KRF

Knowledge Representation Framework Project

Department of Computer and Information Science, Linköping University,
and Unit for Scientific Information and Learning, KTH, Stockholm

Erik Sandewall

The KRF Type System and Ontology

This series contains technical reports and tutorial texts from the project on
the Knowledge Representation Framework (KRF).

The present report, PM-krf-004, can persistently be accessed as follows:

Project Memo URL:	http://www.ida.liu.se/ext/caisor/pm-archive/krf/004/
AIP (Article Index Page):	http://aip.name/se/Sandewall.Erik.-/2008/001/
Date of manuscript:	2008-10-06

Related information can also be obtained through the following www sites:

KRFwebsite:	http://www.ida.liu.se/ext/krf/
AIP naming scheme:	http://aip.name/info/
The author:	http://www.ida.liu.se/∼erisa/

Background and Introduction

The Knowledge Representation Framework (KRF) ([1]) is a set of notations and conventions for knowledge representation that has three major, interrelated usages:

- As the reference notation for an open-source textbook, *Introduction to Knowledge Representation*, by the present author.
- As the primary representation for a relatively large, open-source knowledgebase, the *Common Knowledge Library* (CKL), ([2]).
- As the main structuring framework for an experimental software system, *Leonordo* ([3]).

These usages are interrelated in several ways. In particular, the software tool for maintenance of CKL is based on Leonordo, and so is the document preparation tools that were used for preparing the textbook, as well as for preparing the present report. Also, many of the illustrative examples in the textbook are taken from the CKL knowledgebase, which has made it possible to have examples of nontrivial size.

The major aspects of the Knowledge Representation Framework are the notation and the ontology. The notation is called LDX, for the 'Leonordo Data Expression' language ([4]). For the reader who is not familiar with LDX, for reading the present memo it is sufficient to know that expressions can be formed as terms, sequences, sets, and maplets; a mapping is a set of maplets, and maplets are written as e.g. `[argument value]`. The rest will become evident as we go along.

Like any other software system, Leonordo needs and uses a type system. Likewise, like any other knowledgebase, the Common Knowledge Library needs and uses an ontology. Because of the close integration between software system and knowledgebase, the type system and the ontology are also one integrated design which we shall refer to simply as the ontology for the sake of conciseness.

The present report describes the ontology from both the software system and the knowledgebase point of view. It subsumes the sections about the type system in the KR textbook, but this report also describes the ontology that is actually used in the CKL, and how the type system and some parts of the ontology are used in Leonordo; those aspects are not covered in the textbook.

The ontology is subject to gradual development in a demand-driven fashion and according to the needs that emerge in the evolution of the Leonordo system and the CKL knowledgebase. The present report describes its state as of April, 2008. An earlier report ([5]) described the state of the ontology as of January, 2007.

---

[1] `http://piex.publ.kth.se/krf/`

[2] `http://piex.publ.kth.se/ckl/`

[3] `http://www.ida.liu.se/ext/leonordo/`

[4] To be precise, the publication notation is called KRE, for *Knowledge Representation Expressions*; the LDX notation is an extension of it for use in software systems.

[5] `http://www.ida.liu.se/ext/caisor/pm-archive/leonordo/003/`: "Recent Work and Current State of Applications and Ontology in Leonordo"

Ontology Structure and Major Types

The ontology is expressed in LDX and is organized as a collection of *entities* and of values that are assigned to *attributes* for these entities. Both entities and attribute-values can be either single identifiers or composite expressions. For example, a value can be a set or a sequence of entities, even recursively, or a formula in first-order predicate logic. Entities, attributes and values will be written in `fixed-width` font.

Entities can represent things or various kinds of abstractions, and may be thought of as the representations of concepts ([6]), although some of the entities that occur in the ontology or in applications are there for technical reasons and do not correspond to any concept that is natural for people to use. Each entity has a `type` attribute whose value is another entity; there is no exception to this rule. Entities can also have an optional `subsumed-by` attribute whose value, if present, is also another attribute.

Thingtype

For example, if `truck-12` is one particular truck then it may have a KRF description containing, in part,

```
[type truck]
[has-color black]
[made-by volvo]
```

The description of the entity `truck` may contain

```
[type Thingtype]
[subsumed-by vehicle]
```

There is just a few entities that can appear as alternatives of `Thingtype`: we have `Descriptortype`, `Spacetimetype`, and a few more. The value of the `subsumed-by` attribute is another entity that often has the same type as the given one, so in the example the description of `vehicle` might be

```
[type Thingtype]
[subsumed-by industrial-product]
```

In simple cases, at least, successive `subsumed-by` links define a trail of entities that have the same value for their `type` attribute, and where the trail ends in an entity not having any value for the `subsumed-by` attribute. In the present case this will be the entity `thing` which is then the most general subsumer having the type `Thingtype`.

