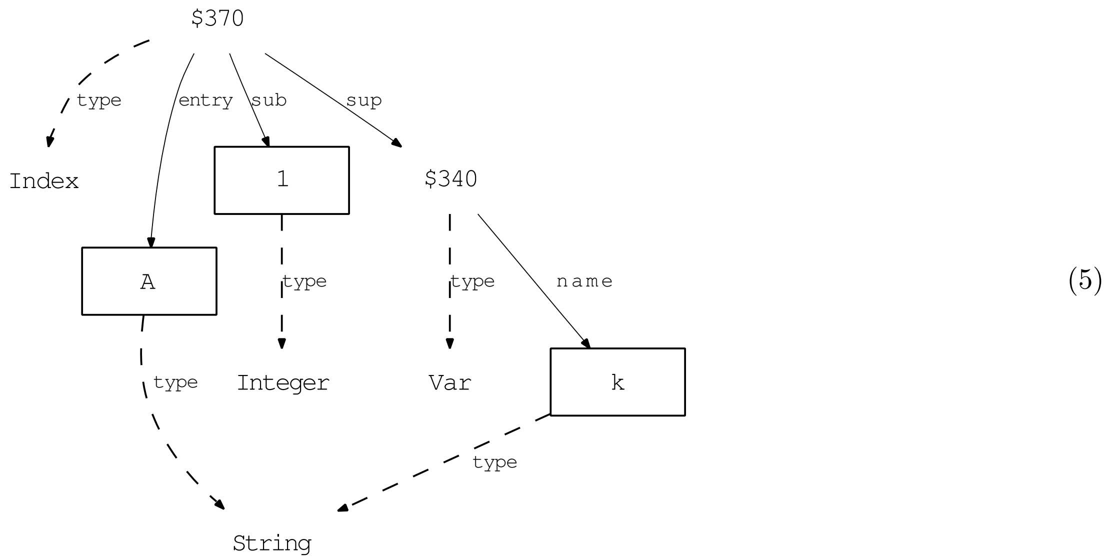
```
Index:
someOf> sub=Expression, sup=Expression, lsub=Expression, lsup=Expression
allOf> entry=Expression
```

We assume that the union `Expression` contains `Integer`, `String` and `Var`.

The expression

$$A_1^k$$

has both indices below and above, and is represented as:

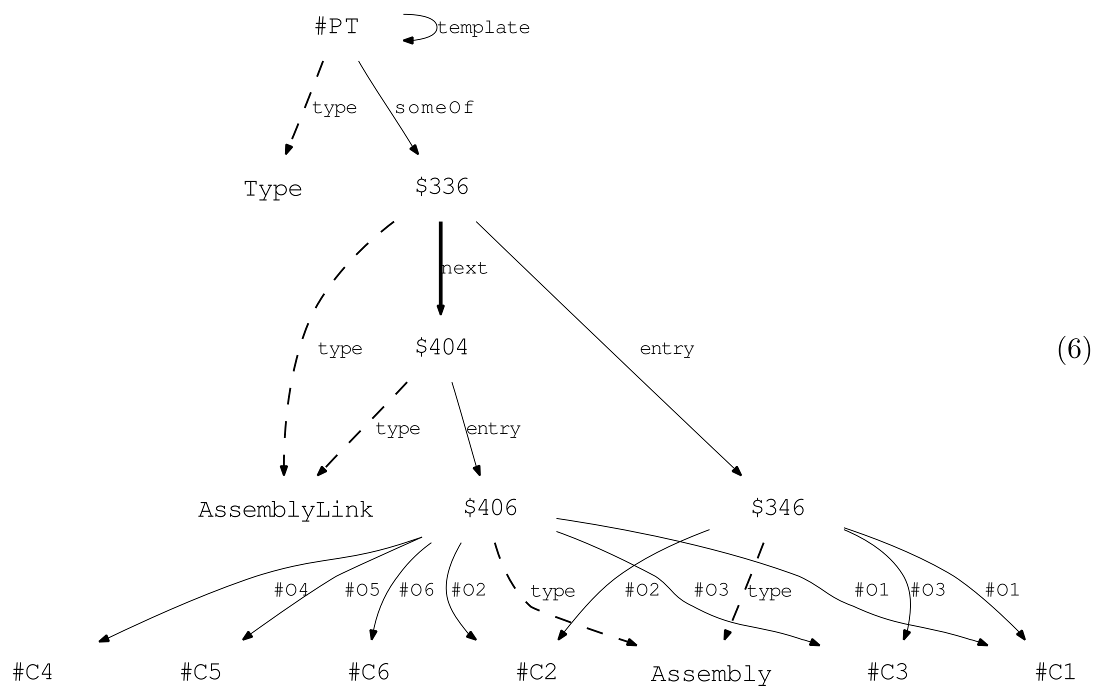


(5)

Representation in the SM. Consider a proper type #`PT` using `someOf`:

```
Test(TypeSystem)::

#PT:
someOf > #O1=#C1
         #O2=#C2
         #O3=#C3 ! etc.
someOf > #O4=#C4
         #O5=#C5
         #O6=#C6 ! etc.
```

This is stored in the SM as the following semantic graph:



(6)

