

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	3
4	Type sheets	5
5	The category of a position	8
6	Well-typed records	23
7	Representation of type declarations in the SM	23
8	Type declarations and unions as types	30

1 Introduction

In order to provide a good basis for the semantical analysis of mathematical language expressing arbitrary mathematical content, 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. Section 2 introduces basic formal notions such as **objects**, **sems**, **records**, etc. and discusses how these are interpreted informally. Section 3 defines how typing applies to objects: requirements can be posed on the constituents of a record in various forms, and furthermore inheritance and subtyping is possible. Section 4 discusses how requirements can be expressed in text documents called type sheets. Section 5 discusses how to interpret type declarations, i.e., defines when the requirements posed on a record are met. In Section 6 we define when a record is **well-typed**. Section 7 defines how declarations, which are the defining texts written by the user, are represented in the framework. In Section 8 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 is inspired by, and representable in, the semantic web [3].

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. Objects may, but need not have **names**, i.e., alphanumeric strings not beginning with a digit; different objects have different names, if any. Unnamed objects may be referred to by artificial names beginning with a dollar-sign (\$) followed by an alphanumeric string. **Empty** is the name of 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=Name` for every object `#name` representing a name,

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

x^2 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`, both not `Empty`. 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 **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`. 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`.

A **record** is a handle `#h` such that only finitely many objects are reachable from `#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 have 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. 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. Also, different nodes of the semantic graph may represent the same object. For example, the information $\frac{12}{4} = 3$ may be represented by the semantic graph in Figure 1.

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

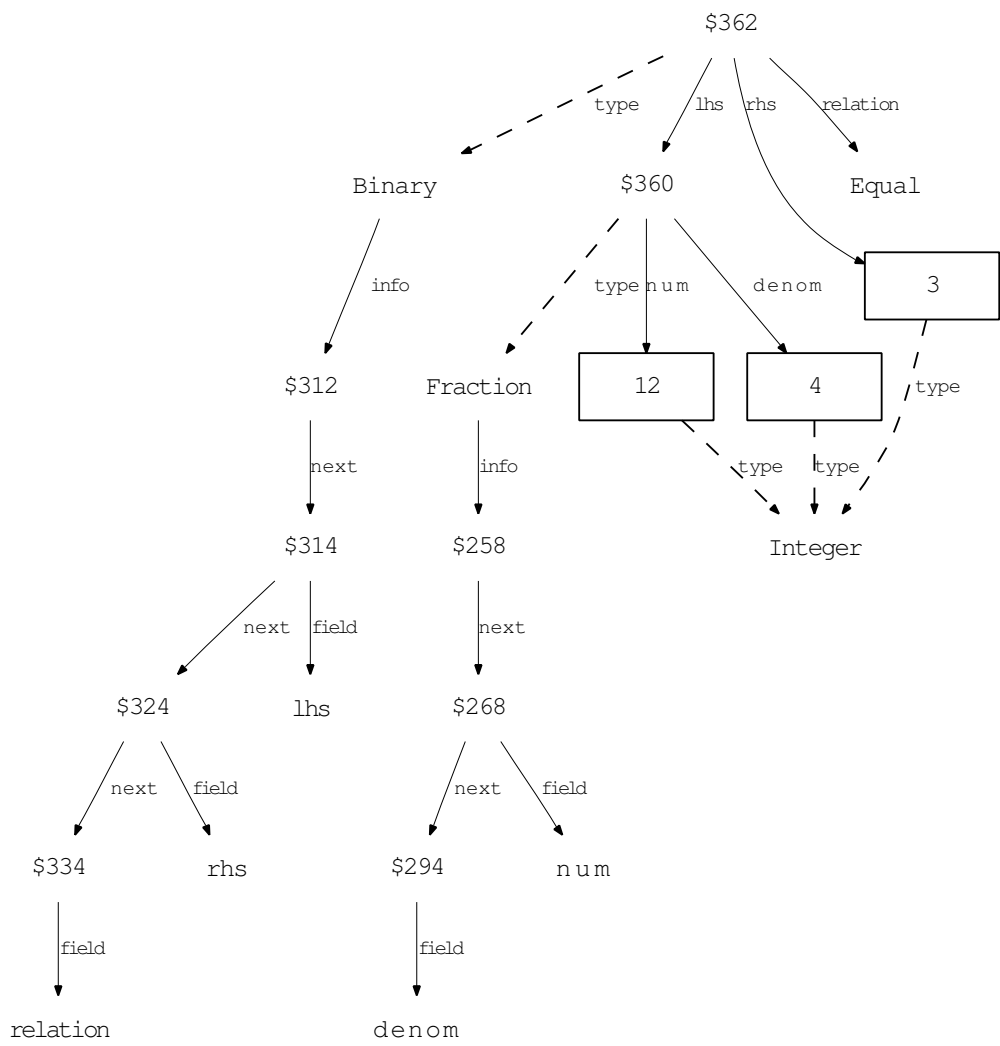


Figure 1: A semantic graph

examine the whole graph every time a record id used. For this reason we assign types to objects (in Section 3.1) and categories to sems (in Section 5). These assignments are always made with respect to a particular type system. In the following, until mentioned otherwise, we always consider a fixed but arbitrary type system $\#TS$ and its associated order relations. In Section 8, we define the type system **BasicTypes** with which one can describe the structure of arbitrary type systems.

A **type system** specifies that certain objects are **categories** of that type system. The object **Empty** is never a category.

In every type system, the set of categories is ordered by an irreflexive partial order relation $<$. If for the categories $\#C1$ and $\#C2$ the relation $\#C1 < \#C2$ holds, we say that $\#C2$ is **the union of $\#C1$** .

We define the relation \ll to be the reflexive and transitive closure of the relation $<$, i.e., $\#C1 \ll \#C2$ iff $\#C1 = \#C2$ or there exist categories $\#c1, \dots, \#cn$ such that $\#C1 < \#c1 < \dots < \#cn < \#C2$. If $\#C1 \ll \#C2$ we say that $\#C2$ **contains $\#C1$** .

A category is called a **type** if it is minimal in the ordering \ll , and a **union** otherwise. Types come in three forms: the **default type Object**, **atomic types**, and **proper types**. All categories except **Object** are declared in the record whose handle is the type system. This record can be created by means of a so-called **type sheet**, a piece of text containing the specifications.

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. The formulation of such requirements is discussed in Section 4, their interpretation in Section 5, and their representation in Section 7.

3.1 The type of an object

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**.

We say that an object **#obj** **matches** a category **#C**, in symbols: **MATCHES(#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

This section discusses which requirements can be posed, and how. Categories can be defined by text called a **type declaration**. A document that contains one or more type declarations is called a **type sheet**.

How to interpret type declarations rigorously and how to derive the types of the constituents of **#obj** is discussed in detail in Section 5. In Section 7 we will discuss how to actually store them in the SM.

The following context-free grammar defines type sheets as the sentences derivable from the starting symbol **TYPESHEET**.

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

An **annotated type sheet** has a more complex syntax, allowing for comments and rendering information. Comments start with an unescaped exclamation mark (!) and end with a newline (\n). Other details about annotation will be described elsewhere.

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.

Every line in a type sheet 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 a greater sign (>) and 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 Tables 1 and 2 (and explained in Section 5) together with some of their properties.

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

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

Table 1: Keywords in declarations of unions and atomics

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. More detailed explanations are given in Section 5. The type declaration

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

Table 2: Keywords in declarations of proper types

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`.

Inheritance. Inheritance adds the specifications from an existing proper type `#T` to a newly defined proper type `#t`.