The hierarchy under `Thingtype` is used for most phenomena in the world; the hierarchy under `Descriptortype` is used for many of the entities that are used in the representation system. For example, `subsumed-by` has the type `Attribute` which in turn has the type `Descriptortype`.

Although trails formed by successive `subsumed-by` links can be fairly long, this does not hold for trails formed by `type` links. Above `Thingtype`, `Descriptortype`, and the like there are only two "high-level" types, called `Type` and `Supertype`. These are related as follows. The type of `Thingtype`, `Descriptortype`, etc. is `Supertype`, and the type of `Supertype` is itself. In general, every `type` trail ends in `Supertype` after a small number of steps. However, just like `thing` is the most general subsumer for entities whose

[6] We prefer to call them entities, and not concepts, since they are just representations of concepts.

type is `Thingtype`, we need a most general subsumer for entities whose type is `Supertype`, and this is what `Type` is. Therefore, the description of `Thingtype` is

```
[type Supertype]
[subsumed-by Type]
```

The same description applies for `Descriptortype`, `Spacetimetype`, etc., and in fact also for `Supertype` itself. Finally, the description of `Type` is simply

```
[type Supertype]
```

and nothing more. It does not have any `subsumed-by` attribute since it is a most general subsumer.

Qualitytype

Entities for concepts such as "hot", "green", or "angry" are not considered as as things, but as *qualities*, and therefore besides `Thingtype` there is `Qualitytype`. For example, the description of `color` may be

```
[type Qualitytype]
[subsumed-by optical-quality]
```

with a `subsumed-by` trail from `color` via `optical-quality` and additional entities, ending up in `quality` which is the most general subsumer for `Qualitytype`. Then the description of `green` may be

```
[type color]
```

Consider furthermore the description of `light-green` which may contain

```
[type color]
[subsumed-by green]
```

indicating that light green is a variety of green.

This shows a significant difference between the entities in the `Thingtype` and the `Qualitytype` hierarchies. Define the *ontological level* of an entity as the number of `type` links that are needed to get to `Supertype`, so that `Qualitytype` and `Thingtype` have level 1, `vehicle` and `truck` have level 2, and so forth. We notice then that subsumption is relevant for both level 2 and level 3 entities in the `Qualitytype` structure, but only for level 2 entities in the `Thingtype` structure, at least in these examples.

Descriptortype and Attribute

Entities such as `type` and `subsumed-by` have the type `Attribute`, which has the following description:

```
[type Descriptortype]
[subsumed-by Descriptor]
```

where `Descriptortype` has the following description which is quite analogous to what we have seen above:

```
[type Supertype]
[subsumed-by Type]
```

We shall return later to other contents of the substructure for `Descriptortype` in the context of type declarations for attribute values.

Signatures

An *entitystate* is a set of entities and their descriptions. A *signature* is an entitystate that is used to characterize the structure of other entitystates, called its *object entitystates*. The signature must always specify what types are used in the object entitystate and what attributes may be used by instances of a type. It may also provide other information, for example, what is the admitted structure for the values for a particular attribute.

A Very Simple Signature

Since signatures are entitystates, it makes sense to ask that a signature shall also have *its* signature. In order to avoid an infinite regress, it is desirable to have first of all a signature that can be used as a signature for itself. Consider first the following very simple example of a signature. We specify a number of entities and the attribute-value assignments for each of them.

```
Type             [type Supertype]
                 [has-attributes {}]
Supertype        [type Supertype]
                 [subsumed-by Type]
                 [has-attributes {subsumed-by has-attributes}]
Descriptortype   [type Supertype]
                 [subsumed-by Type]
                 [has-attributes {subsumed-by}]
Descriptor       [type Descriptortype]
Attribute        [type Descriptortype]
                 [subsumed-by Descriptor]
type             [type Attribute]
subsumed-by      [type Attribute]
has-attributes   [type Attribute]
```

This signature has the following property. For each entity e in the signature, one identifies the value t of its `type` attribute. Then, for every assignment to an attribute a of e, except when t = `type`, a shall be a member of the `has-attributes` attribute of t. Having this property is one of the requirements for being self-describing. As a matter of convention we omit `type` in the value for `has-attributes` since every entity must have a value for `type`.