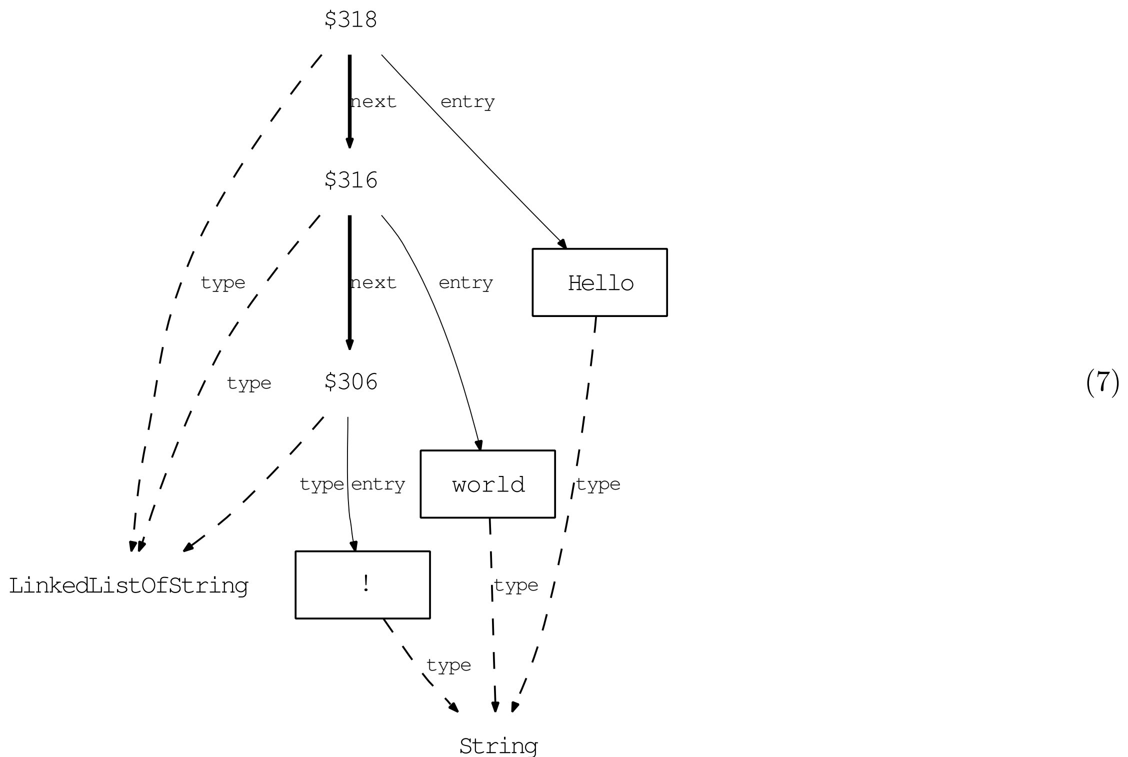
5.1.4 optional

Via `optional`, we require an object, if it has certain fields, to have entries of a certain kind.

Example. A linked list is a data structure in which there is a first value given, and every value, except the last value of the list, has a pointer to the next value. In the SM, a linked list consists of objects that all have a constituent with field `entry` and entry of some kind, and may have a constituent with field `next` that has another object of the linked list as entry. We express these restrictions for a linked list of strings in the type declaration:

```
LinkedListOfString:
allOf> entry=String
optional> next=LinkedListOfString
```

The linked list with entries "Hello", "world" and "!" is then given by the semantic graph:

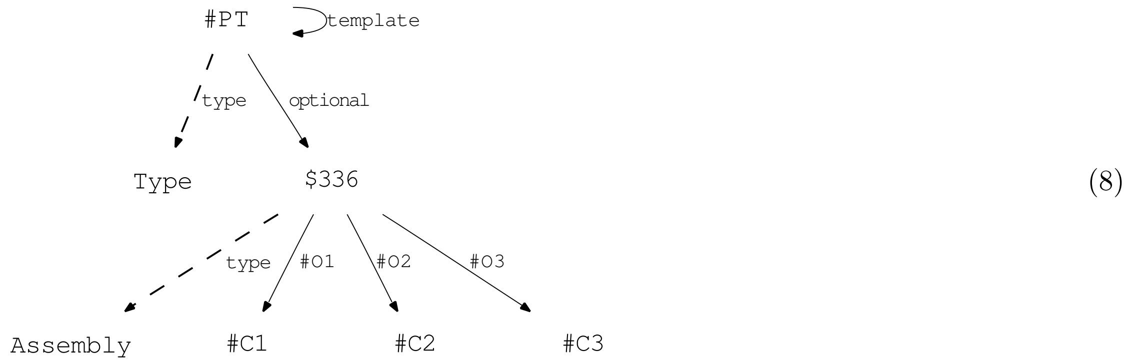


(7)

Representation in the SM. Consider a proper type #PT using `optional`:

```
Test(TypeSystem)::

#PT:
optional > #O1=#C1
           #O2=#C2
           #O3=#C3 ! etc.
```

This is stored in the SM as the following semantic graph:



(8)

5.1.5 fixed

Via `fixed`, we require an object to have a sem with given field and given entry.

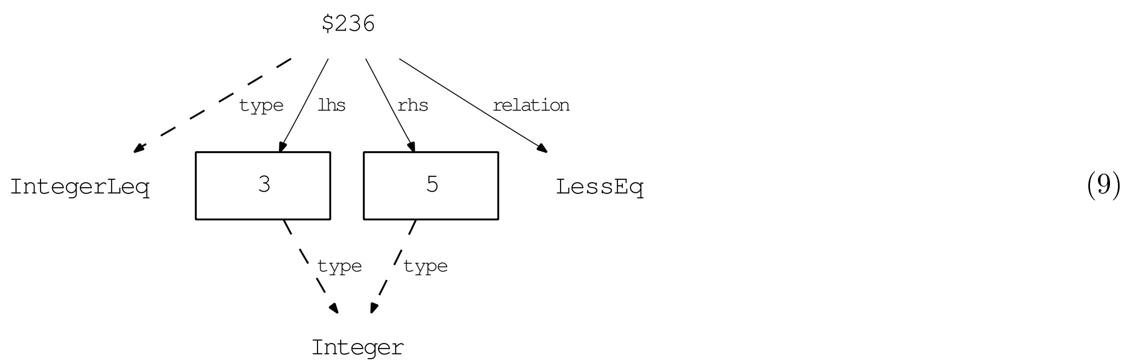
Example. We define a special binary relation `IntegerLessEq`:

```
IntegerLessEq:
allof> lhs=Integer, rhs=Integer
fixed> relation=LessEq
```

Then the relation

$$3 \leq 5$$

would be represented by:

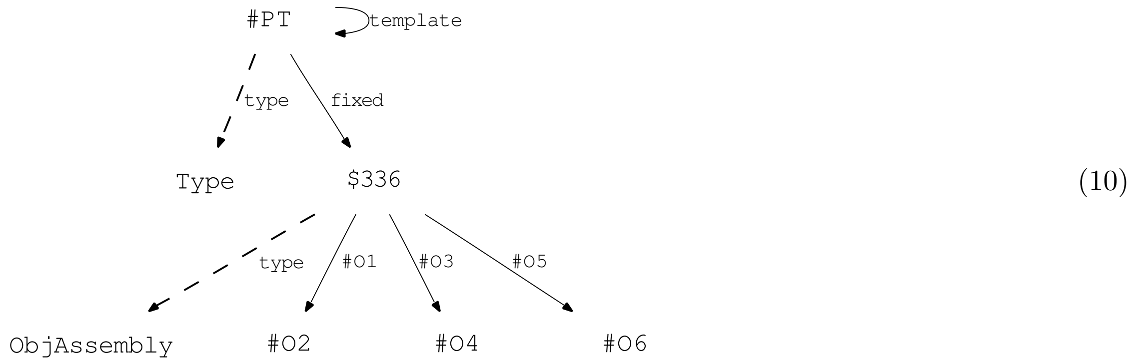


(9)

