Chapter 5

Noos Formalization

The goal of this Chapter is to present a formal description of the Noos representation language. We will present the Noos formal syntax based on feature terms, its semantics, and the formal model of the Noos inference process.

Our approach to formalize Noos syntax and semantics is related to the research based on λN calculus [Dami, 1994], *ψ-terms* [Aït-Kaci and Podelski, 1993] [Carpenter, 1992] [Backofen and Smolka, 1995], and extensible records [Cardelli and Mitchell, 1994] that propose formalisms to model object-oriented programming constructs.

As we have stated in Chapter 3 the Noos representation language is based on feature terms formalism. The intuition behind a feature term is that of providing a way to construct terms embodying *partial information* and *amenable to extension*. The proposal of feature terms is that these two properties can be achieved by building terms with parameters identified by name (regardless of order or position) and with no fixed number of parameters.

More formally, while first order terms can be described by trees with an implicit ordering, feature terms can be seen as a generalization of them and can be described by labeled graphs where nodes are labeled with sorts and edges are labeled with named parameters (called *features*).

The λN calculus formalism is an extension of λ-calculus that introduces named parameters as arguments of lambda abstractions. In λN, a lambda abstraction can have multiple arguments which can be bound separately and in any order. λN provides a clear way to model extensible and recursive records, and also offers a clear mapping among functions and records. We use λN calculus to provide a syntax for Noos feature terms. Moreover, λN calculus capabilities for modeling extensible knowledge are used for modeling the refinemement mechanism of Noos. λN calculus lexical scoping is used for modeling path references and path equality.

The *ψ-term* formalism offers an alternative approach to model relational and object-oriented programming. [Aït-Kaci and Podelski, 1993] presents a semantical interpretation for *ψ-terms* that allows three equivalent representations: terms, clauses, and graphs. *ψ-term* calculus is based on the notions of unification and subsumption. We adopt a related approach to the semantical interpretation of ψ-terms in order to provide a semantical interpretation of Noos feature terms. Following the *ψ-term* formalism, feature terms are interpreted as partial descriptions. This semantical interpretation of feature terms brings an ordering relation among them. We call this ordering relation *subsumption*. The intuitive meaning of subsumption is that of *informational ordering*. We say that a feature term ψ subsumes another feature term ψ' (noted $\psi \sqsubseteq \psi'$) when all information in ψ is also contained in ψ'.

Last elements of the Noos language are the metalevel relation among feature terms, reflection, and the inference process involved in solving a specific problem task. λN calculus and ψ-terms are not the best suited formalisms for modeling the Noos inference process. We formally describe the global inference process in our system using Descriptive Dynamic Logic [Sierra et al., 1996]. Descriptive Dynamic Logic (DDL) is a propositional dynamic logic (PDL) [Harel, 1984] that provides a general framework for describing and comparing reflective knowledge systems. DDL models knowledge systems as a set of units with initial local theories written in possibly different languages. Each unit is also usually allowed to have its own intra-unit deductive system. Moreover, the whole knowledge system is equipped with an additional set of deductive rules, called *bridge rules*, to control the information flow among the different units of the knowledge system. Thus, the DDL approach is very useful to model reflective systems based on the use of several units containing local theories (or meta-theories acting upon theories) that influence and/or modify each other.

Moreover, two specific elements of Noos have to be defined formally: `preferences` and `perspectives`. We presented in Section 3.4 a declarative mechanism for decision making about sets of alternatives we call preferences. Reasoning with preferences is modeled by partially ordered sets with a set of operations for constructing new preferences and combining them.

Perspectives, explained in Section 4.3, are a mechanism to describe declarative biases for retrieval in the Noos episodic memory. Using feature terms, perspectives are formalized as second order feature terms that denote sets of terms.

The structure of this Chapter is as follows: Section 5.1 contains a brief introduction to λN calculus. Sections 5.2 to 5.9 present the syntax and semantics of feature terms. Section 5.10 describe the formal basis for reasoning with preferences in Noos. Section 5.11 presents the formal principles on which perspectives are based. Section 5.12 contains a brief introduction to DDL. Section 5.13 describes the inference in Noos using the DDL formalism. Finally, Section 5.14 summarizes our approach to formalize the Noos representation language.

5.1 Basic notions of λN calculus

The lambda calculus with Names (λN calculus) is an extended λ-calculus developed by Laurent Dami [Dami, 1994] that introduces named parameters as arguments of lambda abstractions. λN uses Bruijn indices [de Bruijn, 1972] as the basis for the introduction of named parameters. In λN, a lambda abstraction can have multiple arguments which can be bound separately and in any order. This feature brings what Dami calls the *extensibility property*: existing software fragments can be augmented with new features while remaining compatible with the original contexts in which they were used. The extensibility property of λN calculus allows to model records, extensible datatypes, and object-oriented programming constructs. Moreover, named parameters are introduced in λN without affecting the semantics of functions because names can be totally dismissed in the Bruijn calculus. Thereby, λN formalism preserves results of the standard lambda calculus such as confluence.