The problem with this very simple signature is of course that it does not even begin to specify the permitted structures for attribute values. We shall add this soon below, but this extension requires introducing quite a number of auxiliary entities. This is because attribute values may be sets of entities, as we observe for the `has-attributes` attribute, and therefore we must introduce a way of characterizing such sets, and this again leads to a need for describing those set-expression characterizers. For this reason, we shall make another and much simpler extension before we turn to the attribute-value specifications.

Categories

In many applications there is a need to put a "flag" on some of the entities, for example for marking entities whose description needs further checking before it can be released. Rather than having to introduce one more attribute for each such flag, it is convenient to have a single attribute whose value is a set of applicable "flags". These flags are called *categories* in the KRF ontology. The following additional definitions are needed.

```
Category          [type Descriptortype]
                  [subsumed-by Descriptor]
                  [has-attributes {applicable-for}]
applicable-for    [type Attribute]
has-categories    [type Attribute]
```

The idea is that each category or "flag" shall be represented as an entity of type `Category` and that the `has-categories` attribute of an entity shall have as value the set of the categories that apply to that entity. Furthermore, each entity of type `Category` shall have an attribute called `applicable-for` whose value shall be a non-empty set of types for entities for which this category may apply.

Since `Category` has a `has-attributes` attribute, it becomes necessary to amend `Descriptortype`, `Descriptor`, and `Attribute` as follows.

```
Descriptortype    [type Supertype]
                  [subsumed-by Type]
                  [has-attributes {subsumed-by has-attributes}]
Descriptor        [type Descriptortype]
                  [has-attributes {}]
Attribute         [type Descriptortype]
                  [subsumed-by Descriptor]
                  [has-attributes {}]
```

These amendments will anyway be needed in later steps, so they are not made only for serving the category concept.

In addition, if it should be desired to assign categories to `Descriptortype`, `Thingtype`, etc. then the description for `Supertype` must be amended as follows.

```
Supertype         [type Supertype]
                  [subsumed-by Type]
                  [has-attributes {subsumed-by has-categories
                         has-attributes}]
```

This addition is probably rarely useful in practice, but we include it here in order to provide additional work for the validation procedure to be defined below.

The Supersignature

We proceed now to a signature that also specifies attribute structure. This signature will be introduced in several steps. First, we repeat the same entity descriptions as above, including the amendments for categories, but introducing in addition an attribute for `Attribute` called `valuetype`. This attribute is going to be used for expressing the permitted structure for the value of the attribute in question.

```
Type            [type Supertype]
                [has-attributes {}]
Supertype       [type Supertype]
                [subsumed-by Type]
                [has-attributes {subsumed-by has-categories
                   has-attributes}]
Descriptortype  [type Supertype]
                [subsumed-by Type]
                [has-attributes {subsumed-by has-attributes}]
Descriptor      [type Descriptortype]
                [has-attributes {}]
Attribute       [type Descriptortype]
                [subsumed-by Descriptor]
                [has-attributes {valuetype}]
Category        [type Descriptortype]
                [subsumed-by Descriptor]
                [has-attributes {applicable-for}]
type            [type Attribute]
subsumed-by     [type Attribute]
has-categories  [type Attribute]
has-attributes  [type Attribute]
valuetype       [type Attribute]
applicable-for  [type Attribute]
```

Next, we extend the definitions of the entities of type `Attribute` so that they also have a value for `valuetype`. This value shall specify what is admissible structures for the value assigned to the attribute in question.

```
type            [type Attribute]
                [valuetype Type]
subsumed-by     [type Attribute]
                [valuetype Type]
has-categories  [type Attribute]
                [valuetype (setof Category)]
has-attributes  [type Attribute]
                [valuetype (setof Attribute)]
valuetype       [type Attribute]
                [valuetype Type]
applicable-for  [type Attribute]
                [valuetype (setof Type)]
```

The second requirement for self-description in a signature is as follows. Consider any assignment of a value v to the attribute a of an entity e in the signature, and let s be the value of the `valuetype` attribute of `a`. If v is an entity then `s` must either be equal to the `type` attribute of v, or it must be an element in the `subsumed-by` trail from the value of the `type` attribute of v. If v is a non-entity expression such as a set or a sequence, then it must conform to s according to the rules for every such kind of expression.