Representation in the SM. Consider a proper type #`PT` using `fixed`:

```
Test(TypeSystem)::

#PT:
fixed> #01=#02
       #03=#04
       #05=#06  ! etc.
```

This is stored in the SM as the following semantic graph:

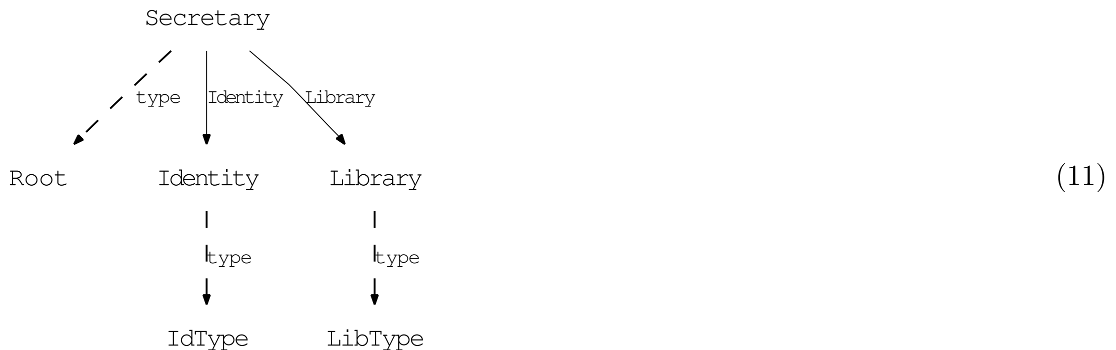


(10)

5.1.6 only

Via `only` we require an object to have all of a collection of sems where the field is the same object as the entry, and the entries need to be of a certain kind.

Example. Consider a category `Root`, which we want to use to represent root nodes of the semantic memory.



(11)

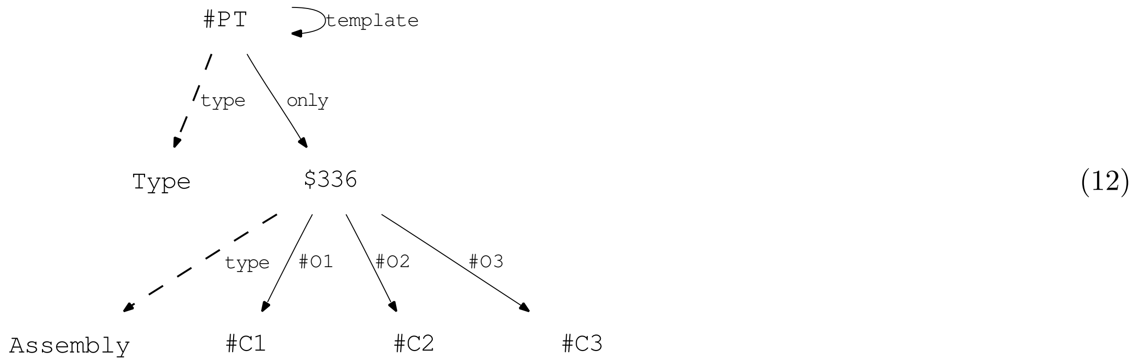
We require constituents with fields `Identity` and `Library` and entries equal to the fields, of type `IdType` and `LibType` respectively. We can express these restrictions via the following type declaration:

```
Root:
only> Identity=IdType
      Library=LibType
```

Representation in the SM. Consider a proper type #`PT` using `only`:

```
Test(TypeSystem)::

#PT:
only> #01=#C1
      #02=#C2
      #03=#C3 ! etc.
```

This is stored in the SM as the following semantic graph:



(12)

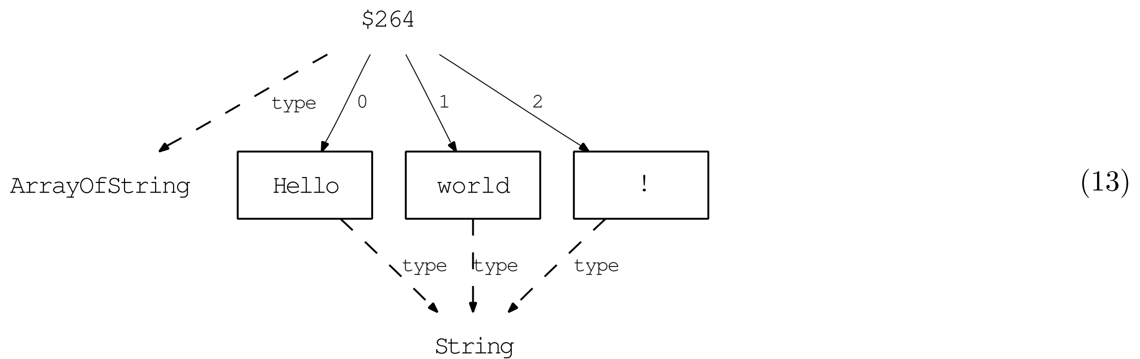
5.1.7 someOfType

Via `someOfType`, we require an object to have fields of a certain kind and entries of a certain kind.

Example. We define a random-access array of strings, where each string is accessable by an integer. So every constituent of this record has to have an integer as a field, and a String as an entry.

```
ArrayOfString:
someOfType> Integer=String
```

The following is the representation of the array of strings with entry 0 is “Hello”, entry 1 is “world”, and entry 2 is “!”.

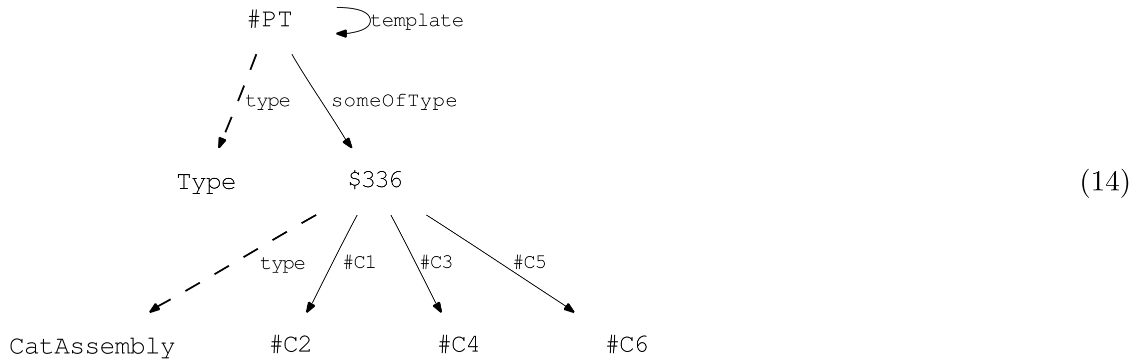


(13)