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
```

A particular field may only have requirements in either the type sheet itself, or in exactly one of the proper types to inherit from. If this is not the case, the type sheet reader issues an error.

Consistency. If #T is the template of the proper type #D, then the requirements in #D and #T have to be consistent. This means the following:

If for a certain field, the entry required by #T is a type #c, then the entry required by #D has to match #c. If the entry required by #T is a union #u, then the entry required by #D has to match #u, or, if #D requires a union #U that does not match #u, then for every minimal element #c with #c<<#U the relation #c<<#u has to hold. If this is not the case, the type sheet reader issues an error.

5 The category of a position

A type declaration for a type #T distinguishes certain positions whose handle is an object #o of type #T in a way now to be discussed; these are called **declared positions**. It also poses requirements on the possible entries in a declared position (#o,#f). If these conditions are not met, the position is called **faulty**; otherwise it is assigned a particular category. An object #obj is called **ill-typed** if some declared position of #obj is faulty. If some object reachable from record #rec is ill-typed then #rec ill-typed. Otherwise the record #rec is **well-typed**.

For some record #rec with the type of #rec being a proper type #DT, we discuss how to interpret the type sheet containing #DT, and how to associate a category to declared sems.

5.1 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.

General definition. Assume the following type declaration:

```
#A:  
nothing>
```

Consider an object #obj with `#obj.type=#TD`. Then the position (`#obj.type`) is faulty.

If #A has a constituent `#A.#f=#e` then position (`#A,#f`) is faulty. Note that in this case, there is not even a position (`#A,type`) allowed.

For convenience, the atomics #A1 ... #Ak may be defined in one type declaration:

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

5.2 union

A union defines the relation <, and hence also the relation << for a type system. If this relation is not irreflexive, or if some #aj is not atomic, the type sheet reader issues an error.

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`, `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.

General definition. Assume the following type declaration

```
#TD:
union> #C1, ... , #Ck
```

where the C_i are existing categories.

This defines that `#C1<#TD`, ..., `#Ck<#TD` in the actual type system.

5.3 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.

General definition. Assume the following type declaration:

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

This is a short-hand notation for

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

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

5.4 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
```

General definition. Assume the following type declaration:

```
#U:
complete>
```

Where `#U` is a union of `#u1, ..., #uk`. Then every declaration that states that `#C<#U` for `#C` distinct from all `#u1, ..., #uk` produces an error.

Templates. A proper type `#D` may refer to another proper type `#T` as a **template**. 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].

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 “lower or equal” (\leq):

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

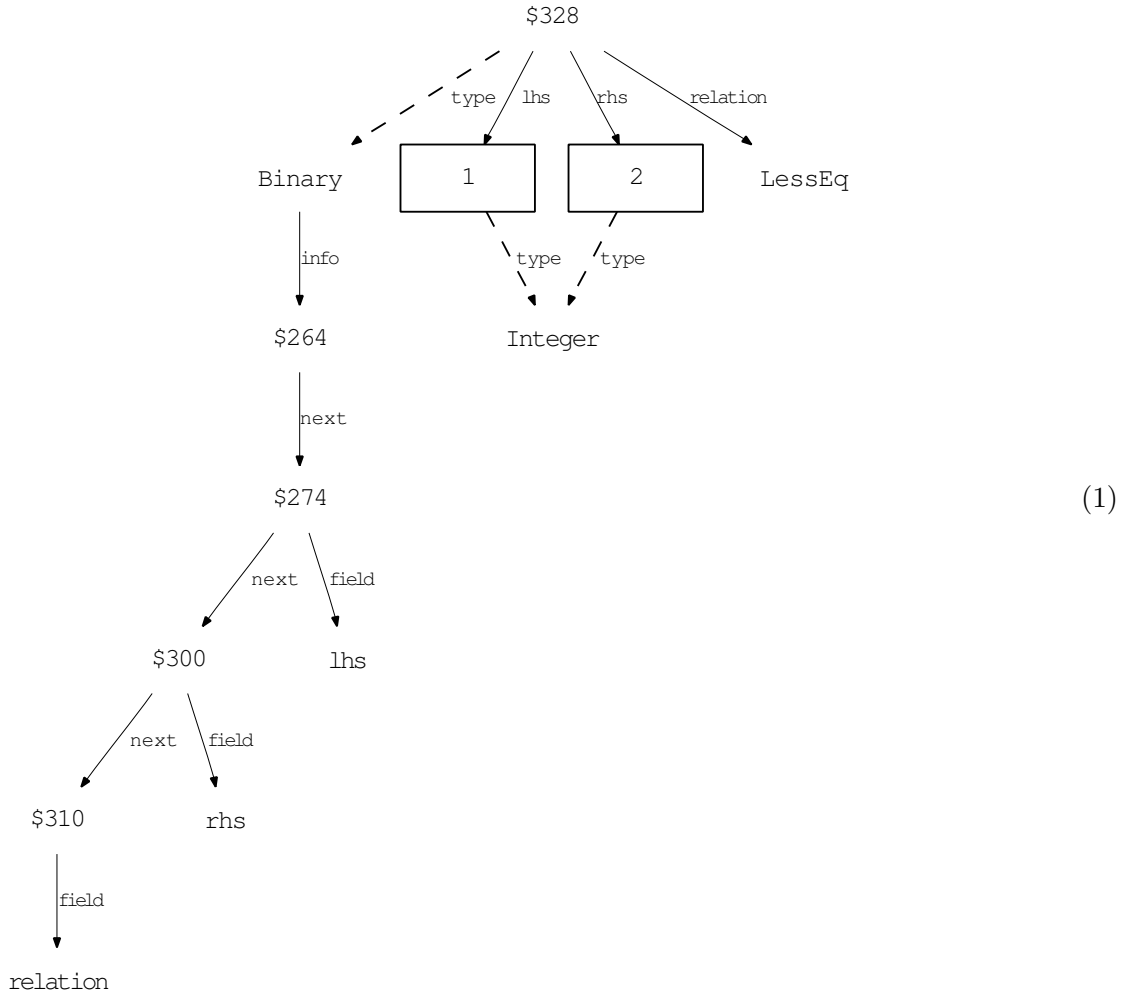
5.5 allOf

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:



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 is an attributed name (defined via a type `AttributedName`).

We can express these restrictions via the following type declaration:

```
LessEq:
allOf> lhs=Integer, rhs=Integer, relation=AttributedName
```

General definition. Consider an object `#obj`, and `#obj.type=#TD`. Suppose that the category `#TD` is declared by the following type sheet:

```
#TD:
allOf> #N1=#C1, #N2=#C2, #N3=#C3, ...
```

where every `#Ci` is a category.

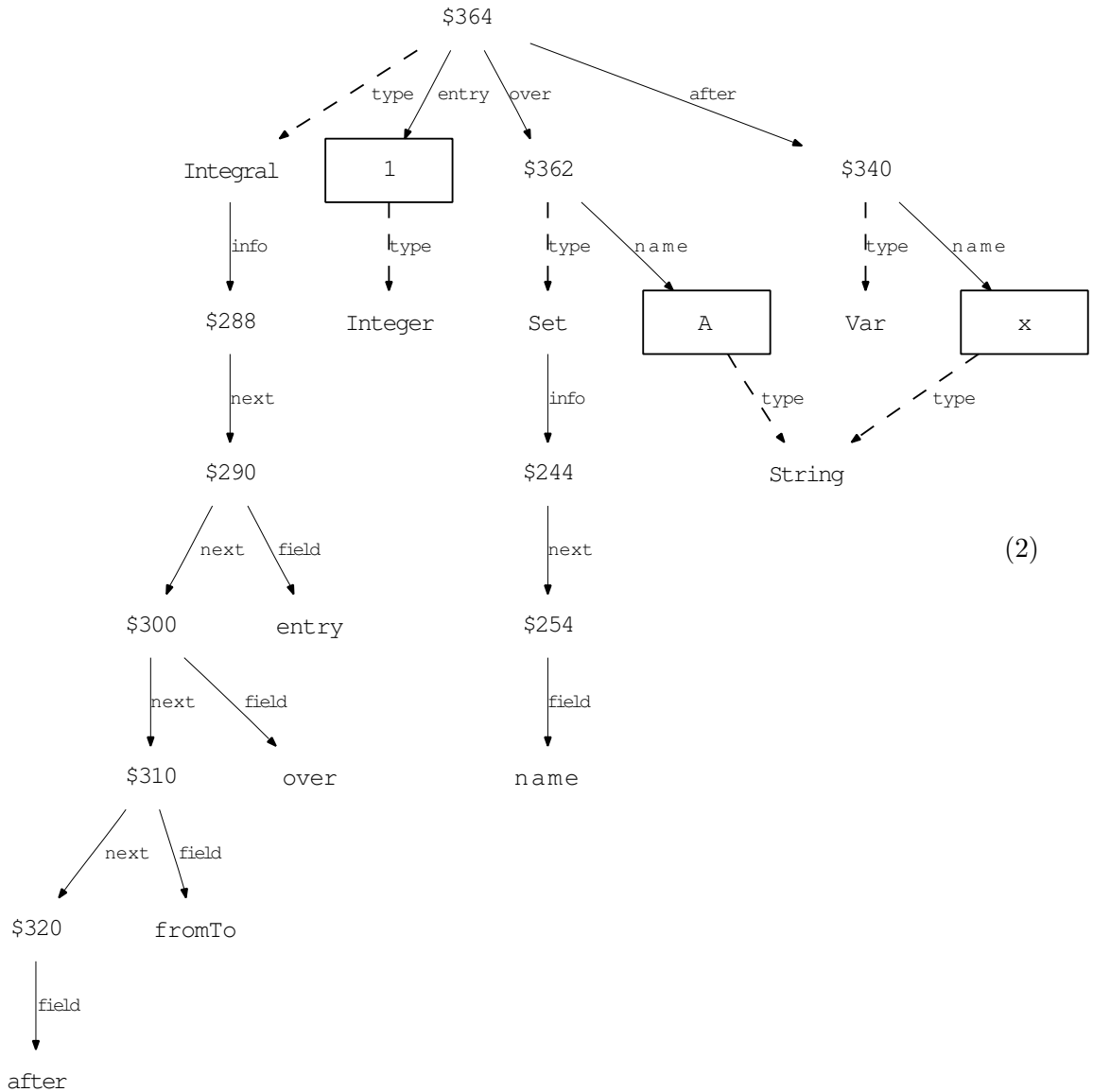
If `#Ni=type` for some i , the type sheet reader issues an error. For every i , `(#obj,#Ni)` is a declared position of `#obj`. The position `(#obj,#Ni)` is said to **have category** `#Ci` if `MATCHES(#obj.#Ni,#Ci)`, and to be **faulty** otherwise.