For example (writing henceforth $e.a$ for the value of the a attribute of e), above, `Attribute.subsumed-by` is `Descriptor` the type of which is `Descriptortype`; `subsumed-by.valuetype` is `Type`; this is accepted since `Type` is on the `subsumed-by` trail from `Descriptortype`. Also, `type.valuetype` is `Type` the type of which is `Supertype`; `valuetype.valuetype` is `Type`; this

is also accepted for the similar reason.

Continuing with composite attribute values, `Descriptortype.has-attributes` is {`subsumed-by has-attributes`} and `has-attributes.valuetype` is `(setof Attribute)`. This is accepted according to the rule for the `setof` expression, since {`subsumed-by has-attributes`} is a set and each of its members has the type `Attribute`.

In order for the entire structure to be self-describing we need to introduce a type for operators such as `setof` (there will be more of them in the sequel) as well as a way for defining their structure. The following definitions do this.

```
Ecomposer       [type Descriptortype]
                [subsumed-by Descriptor]
                [has-attributes {argtypes valtype}]
argtypes        [type Attribute]
                [valuetype Seq-type]
valtype         [type Attribute]
                [valuetype Type]
setof           [type Ecomposer]
                [argtypes (seq Type)]
                [valtype Set-type]
Set-type        [type Supertype]
                [subsumed-by Type]
                [has-attributes {}]
```

Much of this is self-explanatory. The value of the `argtypes` attribute shall be a sequence consisting of the types for the successive arguments of the composer in question, which is why we select its `valuetype` as `Seq-type`. The value of the `valtype` attribute shall be the type of expressions formed using the Ecomposer in question. (Notice the difference between `valuetype` and `valtype`).

This again requires us to introduce a definition of the operator `seqof` which is analogous to `setof`, as well as an operator `seq` of an arbitrary number of arguments, obtaining

```
seqof           [type Ecomposer]
                [argtypes (seq Type)]
                [valtype Seq-type]
seq             [type Ecomposer]
                [argtypes (seqof Type)]
                [valtype Seq-type]
Seq-type        [type Supertype]
                [subsumed-by Type]
                [has-attributes {}]
```

For example, `(seqof vehicle)` is the type for sequences of any number of elements where each element has the type `vehicle` or a type that is subsumed by `vehicle`. Similarly, `(seq person vehicle)` is the type for sequences of exactly two elements where the first element is a `person` and the second element is a `vehicle`. It is clear that the type for the argument-list for `seqof` is `(seq Type)`, and the type for the argument-list of `seq` is `(seqof Type)`.

With this, we obtain the following inferred descriptions for those composite entities occurring above:

```
(setof Category)     [type Set-type]
(setof Attribute)    [type Set-type]
(setof Type)         [type Set-type]
(seqof Type)         [type Seq-type]
(seq Type)           [type Seq-type]
```

This makes it possible to apply the self-description requirement even on those attribute values that are composite entities. For example, `applicable-for.valuetype` is `(setof Type)` whose type is `Set-type`, `valuetype.valuetype` is `Type`, and this is accepted since `Type` is on the `subsumed-by` trail from `Set-type`.

In order to verify the choice of `argtypes.valuetype`, consider for example the attribute assignment for `seq.argtypes` as `(seqof Type)`, the type of which is `Seq-type`. This agrees with `argtypes.valuetype` which is also `Seq-type` so the assignment is accepted.

Notice by the way the distinction that is made between *composite entities* such as `(setof Type)`, and LDX expressions that are not entities but e.g. sets or sequences, such as {`argtypes valtype`} or ⟨`Type`⟩. Entities whose type is `Ecomposer` are operators for forming composite entities, and composite entities have a type and other attributes just like atomic entities.

The actual supersignature that is used in Leonordo and CKL at present (April, 2008) contains the following additional entities in addition to those described above.

```
setofall          [type Ecomposer]
                  [argtypes (seq Type)]
                  [valtype Set-type]
setofsome         [type Ecomposer]
                  [argtypes (seq Type)]
                  [valtype Set-type]
join              [type Ecomposer]
                  [argtypes (seqof Type)]
                  [valtype Type]
```

Of these, `setofall` and `setofsome` are used like `setof` but provide additional information concerning whether the set in question shall be assumed to be the complete set or not; this has been described in the textbook. The `join` operation can be used in the `valuetype` attribute in order to form the union of several given types. It is not used in the supersignature itself, but its definition has been included there so that it is available for other signatures that are based on the supersignature.