Representation in the SM. Consider a proper type #`PT` using `someOfType`:

```
Test(TypeSystem)::

#PT:
someOfType > #C1=#C2
             #C3=#C4
             #C5=#C6  ! etc.
```

This is stored in the SM as the following semantic graph:



(14)

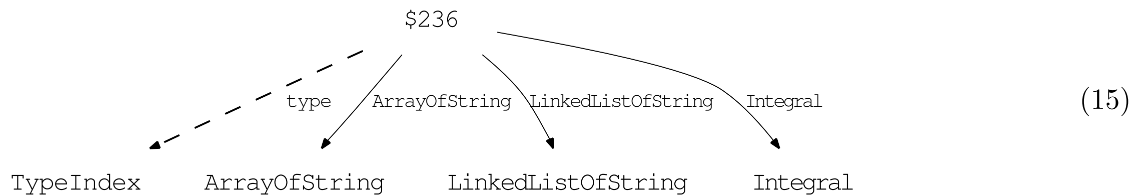
5.1.8 itself

Via `itself`, we require an object to have fields of a certain kind and entries equal to the field.

Example. We define an index of types:

```
TypeIndex:
itself> Type
```

We give an example of such an index containing the proper types `ArrayOfString` and `LinkedListOfString` and `Integer`.

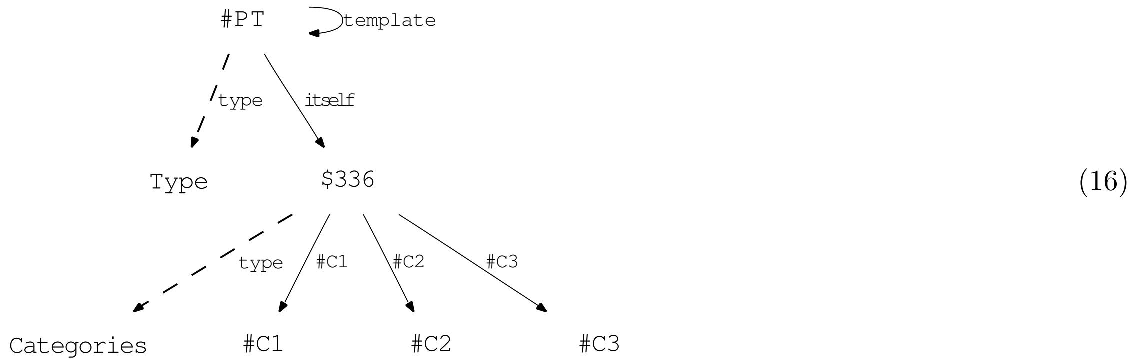


(15)

Representation in the SM. Consider a proper type #`DT` using `itself`:

```
Test(TypeSystem)::

#PT:
itself> #C1
        #C2
        #C3  ! etc.
```

This is stored in the SM as the following semantic graph:



(16)

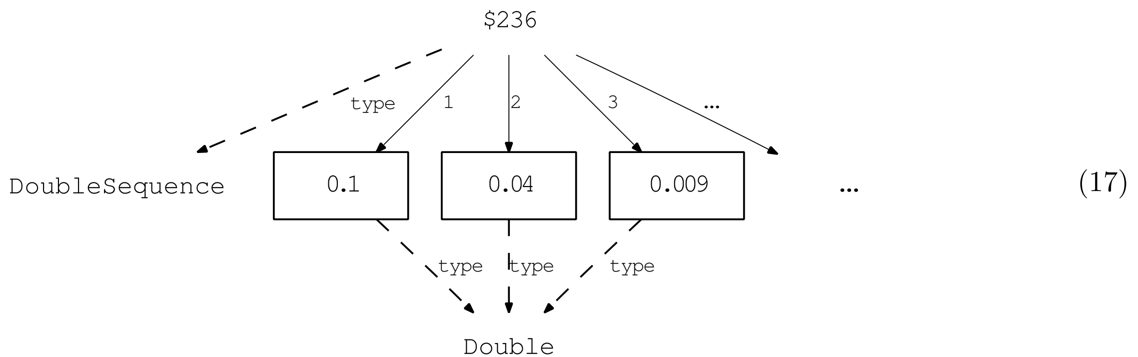
5.1.9 array

Scopes are objects describing a well-ordered set ob objects contained in the scope; see [1]. Via `array`, we require an object to have all the fields in a finite scope, and that their entries are of a certain kind.

Example. Consider a category `DoubleSequence` where the scope is the set of integers between 1 and 10 (represented by the object `From1To10`) and the entries are double precision floats:

```
DoubleSequence:
array> From1To10=Double
```

The sequence $(\frac{n^2}{10^n})_{n=1:10}$ would be represented by:

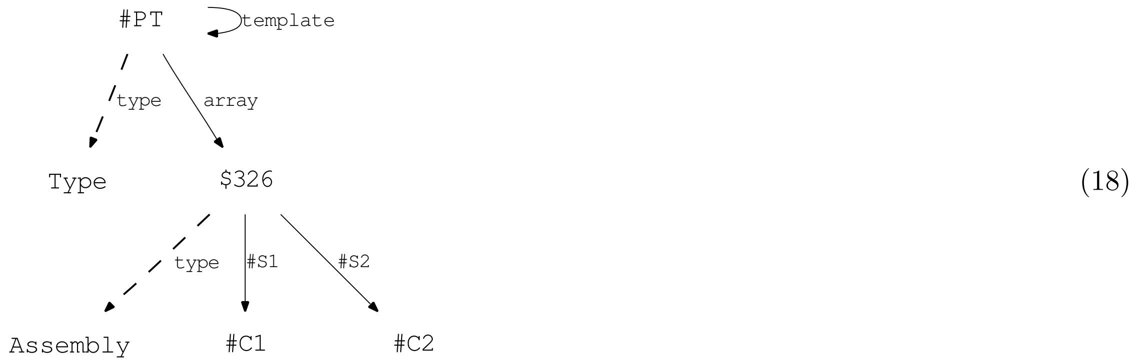


(17)

Representation in the SM. Consider a proper type `#PT` using `array`:

```
Test(TypeSystem)::

#PT:
array> #S1=#C1
       #S2=#C2  ! etc.
```

This is stored in the SM as the following semantic graph:



(18)

5.1.10 index

For some categories, we want to keep track of all their instances. Via `index`, we require each instance of some category to be listed in some assigned record.