5.6 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

$$\int_A 1 dx$$



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
```

General definition. Consider an object `#obj` with `#obj.type=#TD`. Suppose that the category `#TD` is declared by the following type declaration:

```
#TD:
oneOf> #N1=#C1, #N2=#C2, #N3=#C3, ...
```

where every `#Ci` is a category.

If `#Ni=type` for some i , the type sheet reader issues an error. For every i , $(\#obj, \#Ni)$ is a declared position of `#obj`.

If exactly one declared position $(\#obj, \#Nj)$ is occupied and `MATCHES(#obj.#Nj,#Cj)` then the the position $(\#obj, \#Nj)$ has category `#Cj` and is faulty otherwise.

Note that each line beginning with `oneOf>` requires the object to have exactly one of the specified positions.

5.7 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 $(\#obj, \text{sub})$, $(\#obj, \text{sup})$, $(\#obj, \text{lsub})$ and $(\#obj, \text{lsup})$ to be occupied by an expression. We express these requirements in the type declaration:

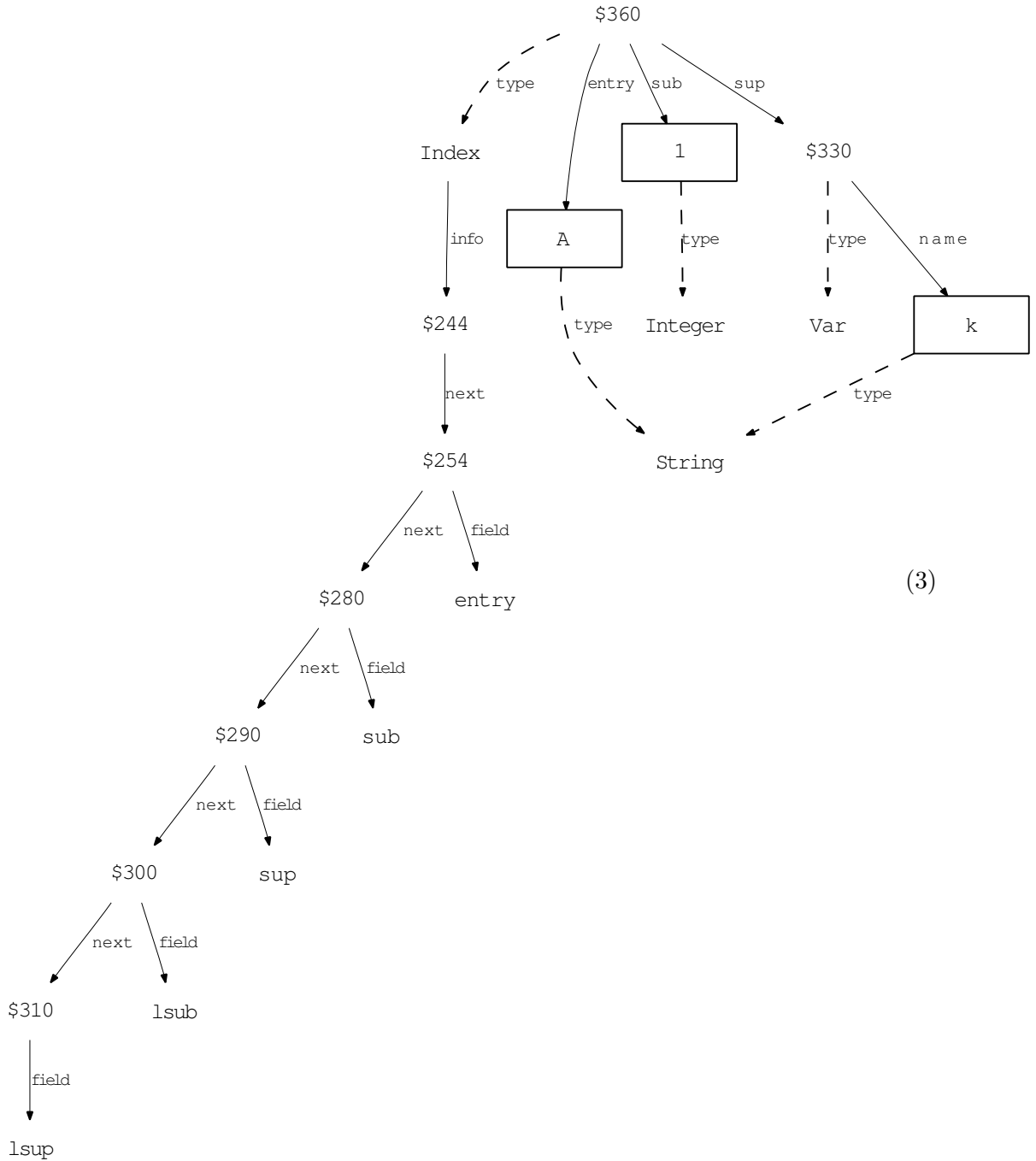
```
Index:
someOf> sub=Expression, sup=Expression, lsub=Expression, lsup=Expression
allOf> entry=Expression
```

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:



(3)

General definition. Consider an object $\#obj$ with $\#obj.type = \#TD$. Suppose that the category $\#TD$ is declared by the following type declaration:

$\#TD$:
 $someOf \langle \#N1 = \#C1, \#N2 = \#C2, \#N3 = \#C3, \dots \rangle$

where every $\#Ci$ is a category.

If $\#Ni = type$ for some i , the type sheet reader issues an error. For every i , $(\#obj, \#Ni)$ is a declared position of $\#obj$.

If at least one declared position $(\#obj, \#Nj)$ is occupied and for all occupied declared positions $MATCHES(\#obj.\#Nk, \#Ck)$ then the position $(\#obj.\#Nk)$ has category $\#Ck$, and is faulty otherwise.

Note that each line beginning with `someOf>` requires the object to have at least one of the specified positions.

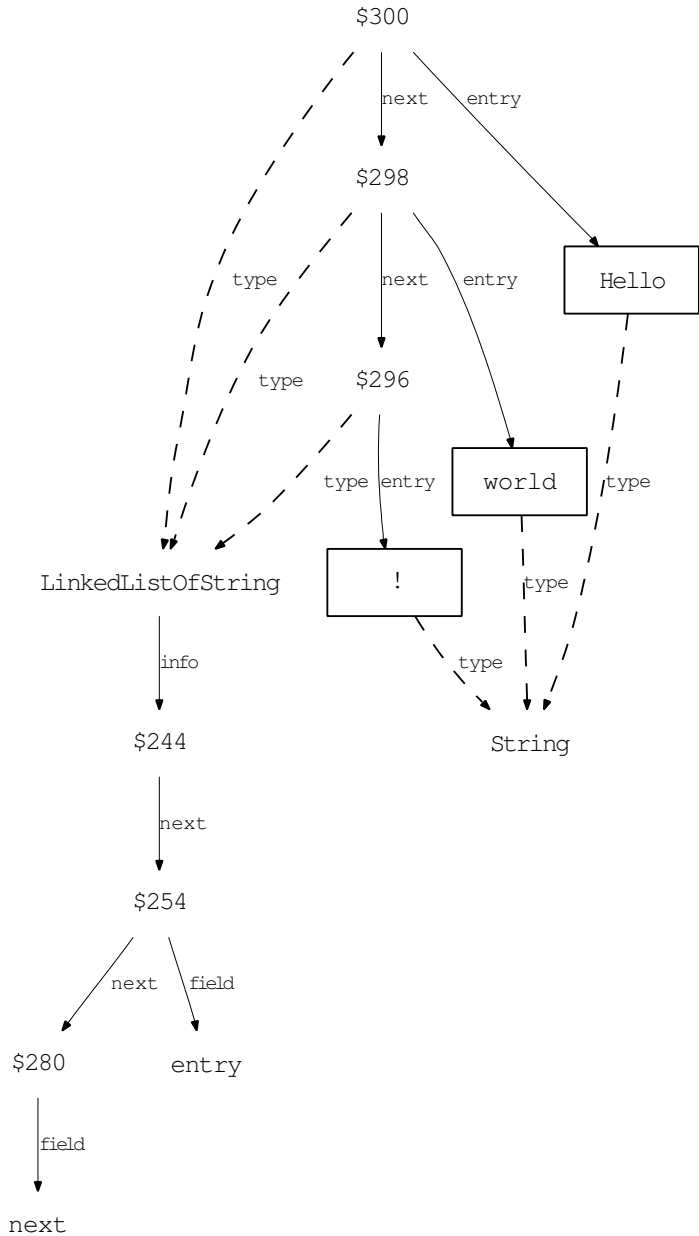
5.8 optional

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:



(4)

General definition. Consider an object $\#obj$ with $\#obj.type=\#TD$. Suppose that the category $\#TD$ is declared by the following type declaration:

```
#TD:
optional> #N1=#C1, #N2=#C2, #N3=#C3, ...
```

where every $\#C_i$ is a category.

If $\#N_i=type$ for some i , the type sheet reader issues an error. For every i , $(\#obj, \#N_i)$ is a declared position of $\#obj$.

If for all *occupied* declared positions $MATCHES(\#obj.\#N_k, \#C_k)$ then the position $(\#obj.\#N_k)$ has category $\#C_k$, and is faulty otherwise.

5.9 fixed

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

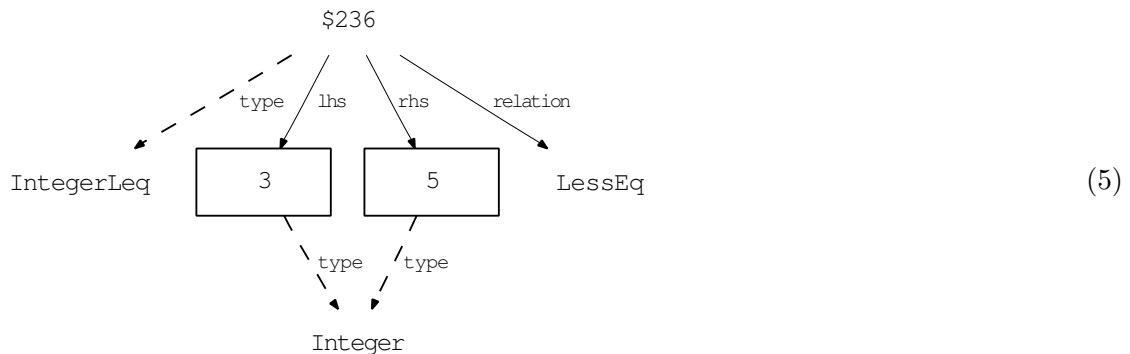
Example. We define a special binary relation `IntegerLessEq`:

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

Then the relation

$$3 \leq 5$$

would be represented by:



General definition. Consider an object `#obj` with `#obj.type=#TD`. Suppose that the category `#TD` is declared by the following type declaration:

```
#TD:
fixed> #01=#o1, #02=#o2, #03=#o3, ...
```

where every `#0i` and every `#oi` is an object.

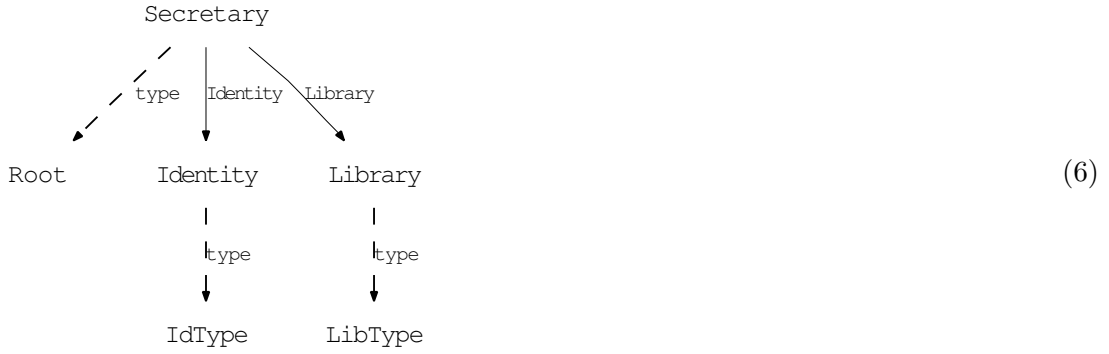
If `#0i=type` for some `i`, the type sheet reader issues an error. Every position `(#obj,#0i)` for some `i` is a declared position of `#obj`.

If for all declared positions `(#obj,#0i)` if `#obj.#0i=#oi` then the position `(#obj,#0i)` has category `#oi`, and is faulty otherwise.

5.10 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.



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
```

General definition. Consider an object `#obj` with `#obj.type=#TD`. Suppose that the category `#TD` is declared by the following type declaration:

```
#TD:
only> #01=#C1, #02=#C2, #02=#C3, ...
```

where every `#0i` is an object and every `#Ci` is a category.

If `#0i=type` for some `i`, the type sheet reader issues an error. Every position `(#obj,#f)` with `#f=#0i` for some `i` is a declared position of `#obj`.

If for all declared sems `#obj.#0i=#0i` and `MATCHES(#0i,#Ci)` then the position `(#obj.#0i)` has category `#Ci`, and is faulty otherwise.

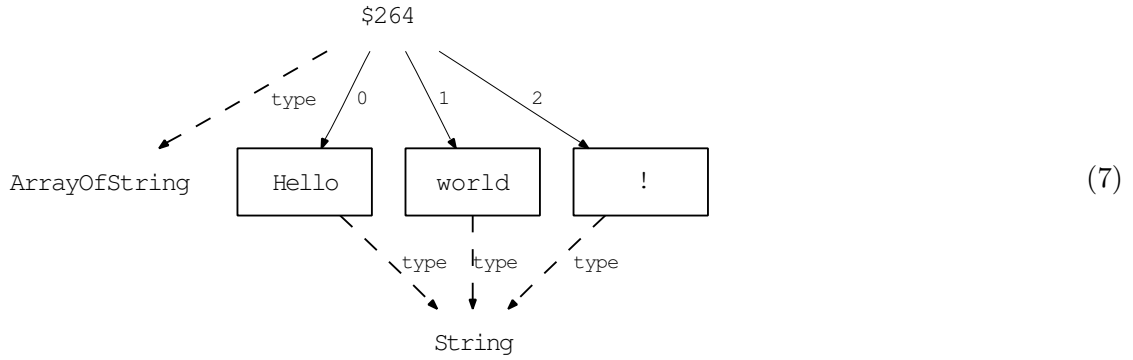
5.11 someOfType

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 accessible 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 “!”.



General definition. Consider an object $\#obj$ with $\#obj.type=\#TD$. Suppose that the category $\#TD$ is declared by the following type declaration:

```
#TD:
someOfType> #C1=#c1, #C2=#c2, #C3=#c3, ...
```

where every $\#C_i$ and every $\#c_i$ is a category.

Every position $(\#obj,\#f)$ with $MATCHES(\#f,\#C_i)$ for some i and $\#f \neq type$ is a declared position of $\#obj$.

If at least one declared position is occupied, for all declared sems $\#obj.\#f = \#e$, $MATCHES(\#f,\#C_i)$ implies $MATCHES(\#e,\#c_i)$ then the position $(\#obj.\#f)$ has category $\#c_i$ and is faulty otherwise.

Note that each line beginning with `someOfType>` requires the object to have at least one of the specified positions.

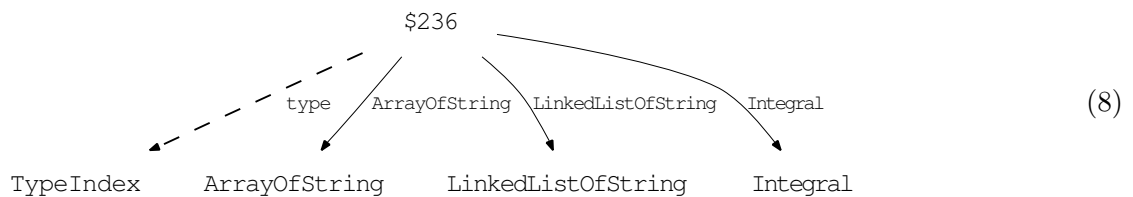
5.12 itself

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`.



General definition. Consider an object `#obj` with `#obj.type=#TD`. Suppose that the category `#TD` is declared by the following type declaration:

```
#TD:
itself> #C1, #C2, #C3, ...
```

where every `#Ci` is a category.