Extensions to the Supersignature

The supersignature is self-describing in the sense that it is an adequate description of itself with respect to what attributes are used for which entity types, and what structure is admitted in attribute values. Signatures for applications may be thought of as a three-step structure consisting of the actual knowledgebase (K), the signature for the knowledgebase (S), and the signature for the signature which is the supersignature (SS). However, it is more fruitful to think of it as follows: S is a signature having the property

that the union of S and SS is self-describing, and likewise K is a signature having the property that the union of K, S and SS is self-describing.

The advantage with this way of seeing things is that it is modularity-oriented: it makes it possible to build a library of signature modules which can be assembled according to need. Global library information specifies which modules depend on which other modules, and the union of a given set of modules is self-describing provided that for each module in the set, the set contains all the modules that the given module depends on (provided that the modules have been designed correctly and do not have any conflicts).

Self-description is important because it is a way of expressing type-checking. In the simple case of K, S and SS for a knowledgebase and its signature, the union of K, S and SS can only be self-describing if the contents of K conform to the type information that is given in S and the signature S is organized according to the conventions that are represented in SS.

We proceed now to describing a few small signature modules that are often used.

The Scalar Signature

Most applications require the use of attribute values that are strings, numbers or other scalars. The signatures for such applications require the use of the appropriate scalar types, which are defined as follows.

```
Scalar-type   [type Descriptortype]
              [subsumed-by Descriptor]
              [has-attributes {}]
String        [type Descriptortype]
              [subsumed-by Scalar-type]
              [has-attributes {}]
Number        [type Descriptortype]
              [subsumed-by Scalar-type]
              [has-attributes {}]
Niltype       [type Descriptortype]
              [subsumed-by Scalar-type]
              [has-attributes {}]
Symbol        [type Descriptortype]
              [subsumed-by Descriptor]
              [has-attributes {}]
```

The absence of a value for an attribute is defined as equivalent to the attribute having the entity `nil` as value, and this entity is by definition the sole instance of the type `Niltype`.

The entity `Symbol` is included as a catch-all that can be used when an attribute has values that are implemented like entities but without any value for the `type` attribute.

The Signature for Records and Predicates

Besides sets and sequences, the LDX notation also allows the use of *records* which are written on the form e.g. `[R a b c]` consisting of a *record composer* `R` and an unspecified number of arguments. A record composer does not in itself specify the number and the types of the arguments, so when an attribute is going to have records as values then the `valuetype` of that attribute must specify any restrictions on the number or types of the arguments.

However, there is also the more specific case of *predicates* which are record composers that do specify the number and the type restrictions for their arguments. Records formed using predicates are called *literals* and are used to form logic formulas in LDX.

The following is the signature extension for records and predicates.

```
Record-type    [type Supertype]
               [subsumed-by Type]
               [has-attributes {}]
Rcomposer      [type Descriptortype]
               [subsumed-by Descriptor]
               [has-attributes {}]
record         [type Ecomposer]
               [argtypes (seq Rcomposer)]
               [valtype Record-type]
record-args    [type Ecomposer]
               [argtypes (seq Rcomposer (seqof Type))]
               [valtype Record-type]
Literal-type   [type Supertype]
               [subsumed-by Record-type]
               [has-attributes {}]
Predicate      [type Descriptortype]
               [subsumed-by Rcomposer]
               [has-attributes {argtypes}]
literal        [type Ecomposer]
               [argtypes (setof Predicate)]
               [valtype Literal-type]
```

For example, suppose `Email` is a record composer whose arguments are intended to be the `to`, `cc`, `Subject`, etc. fields in an electronic mail message. Suppose also that the value of the attribute `mail-exchange` is going to be a sequence of such records. The signature information for this is

```
mail-exchange [valuetype (seqof (record Email))]
```

If instead it is required that each record in that attribute shall only contain two arguments, one with the type `person` and the other with the type `String`, then the signature information should be

```
mail-exchange [valuetype
                 (seqof (record-args Email ⟨person String⟩))]
```

Finally, in a scenario where a university dean wishes to keep track of which of her faculty are consulting for which outside companies, the value of the `consulting` attribute for each department may be a set of literals of the form `[consults-for person company]`. This would be defined through

```
consults-for    [type Predicate]
                [argtypes (seq person company)]
consulting      [type Attribute]
                [valuetype (setof (literal {consults-for}))]
```

The argument for the operator `literal` is a set of predicates in order to allow for more than one predicate being used in a particular collection of literals.