Example. Consider a category `Equation`, with an object in `Lhs` and an object in `Rhs`, which can be expressed via `allOf`. But furthermore, each object that is of type `Equation` should be listed in a record `EquationList`, such that all objects in the semantic memory which are of type `Equation` can be found as fields of user defined object.

Assume the equation

$$x = y$$

represented in record #`rec`

(19)

which should be listed in the object `EquationIndex` :

(20)

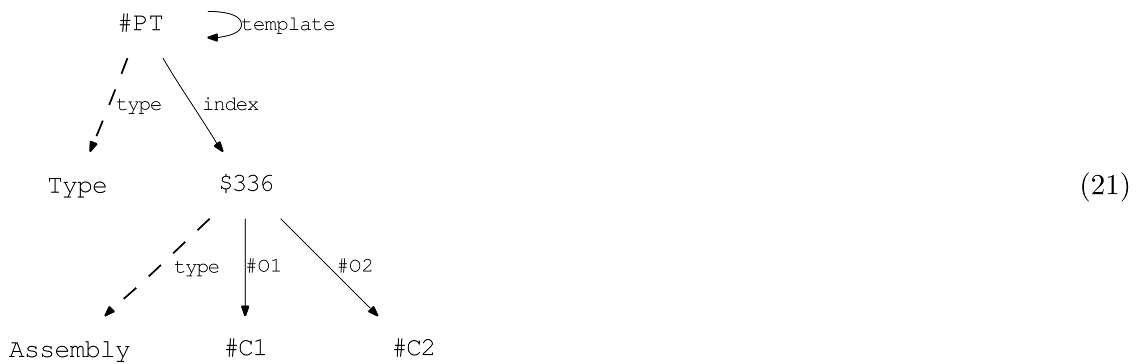
Besides the requirements on the constituents of the record of type `Equation` we now add requirements to an object that acts as an index of all records of this type, in this case, the object `EquationIndex`. All the equations are to be stored in `EquationIndex.ListOfEq`, and `EquationIndex.ListOfEq.type` = `ListOfEq` We do this by the type declaration:

```
ListOfEq:
itself> Equation

Equation:
allof> Lhs=Object, Rhs=Object
index> EquationIndex = ListOfEq
```

Representation in the SM. Consider a proper type #PT using `index`:

```
Test(TypeSystem)::

#PT:
index> #O1=#C1
       #O2=#C2  ! etc.
```

This is stored in the SM as the following semantic graph:



(21)

5.1.11 template

For representation of a type declaration containing the line `template> #C` is equivalent to inserting the body of the type declaration of `#C` in place of this line.

Example. The general form of a binary relation is required in the type `BinaryRel`.

```
BinaryRel:
allOf> lhs=Expression
       rhs=Expression
       relation=Type
```

This is used as a template for the more specific proper type `LessEq` which is used to express the relation $\leq$:

```
LessEq:
template> BinaryRel
allOf> lhs=Term
       rhs=Term
fixed> relation=LessEq
```

Representation in the SM. Consider a proper type #`PT` using `template`:

```
Test(TypeSystem)::

#PT:
template> #T1
```

This is stored in the SM as the following semantic graph:

(22)

5.1.12 nothingElse

Via `nothingElse`, we require an object to have *only* the required constituents and the field `type`.

Example. The type declaration `Var` should only have a constituent with field `name` and a string as entry, but no other constituents (except for the field `type` which is always present in a proper type).

```
Var:
allOf> name=String
nothingElse>
```

(23)

Representation in the SM. Consider a proper type #`DT` using `nothingElse`:

```
Test(TypeSystem)::

#PT:
nothingElse>
```

This is stored in the SM as the following semantic graph:

(24)

5.2 Type systems

A type system is an object #`TS` in the semantic memory with #`TS.type` = `TypeSystem`. All categories #`C` belonging to the type system are stored in #`TS`.#`C` = #`C`.

5.3 Type declarations of atomics

5.3.1 nothing

The operator `nothing>` defines an atomic type. Atomic types are objects that have a fixed semantic meaning, and must not have any constituents, not even a field `type`.

An atomic type does not pose any requirements on objects except itself, hence #`obj.type` = #`A` for some atomic type #`A` is meaningless.

Representation in the SM. Consider a type declaration of type #`A` using `nothing`, which defines the atomic type #`A`. Since #`A` must not have any constituents, it suffices to store that #`A` is part of the type system.

(Note that the typesheet reader also ensures that the type sheet contains a union `Atomic` that has all the atomic types of this type system as subtypes.)

```
Test(TypeSystem)::

#A:
nothing>
```

This is stored in the SM as the following semantic graph:

(25)

5.4 Type declarations of unions

5.4.1 union

A union defines the relation <, and hence also the relation << for a type system.

Example. We want to define the union `Rational` containing `Integer`, `Float` and `Double`, and the union `Number` containing `Integer`, `Float`, `Double`, `Rational` and `Real`. We specify this in the type sheet by

```
Rational:
union> Integer, Float, Double

Number:
union> Real, Rational
```

Note that `Float`, `Double`, `Real` and `Integer` have to be categories.

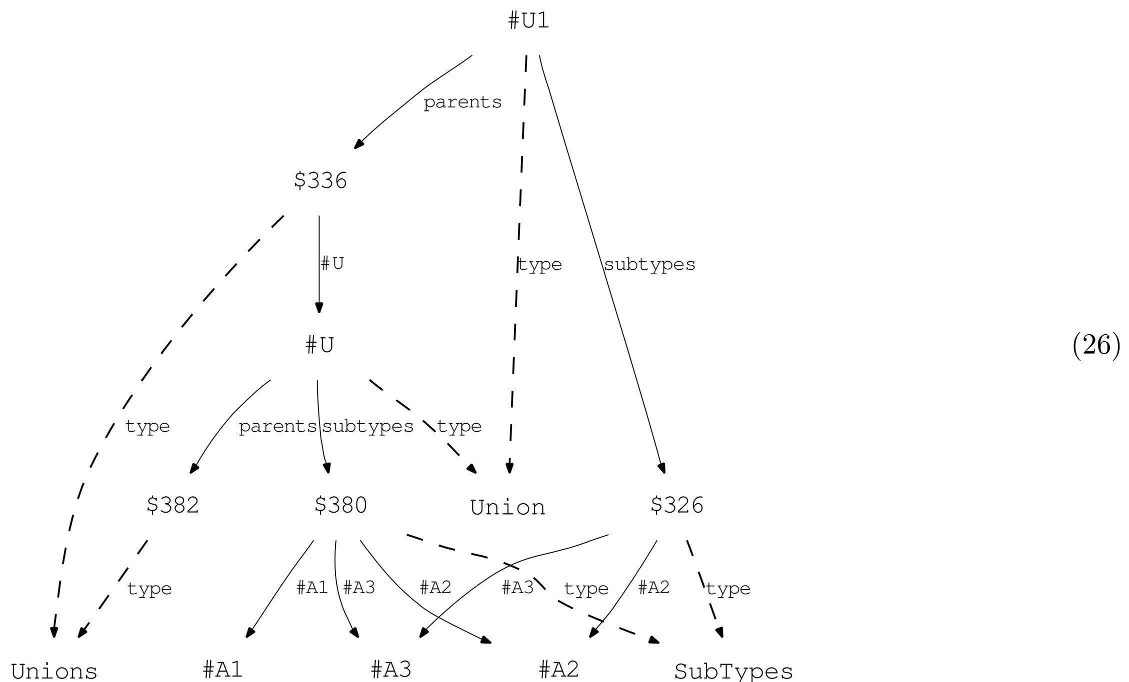
In the type sheet above, e.g., `Real` < `Number` and `Float` < `Real` are defined. Due to transitivity, e.g., `Float` << `Number` follows.