Every position `(#obj,#f)` with `MATCHES(#f,#Ci)` for some i and `#f≠type` is a declared position of `#obj`.

If for all declared sems `#obj.#f = #e`, `MATCHES(#f,#Ci)` implies `#f = #e` then the position `(#obj.#f)` has category `#Ci`, and is faulty otherwise.

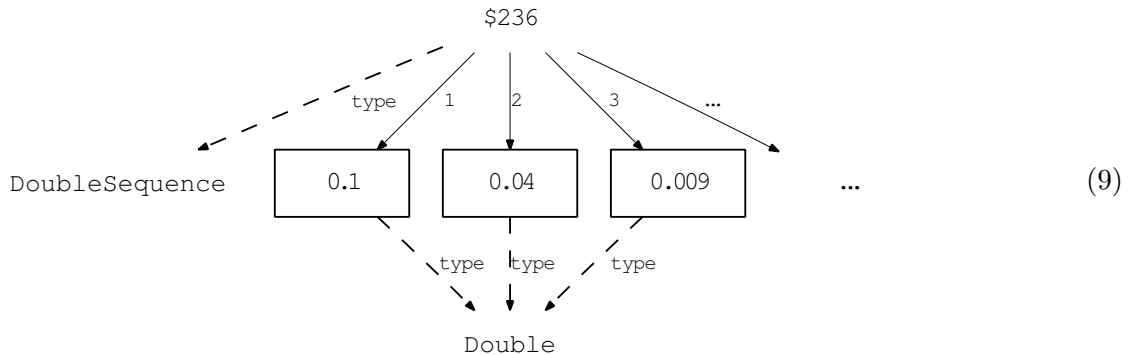
5.13 array

Scopes are objects describing a well-ordered set of 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:



General definition. Consider an object `#obj` with `#obj.type=#TD`. Suppose that the category `#TD` is declared by the following type declaration:

```
#TD:
array> #S1=#c1, #S2=#c2, #S3=#c3, ...
```

where every `#Si` is a scope and every `#ci` is a category.

Every position `(#obj,#f)` with `#f` in the scope `#Si` for some i and `#f≠type` is a declared position of `#obj`.

If all declared positions are occupied, for all declared sems `#obj.#f = #e`, `#f` in scope `#Si` implies `MATCHES(#e,#ci)` then the position `(#obj.#f)` has category `#ci`, and is faulty otherwise.

5.14 index

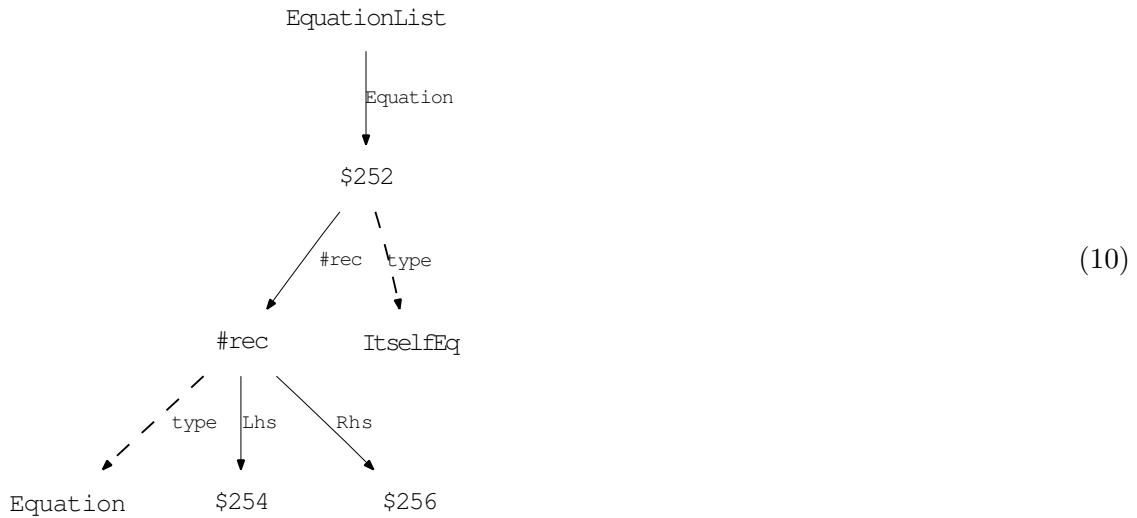
For some categories, we want to keep track of all their instances. Via `index`, we require each instance of some category to be listed in some assigned record.

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 records 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`, which should be listed in the `EquationList`:



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 `EquationList`. We do this by the type declaration:

```
Equation:
allof> Lhs=Object, Rhs=Object
index> EquationList = ItselfEq
```

General definition. Consider an object `#obj` with `#obj.type=#C`. Suppose that the category `#C` is declared by the following type declaration:

```
#C:
index> #I1=#S1, #I2=#S2, ...
```

where every `#Ii` is an object and every `#Si` is a category.

The sems `#Ii.#C.type=#Si` are created by the type sheet reader for all `i`.

If `#Ii.#C.#obj≠#obj` for some `i` then `#obj` is ill-typed.

If the declaration of the categories `#Si` for all `i` does not accord to the type declaration

```
#Si:
itself> #C
```

then `#obj` is ill-typed.

5.15 template

For examples, see Section 4.

General definition. Consider an object `#obj` with `#obj.type=#TD`. Suppose that the category `#TD` is declared by the following type declaration:

```
#TD:  
template> #C  
.br/>.br/.
```

where `#C` is a category.

For the assignment of categories to sems and the decision whether a position is faulty or not, the type declaration is equivalent to inserting the body of the type declaration for `#C` in place of the line `template> #C`.

5.16 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>
```



General definition. Consider an object `#obj` with `#obj.type=#TD`. The type `#TD` is declared by a type declaration that contains the line `nothingElse`:

```
#TD:  
.br/>.br/.  
nothingElse>
```

If `#obj` has a constituent `#obj.#f=#e` where `#f` is not `type` that is not a declared sem then position `(#obj,#f)` is faulty.

6 Well-typed records

If every declared sem reachable from the record `#rec` has a unique type, and no object reachable from `#rec` is ill-typed, then `#rec` is **well-typed**. If `#rec.type = Empty` then `#rec` is well-typed.

Otherwise `#rec` 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 emphasize that every declared sem has to have a *unique* type, because from the definition of the type of a sem, a record `#rec` of type `#DT` with type declaration

```
#DT:
someOf> entry=Integer
optional> entry=Object
```

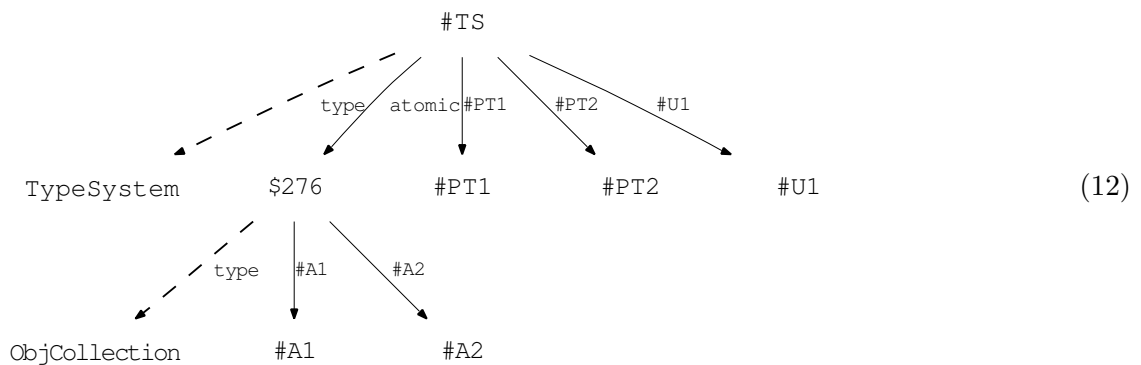
could have a constituent `#rec.entry=4` which would be both of type `Integer` and `Object`.

7 Representation of type declarations in the SM

A type system is an object `#TS` in the semantic memory with `#TS.type=TypeSystem`. The unions `#U1`, `#U2`, etc. and proper types `#PT1`, `#PT2`, etc. belonging to the type system are stored in `#TS.#U1=#U1` etc.

The atomic types `#A1`, `#A2` etc. are stored in `#TS.atomic.#A1=#A1` etc.

Note that `atomic` is therefore a reserved name.



7.1 nothing

Consider a proper type `#DT` using `nothing`:

```
#PT:
nothing>
```

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



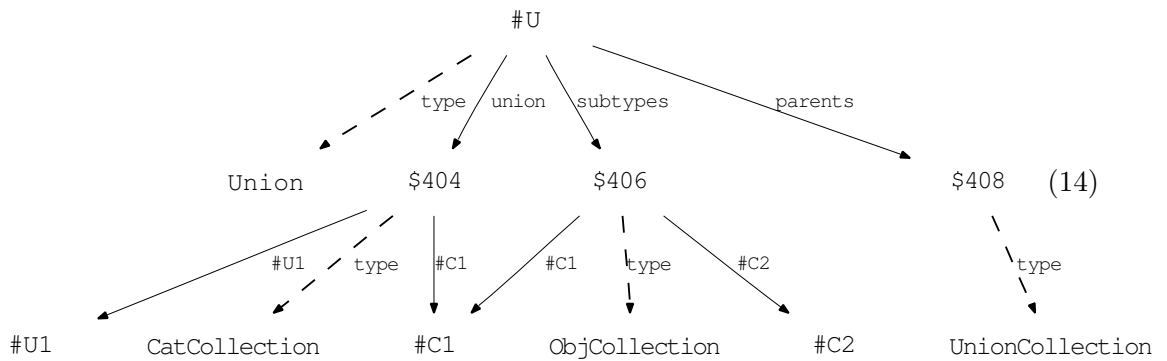
7.2 union

Consider a union **#U**:

#U:

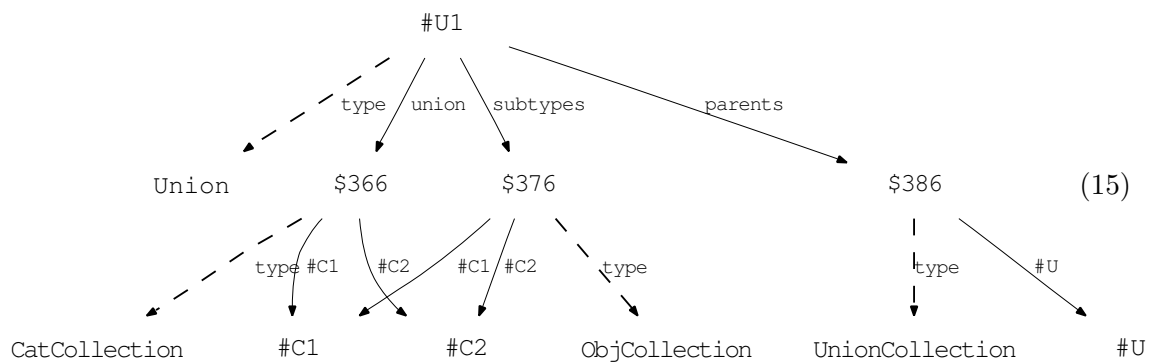
union> **#C1, #U1 ! etc.**

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



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

Furthermore, the union **#U** is stored as a parent of the union **#U1**:



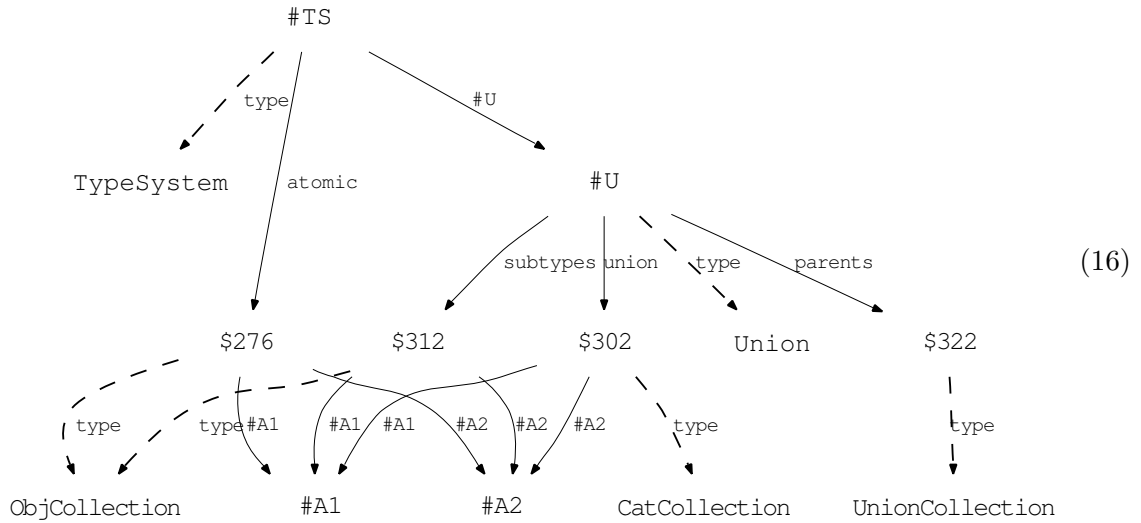
7.3 atomic

Consider a union #U of (new) atomics:

#U:

atomic> #A1, #A2 ! etc.

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



7.4 complete

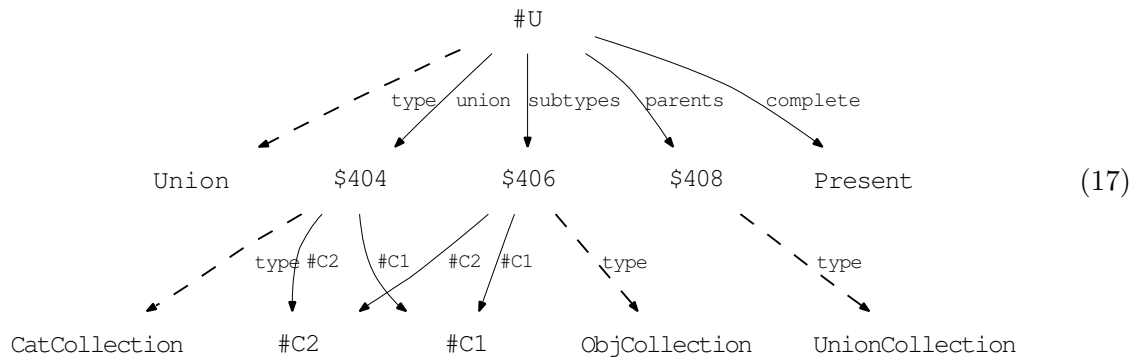
Consider a union #U using complete:

#U:

union> #C1, #C2 ! etc.

complete>

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

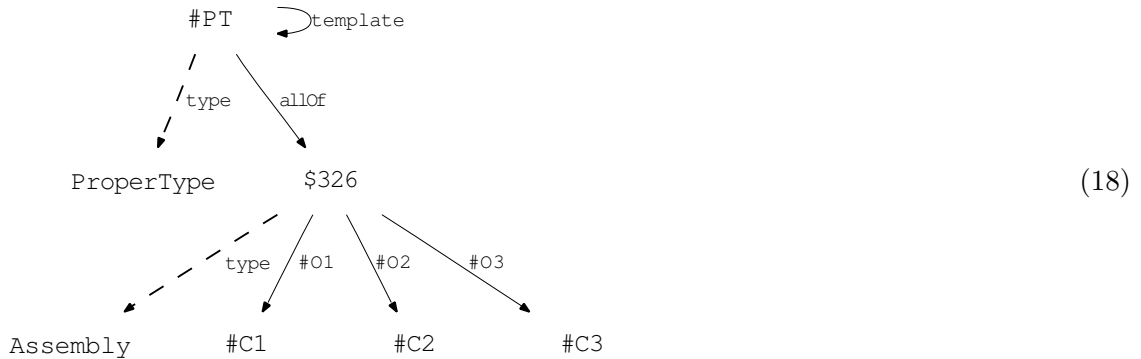


7.5 allOf

Consider a proper type #DT using allOf:

#PT:
allOf > #01=#C1, #02=#C2, #03=#C3 ! etc.

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

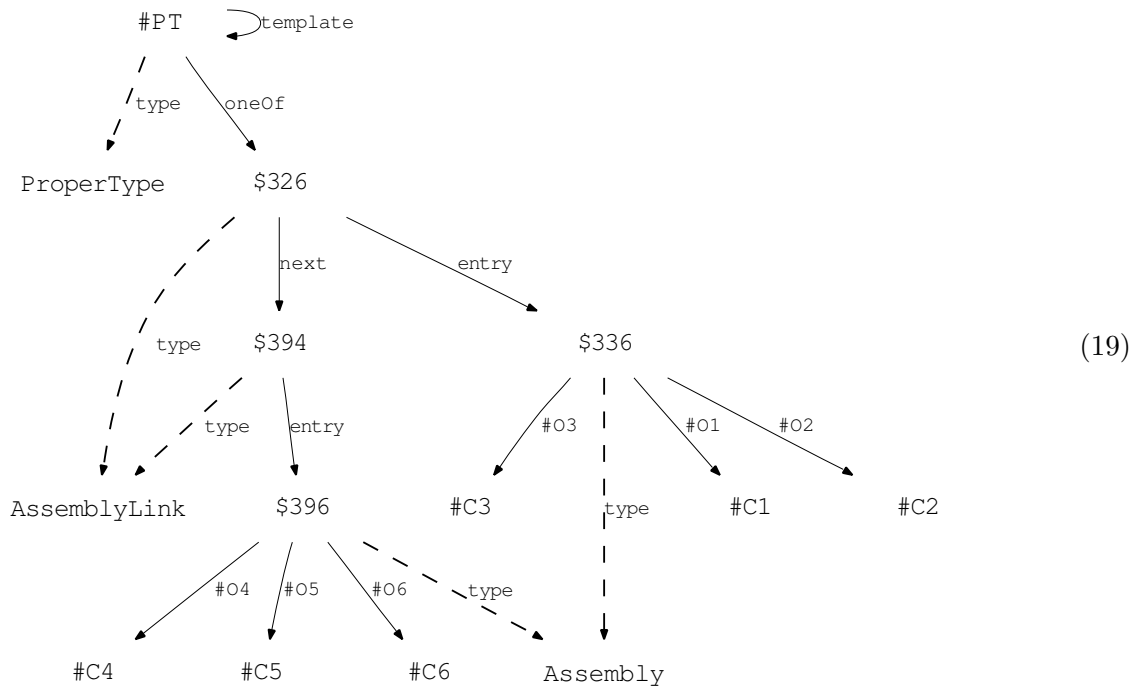


7.6 oneOf

Consider a proper type `#DT` using `oneOf`:

#PT:
oneOf > #01=#C1, #02=#C2, #03=#C3 ! etc.
oneOf > #04=#C4, #05=#C5, #06=#C6 ! etc.

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

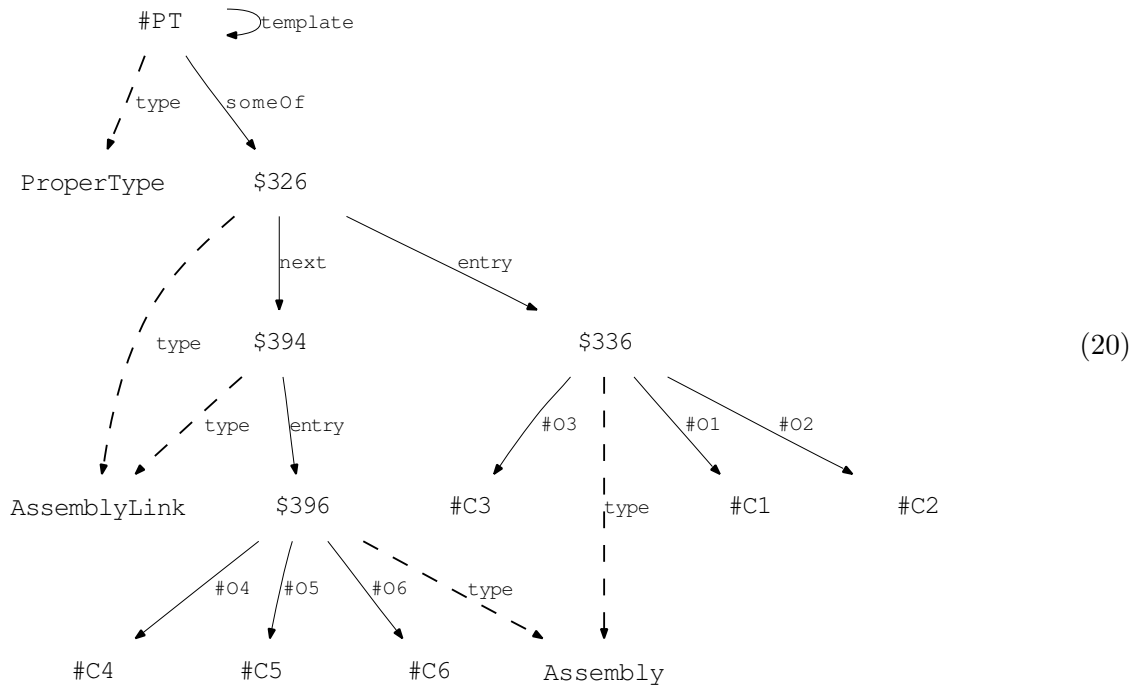


7.7 someOf

Consider a proper type `#DT` using `someOf`:

#PT:
someOf > #01=#C1, #02=#C2, #03=#C3 ! etc.
someOf > #04=#C4, #05=#C5, #06=#C6 ! etc.