Aggregations

Entityfiles

Knowledgeblocks

Signature Checking

Taxonomy

Appendix: Type Structure and Ontology in Leonordo

The type structure and ontology are fundamental for organizing the contents of Leonordo, including both 'programs' and 'data' to the extent that this distinction can still be made. However, there are a number of additional aspects that are important in how Leonordo uses these.

When an entityfile is written to a persistent file then each entity may have an attribute called `latest-rearchived` which is used by the archiving and version management facility, which is documented separately. Furthermore, each entity of type `entityfile` may have an attribute `latest-rearchived-entity` which is also used for the same purpose. These attributes are not documented in the values of `has-attributes`.

In order to avoid writing a lot of attributes with the value `nil`, recorded entityfiles contain a pseudo-attribute called `nullvalued` whose value is a set of attributes that shall be set to `nil` when the entityfile is loaded.

The presentations of knowledge modules in CKL contain only the values of external attributes, i.e., those listed in the `has-attributes` attribute. The signature checking routine only considers external attributes, and it therefore disregards the `attributes` attribute.

Types have both a `has-attributes` attribute and an `attributes` attribute, where the value of the former is a subset of the value of the latter. The difference is that `has-attributes` lists those attributes that are shown externally while `attributes` contains the larger set of attributes that are maintained internally, including *internal attributes* that are only used in the operation of the software system, or whose contents are preliminary and not yet ready to be published.

The following are some examples of internal attributes, besides the attribute `attributes` itself. The type `entityfile` has an attribute called `latest-written` showing a timestamp of the most recent writing of the entityfile to file.

More here?

Revisions and Current Usage

Supersignature

In view of the multiple uses of the Knowledge Representation Framework that were described at the beginning of this report, it is very important that the type system and the ontology that are used in these various places are consistent. At the same time there is by necessity a certain evolution in all parts of the design. This evolution is documented periodically in the project's design document, most recently in the January, 2007 report. The following is how the type system and ontology described here relates to these other places.

The type system that is described in the section "Signature Defining Attribute Structure" is consistent with what is described in our KR textbook, except that the definitions for categories have been added here. The present (April, 2008) version of the Leonordo system uses an entityfile called `supersignature` containing a superset of supersignature described here. This entityfile is also used by the module validation facility in the Common Knowledge Library and is posted on its website.

This signature has only minor differences from the one that was described in our January, 2007 system development notes, namely, as follows:

- The entities `Type` and `Supertype` had other names there. The structure was isomorphic however.
- The `Descriptortype` hierarchy had not been introduced there.

Ad-hoc Deviations in Supersignature

The following are ad-hoc deviations between the supersignature described above and the one actually used at present in the Leonordo system. These deviations are ad-hoc in the sense that they are only motivated by ideosyncracies in the actual use of the type system, and that they should ideally be removed.

- The entity `Type` has sometimes been thrown in as the type of a particular type instead of identifying a more precise type. This was not intended but has happened. Therefore we have

  `Type` `[has-attributes ⟨has-attributes has-categories subsumed-by⟩]`

- Compared to what has been described above, `has-categories` is not present in `Supertype.has-attributes`, but it is present in `Descriptortype.has-attributes`. This reflects actual usage.

- The `subsumed-by` attribute is missing in `Attribute`, `Category`, `Ecomposer`, compared to what was shown above.
- `applicable-for` is only an internal attribute for `Category`.
- `has-attributes.valuetype` is `(setofall Attribute)` rather than `(setof Attribute)`. This distinction is covered in the textbook and should be of minor concern here.

The following entityfiles related to the type system and the ontology are loaded with the core of Leonordo:

- `signature-symbfuns`, containing procedural definitions for creating composite entities such as `(seqof Type)`;
- `supersignature`, containing the same material as shown here, with the minor deviations listed below;
- `scalars-signature`, containing definitions for entities such as `String` and `Integer` as defined above.

In addition there are the following ontology-related entityfiles that have to be loaded separately by those computational contexts that need them.

- `records-signature`, containing definitions for records in the sense of the LDX notation;
- `coreonto`, containing older type definitions that are no longer needed in principle but which may linger in some odd entityfiles;
- `cognonto`, containing definitions of types and qualities that pertain to the real world (rather than to internal Leonordo artifacts), but which are also used in the operation of Leonordo. This includes concepts such as dates, languages, and software products, for example.