Representation in the SM. Consider a union #`U`:

```
Test(TypeSystem)::

#U:
union> #A1, #U1 !etc.
```

where #`U1` is a union of the atomics #`A2` and #`A3`.

This is stored in the SM as the following semantic graph:



(26)

A union #`U` knows about all minimal categories #`C` with #`C` << #`U`, this is necessary for matching. And #`U` has to know its immediate parents, i.e., categories #`C` with #`U` < #`C` to be able to recursively propagate new categories contained by #`U` upwards.

5.4.2 atomic

The operator `atomic` defines a type as a union of atomics. The atomics need not exist at that point, so as a byproduct, this may result in the definition of new atomic types.

```
#TD:
atomic> #A1, ... , #Ak
```

This is a short-hand notation for

```
#A1, ... , #Ak:
nothing>

#TD:
union> #A1, ... , #Ak
```

5.4.3 complete

This operator declares that no further categories can be added to a union.

Usually, one may add more categories to a union later, e.g.:

```
Number: Number +
union> Complex
```

But this is forbidden if the definition of `Number` contains a line `complete>`.

Example. The type declaration `Documents` should only contain the categories `Article`, `Report` and `Book`, but not anything else.

```
Documents:
union> Article, Report, Book
complete>
```

While it is still possible to later define a new category `Shortbook` with the property `Shortbook` $<<$ `Documents`, e.g., with the declaration

```
Book:
union> Shortbook
```

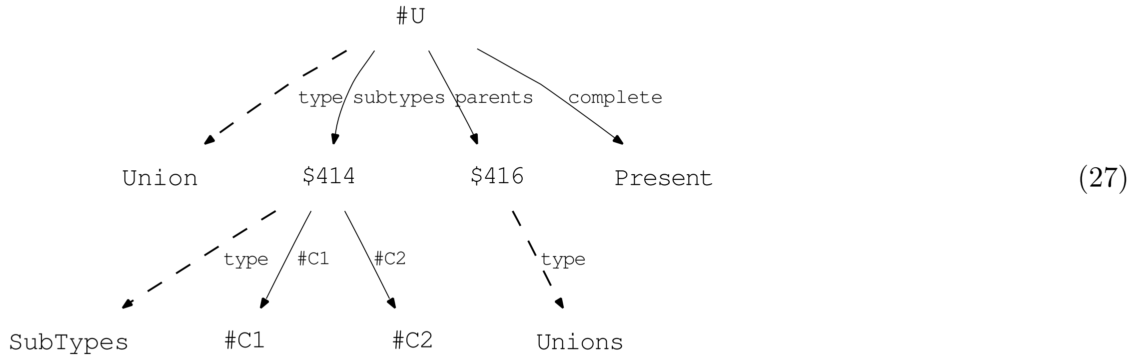
it is not possible to add `Shortbook` with the property `Shortbook` $<$ `Documents`, e.g., with the declaration

```
Documents: Documents +
union> Shortbook
```

Representation in the SM. Consider a union #`U` using `complete`:

```
Test(TypeSystem)::

#U:
union> #C1, #C2  ! etc.
complete>
```

This is stored in the SM as the following semantic graph:



(27)

5.4.4 index

This applies the requirement to index all instances to all the subtypes of a union.

Example. Assume we want to have an index that lists all instances of inequalities. Inequalities are the objects that have either type `LessEq`, `Less`, `GreaterEq` or `Greater`, and we want them to be indexed in the object `IndexOfIneq`.

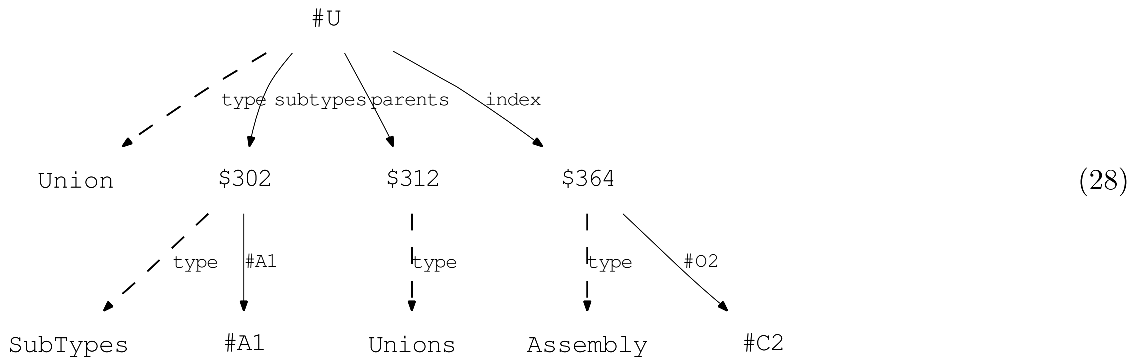
```
Inequality:
union> LessEq, Less, GreaterEq, Greater
index> IndexOfIneq=Inequalities
```

Representation in the SM. Consider a union #`U` using `index`:

```
Test(TypeSystem)::

#U:
atomic> #A1
index> #O1=#C1
       #O2=#C2  ! etc.
```

(This type declaration also declares an atomic type, which is necessary to characterize it as a union, else it would be treated as a proper type according to Subsection 5.1.10.)

This is stored in the SM as the following semantic graph:



(28)

This information is propagated to all the subtypes of #`U`, and represented in the semantic memory according to Subsection 5.1.10.