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

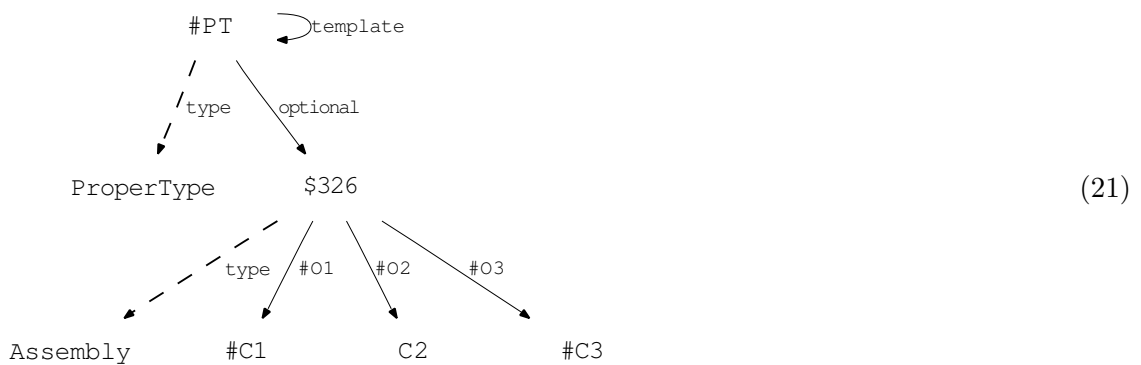


7.8 optional

Consider a proper type #DT using optional:

#PT:
optional > #01=#C1, #02=C2, #03=#C3 ! etc.

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

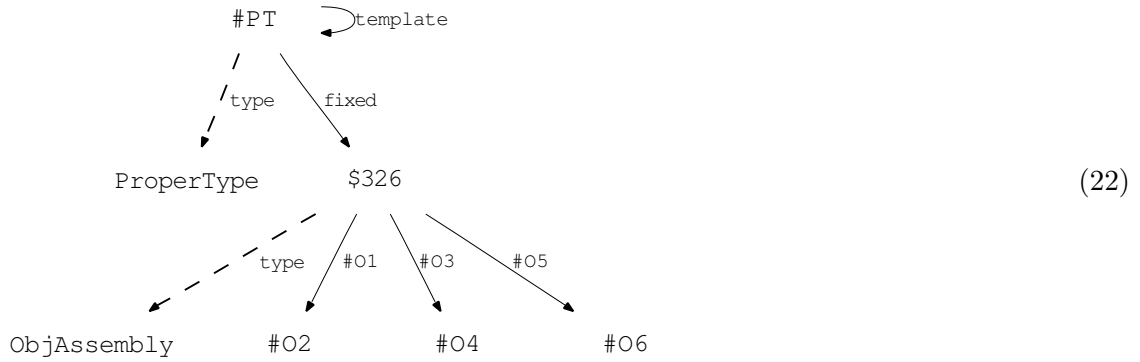


7.9 fixed

Consider a proper type #DT using fixed:

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

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



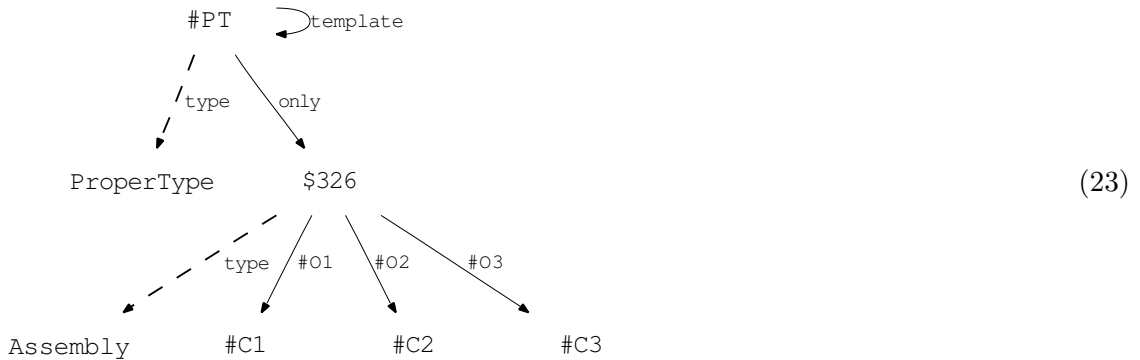
7.10 only

Consider a proper type #DT using only:

#PT:

only> #01=#C1, #02=#C2, #03=#C3 ! etc.

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



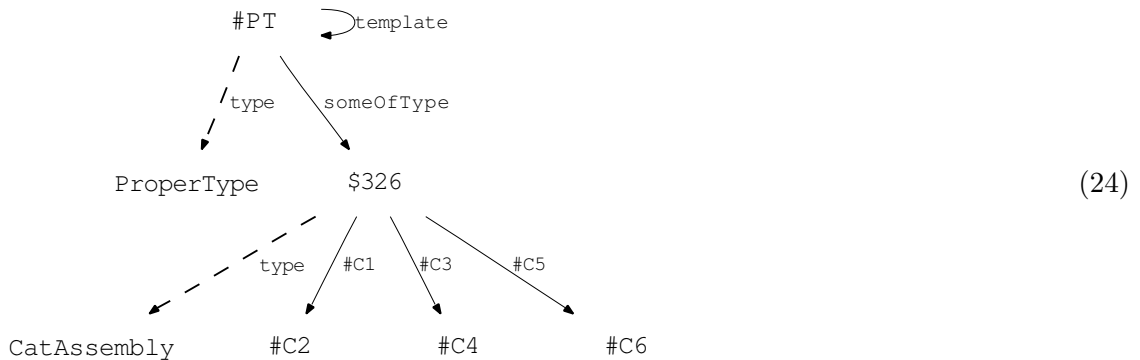
7.11 someOfType

Consider a proper type #DT using someOfType:

#PT:

someOfType > #C1=#C2, #C3=#C4, #C5=#C6 ! etc.

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

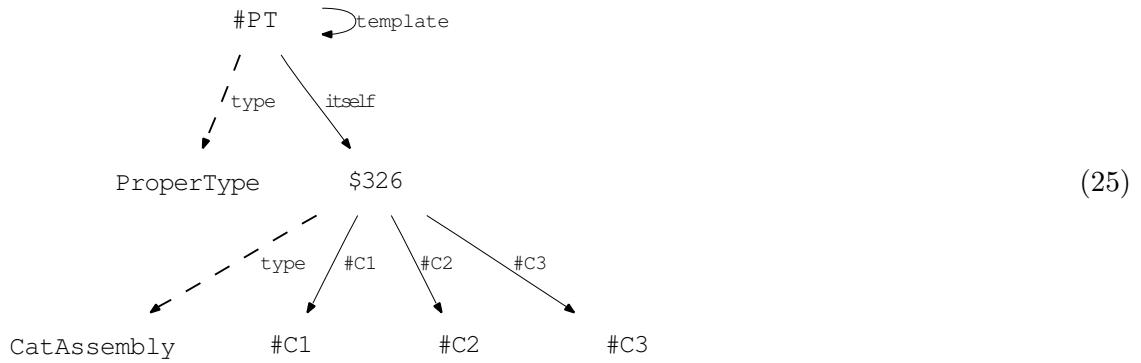


7.12 itself

Consider a proper type #DT using `itself`:

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

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

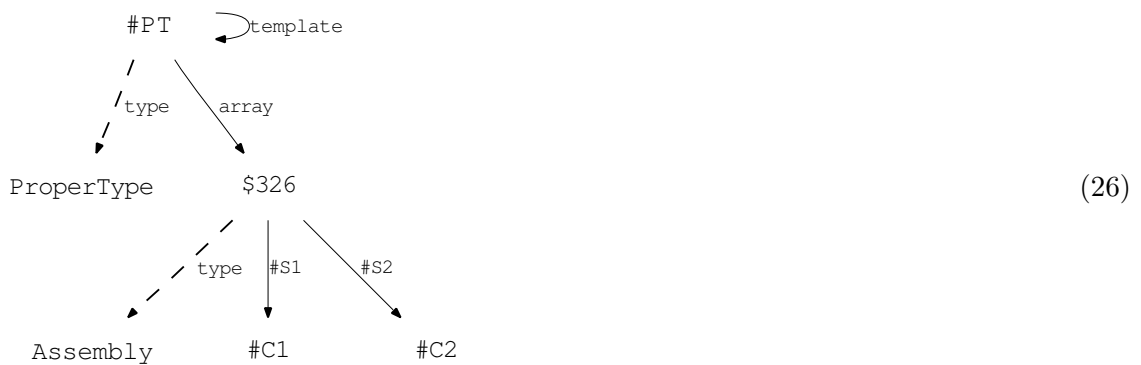


7.13 array

Consider a proper type #DT using `array`:

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

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

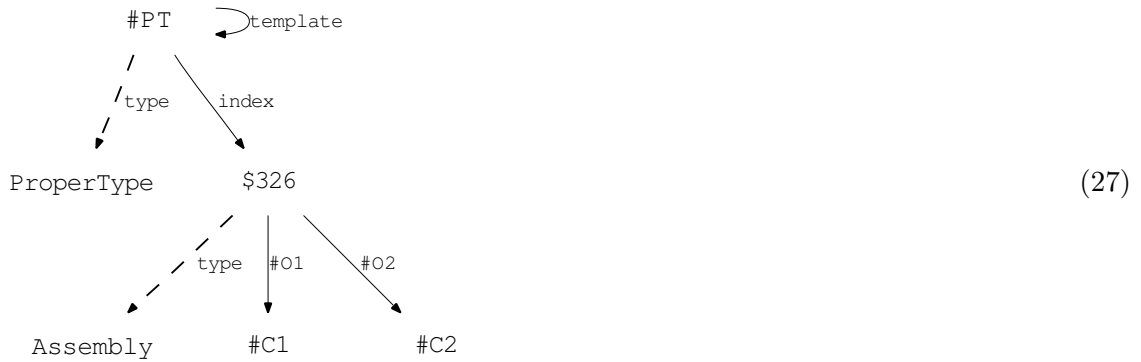


7.14 index

Consider a proper type #DT using `index`:

```
#PT:  
index> #01=#C1, #02=#C2 ! etc.
```

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



7.15 template

Consider a proper type #DT using `template`:

```
#PT:
template> #T1
```

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



7.16 nothingElse

Consider a proper type #DT using `nothingElse`:

```
#PT:
nothingElse>
```

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



8 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.

BasicTypes(TypeSystem)::

```
! Type systems
! -----
!
! Arnold Neumaier, Peter Schodl and Ferenc Domes
!
! February 9, 2011
!
! This type system defines the concept of a TypeSystem in a way that it
! comprises both context-free grammars for abstract syntax trees and
! the types needed for efficiently organizing the semantic memory.
!
! Details are given in the types paper (types.pdf).
! The basic types defined there are part of this typesheet.
! The present information will become an appendix of the latter.
!
! Note that a type sheet is just a special version of a record sheet
! in which lots of redundant information is deleted for the sake of
! optimal readability. In particular, each object declared by a line
! starting with '<name>:' is automatically assumed to be a category,
! namely a union if it has a 'union' or 'atomic' field, and a type
! otherwise.
!
```

Present:

```
nothing>
```

ProperType:

```
index> Index = ProperTypesCollection
allOf> template=ProperType
someOf> allOf=Assembly, someOf=AssemblyLink, optional=Assembly,
        oneOf=AssemblyLink, someOfType=CatAssembly, index=Assembly,
        itself=CatCollection, nothingElse=Present, fixed=ObjAssembly,
        array=Assembly, only=Assembly
optional> index=Assembly
! array can be tightened when specified
```

Union:

```
index> Index = Unions
allOf> subtypes=ObjCollection, union=CatCollection, parents=UnionCollection
optional> complete=Present
```

TypeSystem:

```
index> Index = TypeSystemCollection
itself> Category
allOf> atomic = ObjCollection
```

Category:

```
union> Union, ProperType
```

Assembly:

```
someOfType> Object=Category
```

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

```
CatAssembly:  
  someOfType> Category=Category
```

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

```
TypeSystemCollection:  
  itself> TypeSystem
```

```
ProperTypesCollection:  
  itself> ProperTypes
```

```
ObjCollection:  
  itself> Object
```

```
CatCollection:  
  itself> Category
```

```
UnionCollection:  
  itself> Union
```

```
! Note that the same production may belong to several categories.  
! Also the same category may belong to several grammars (typically  
! when these are produced by inheritance using + from simpler grammars).  
! but this is possible only if the categories agree in all their  
! productions.  
! The type checker does not check that the names of all categories  
! are distinct; this must be enforced semantically rather than through  
! types.
```


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>.