6 Well-typed records

Assume a record with handle #rec, and let the type of #rec be #T.

To define when this record is well-typed, we first define which position of an object are a **declared position** and in which case a position is **faulty**.

If no position reachable from the handle #rec is faulty, then the record is **well-typed** of type #T.

Otherwise the record is **ill-typed**.

Note that it can be checked in time linear in the number of sems that are reachable whether or not the record is well-typed.

We consider an object #obj with #obj.type = #TD and #TD.type = Type.

6.1 allOf

For every field #f, #f≠type of #TD.allOf, (#obj/#f) is a declared position of #obj.

If for one of the declared positions (#obj/#f) is empty, or not **m**(#obj.#f/#TD.allOf.#f) then (#obj/#f) is faulty.

6.2 oneOf

For all $k = 0, 1, 2, \dots$ and for every field #f, #f ≠ type of #TD.oneOf.next^k.entry, the position (#obj/#f) is a declared position of #obj.

If a declared position (#obj/#f) is nonempty and not **m**(#obj.#f/#TD.oneOf.next^k.entry.#f) then it is faulty.

If not for all k exactly one of the declared postions (#obj/#TD.oneOf.next^k.entry.#f) is occupied then the the position (#obj/#f) is faulty.

6.3 someOf

For all $k = 0, 1, 2, \dots$ and for every field #f, #f ≠ type of #TD.someOf.next^k.entry, the position (#obj/#f) is a declared position of #obj.

If a declared position (#obj/#f) is nonempty and not **m**(#obj.#f/#TD.someOf.next^k.entry.#f) then it is faulty.

If not for all k at least one of the declared postions (#obj/#TD.someOf.next^k.entry.#f) is occupied then the the position (#obj/#f) is faulty.

6.4 optional

For every field #f, #f ≠ type of #TD.optional, (#obj/#f) is a declared position of #obj.

If a declared position (#obj/#f) is nonempty and not **m**(#obj.#f/#TD.optional.#f) then it is faulty.

6.5 fixed

For every field #f, #f≠type of #TD.fixed, (#obj/#f) is a declared position of #obj. If a declared positions (#obj/#f) is either empty or does not satisfy #obj.#f = #TD.fixed.#f then (#obj/#f) is faulty.

6.6 only

For every field #f, #f $\neq$ type of #TD.only, every position (#obj/#f) is a declared position of #obj. If a declared positions #obj.#f does not satisfy both #obj.#f = #f and **m**(#obj.#f/#TD.only.#f) then (#obj/#f) is faulty.

6.7 array

Not implemented yet.

6.8 itself

For every field #f, #f $\neq$ type of #TD.itself, every position (#obj/#F) with **m**(#F/#f) is a declared position of #obj. For all declared positions (#obj/#F), if #obj.#F $\neq$ #F then (#obj/#F) is faulty.

6.9 someOfType

For every field #f, #f $\neq$ type of #TD.someOfType, every position (#obj/#F) with **m**(#F/#f) is a declared position of #obj. If a declared position (#obj/#F) does not satisfy **m**(#obj.#F/#TD.someOfTyp
then (#obj/#f) is faulty.

6.10 nothingelse

If #TD.nothingelse = Present, and there exists an occupied position (#obj/#f) of #obj that is not a declared position, then (#obj/#f) is faulty.

7 Type declarations and unions as types

In this section, we give a type system that defines the type of a type system, both as a type sheet and represented in the semantic matrix. As a type sheet, the type of a type system has 40 lines. When represented in the semantic memory, the record has 126 sems. check!

```
BasicTypes::

Type:
  index> Index = Types
  allOf> template=Type
  someOf> allOf=Assembly
          someOf=AssemblyLink
          optional=Assembly
          oneOf=AssemblyLink
          someOfType=CatAssembly
          index=Assembly
          itself=Categories
          nothingElse=Present
          fixed=ObjAssembly
          array=Assembly
          only=Assembly
  optional> index=Assembly
            parents=Unions
```

```
            extends=Type
  ! array can be tightened when specified

Atomic:
  atomic> Present
  index> Index = Atomics

Union:
  index> Index = Unions
  allOf> subtypes = SubTypes
         union=Categories
  optional> atomic=Atomics
            complete=Present
            parents=Unions
            extends=Union
            index=Assembly

SubType:
  union> Atomic, Type

Category:
  union> Union, Type, Atomic

TypeSystem:
  index> Index = TypeSystems
  itself> Category


! * BASIC COLLECTIONS *

Atomics:
  itself> Atomic

Types:
  itself> Type

SubTypes:
  itself> SubType

Unions:
  itself> Union

Categories:
  itself> Category

TypeSystems:
  itself> TypeSystem

! * BASIC DEFINITIONS *

Assembly:
  someOfType> Object=Category

ObjAssembly:
```

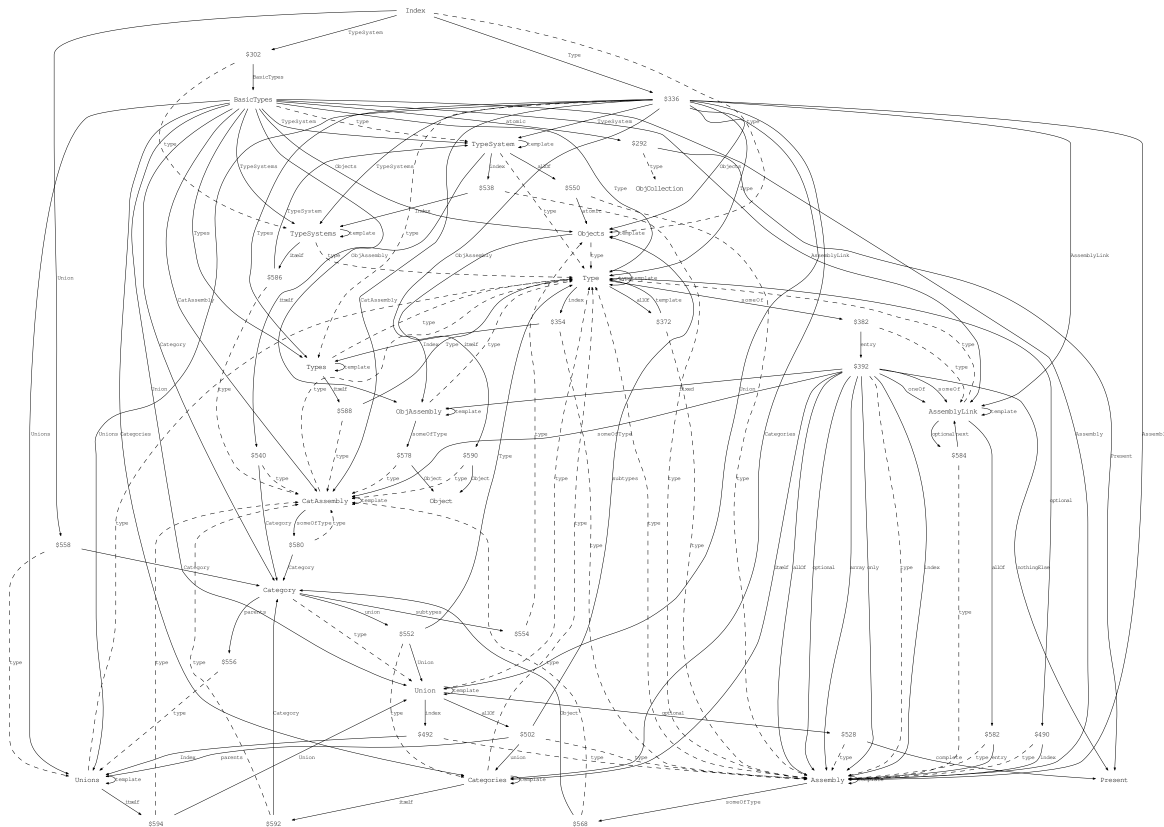
```
  someOfType> Object=Object      ! This does nothing

CatAssembly:
  someOfType> Category=Category

AssemblyLink:
  allOf> entry=Assembly
  optional> next=AssemblyLink
```

This type sheet is represented in the semantic memory ba the following set of sems:

```
Index.type=Objects
Index.Type=$336
Index.Union=$558
Index.TypeSystem=$302
Objects.type=Type
Objects.template=Objects
Objects.itself=$590
Type.type=Type
Type.template=Type
Type.index=$354
Type.allOf=$372
Type.someOf=$382
Type.optional=$490
BasicTypes.type=TypeSystem
BasicTypes.Objects=Objects
BasicTypes.Type=Type
BasicTypes.Union=Union
BasicTypes.TypeSystem=TypeSystem
BasicTypes.atomic=$292
BasicTypes.TypeSystems=TypeSystems
BasicTypes.Types=Types
BasicTypes.Assembly=Assembly
BasicTypes.AssemblyLink=AssemblyLink
BasicTypes.CatAssembly=CatAssembly
BasicTypes.Categories=Categories
BasicTypes.ObjAssembly=ObjAssembly
BasicTypes.Unions=Unions
BasicTypes.Category=Category
Union.type=Type
Union.template=Union
Union.index=$492
Union.allOf=$502
Union.optional=$528
TypeSystem.type=Type
TypeSystem.template=TypeSystem
TypeSystem.index=$538
TypeSystem.allOf=$550
TypeSystem.itself=$540
$292.type=ObjCollection
$292.Present=Present
$302.type=TypeSystems
$302.BasicTypes=BasicTypes
TypeSystems.type=Type
TypeSystems.template=TypeSystems
TypeSystems.itself=$586
$336.type=Types
$336.Objects=Objects
$336.Type=Type
$336.Union=Union
$336.TypeSystem=TypeSystem
$336.TypeSystems=TypeSystems
$336.Types=Types
$336.Assembly=Assembly
$336.AssemblyLink=AssemblyLink
$336.CatAssembly=CatAssembly
$336.Categories=Categories
$336.ObjAssembly=ObjAssembly
$336.Unions=Unions
Types.type=Type
Types.template=Types
Types.itself=$588
$354.type=Assembly
$354.Index=Types
Assembly.type=Type
Assembly.template=Assembly
Assembly.someOfType=$568
$372.type=Assembly
$372.template=Type
$382.type=AssemblyLink
$382.entry=$392
AssemblyLink.type=Type
AssemblyLink.template=AssemblyLink
AssemblyLink.allOf=$582
AssemblyLink.optional=$584
$392.type=Assembly
$392.index=Assembly
$392.allOf=Assembly
$392.someOf=AssemblyLink
$392.optional=Assembly
$392.oneOf=AssemblyLink
$392.someOfType=CatAssembly
$392.itself=Categories
$392.nothingElse=Present
$392.fixed=ObjAssembly
$392.array=Assembly
$392.only=Assembly
CatAssembly.type=Type
CatAssembly.template=CatAssembly
CatAssembly.someOfType=$580
Categories.type=Type
Categories.template=Categories
Categories.itself=$592
ObjAssembly.type=Type
ObjAssembly.template=ObjAssembly
ObjAssembly.someOfType=$578
$490.type=Assembly
$490.index=Assembly
$492.type=Assembly
$492.Index=Unions
Unions.type=Type
Unions.template=Unions
Unions.itself=$594
$502.type=Assembly
$502.subtypes=Objects
```

```
$502.union=Categories
$502.parents=Unions
$528.type=Assembly
$528.complete=Present
$538.type=Assembly
$538.Index=TypeSystems
$540.type=CatAssembly
$540.Category=Category
Category.type=Union
Category.subtypes=$554
Category.union=$552
Category.parents=$556
$550.type=Assembly
$550.atomic=Objects
$552.type=Categories
$552.Type=Type
$552.Union=Union
$554.type=Objects
$556.type=Unions
$558.type=Unions
$558.Category=Category
$568.type=CatAssembly
$568.Object=Category
$578.type=CatAssembly
$578.Object=Object
$580.type=CatAssembly
$580.Category=Category
$582.type=Assembly
$582.entry=Assembly
$584.type=Assembly
$584.next=AssemblyLink
$586.type=CatAssembly
$586.TypeSystem=TypeSystem
$588.type=CatAssembly
$588.Type=Type
$590.type=CatAssembly
$590.Object=Object
$592.type=CatAssembly
$592.Category=Category
$594.type=CatAssembly
$594.Union=Union
```

The same set of sems displayed as a semantic graph:



References

[1] F. Domes, K. Kofler, A. Neumaier, and P. Schodl. CONCISE – The FMathL programming system. *Manuscript*, 2010.

[2] D. Lee and W.W. Chu. Comparative analysis of six XML schema languages. *ACM Sigmod Record*, 29(3):76–87, 2000.
Also available as `http://pike.psu.edu/publications/sigmod-record-00.pdf`.

[3] T.B. Lee, J. Hendler, O. Lassila, et al. The semantic web. *Scientific American*, 284(5):34–43, 2001.

[4] A. Neumaier. The FMathL mathematical framework. *Draft Version 1.04*, 2009. Available at "http://www.mat.univie.ac.at/ neum/ms/fmathl.pdf".