The λN calculus is constructed from a set $\mathcal{V}$ of variables (or labels). Using this set of variables and extending the λ-calculus syntax, the set of terms Λ_N of the λN calculus is built from the following basic syntax[1]:

$$\begin{array}{lcl} \mathrm{a} & ::= & \lambda(x_1 \ldots x_n)\mathrm{a} \mid \mathrm{a}(x \to \mathrm{b}) \mid \mathrm{a}! \mid \mathrm{v} \\ \mathrm{v} & ::= & x \mid \backslash \mathrm{v} \end{array}$$

Figure 5.1. Basic λN syntax.

In λN notation, a lambda abstraction $\lambda(x_1 \ldots x_n)\mathrm{a}$ can have several arguments—identified by a named parameters and declared as a list of variables $x_1 \ldots x_n$ inside two parentheses—which can be bound separately and in any order. As a consequence of this approach, the functional application operation of λ-calculus has to be split in two different operations, called *bind* and *close* operations. An expression of the form $a(x \to b)$ (called *bind expression*) binds b to the parameter x in the abstraction a. An expression of the form $a!$ (called *close expression*) ends a sequence of bind expressions.

In the context of λN, the usual β-reduction rule of lambda calculus is split in two *lambda-bind* and *lambda-close* rules:

$$(\lambda(x_1 \ldots x_n)\mathrm{a})(x_i \to \mathrm{b}) \quad \longrightarrow_\beta \quad \lambda(x_1 \ldots x_n)\mathrm{a}[x_i := \mathrm{b}] \tag{5.1}$$

$$(\lambda(x_1 \ldots x_n)\mathrm{a})! \quad \longrightarrow_\beta \quad a[x_* := \mathbf{err}] \tag{5.2}$$

where x_* denotes any unbound variable. The **err** constant is represented in λN as a lambda abstraction that always reduces to itself—and that in object-oriented systems corresponds to the “message not understood” error.

The main difference with λ-calculus is that the *lambda-bind* rule (5.1) performs a substitution without removing the outermost abstraction level (the ‘λ’), while the *lambda-close* rule (5.2) removes the ‘λ’ and substitutes any remaining unbound variables by **err** (the constant representing run-time errors).

Lexical scoping of variables is treated in the following way: a local declaration (a variable declared as a parameter of a lambda abstraction) takes precedence

[1] Notation remark: we will use x to denote variables; and $\mathrm{a}, \mathrm{b}, \mathrm{c}, \ldots$ to denote terms.

over the declaration of a variable with the same name in outer lambda abstractions. In case the same name is used at different lambda abstraction levels, λN provides a scope escape operator '\' (backslash). The use of n occurrences of this escape operator preceding a variable x specifies that the corresponding declaration of x will be looked for ignoring the inner n abstraction levels. For instance, consider the expression

$$\lambda(xy)\lambda(xz)x + y + z + \backslash x$$

and assume that infix addition is part of the language; then x and $\backslash x$ are references to different variables. The first x reference is bound to the x parameter defined in the inner lambda abstraction $\lambda(xz)$, while $\backslash x$ reference is bound to the x parameter defined in the outer lambda abstraction $\lambda(xy)$.

Recursion is defined in λN by means of a fixed point operation. Specifically, a fixed point operation over a functional $\lambda(x)a$ corresponds to the usual combinator Y in the pure lambda calculus. An expression with recursion over parameter x is written $\mu(x)a$ and the translation **T** into de basic syntax is the following:

$$\begin{array}{lcl} \mathbf{T}(\mu(x)a) & = & Y_x(x \rightarrow \lambda(x)a) \\ Y_x & = & \lambda(x)(\lambda(x)\backslash x(x \rightarrow x(x \rightarrow x)!)!)(x \rightarrow (\lambda(x)\backslash x(x \rightarrow x(x \rightarrow x)!)!)) \end{array}$$

Extensible records

From the basic λN syntax, *extensible records* can be introduced in the λN formalism [Dami, 1994]. Extensible records are written with braces; fields are written with the $\doteq$ symbol; field selection uses the common dot notation. The syntax of these constructs and the translation **T** into the basic syntax is the following:

$$\begin{array}{lcl} \mathbf{T}(\{x_1 \doteq a_1 \ldots x_n \doteq a_n\}) & = & \lambda(sel)sel(x_1 \rightarrow a_1)\ldots(x_n \rightarrow a_n) \\ \mathbf{T}(a.x) & = & a(sel \rightarrow \lambda(x).x)!! \end{array}$$

A record is translated to a function which takes a selector *sel* and binds all fields to corresponding named parameters in that selector. A selector for field x is just an identity function on that name, so a field selection operation simply binds the appropriate selector to the *sel* argument of the record. Notice that the field selection operation is defined in basic syntax as an identity function plus two close operations. The first close operation is used to close the bindings of record a, while the second close operation binds the appropriate selector to the *sel* argument of the record.

Next, a record concatenation operation '$\ll$' can be introduced providing an incremental mechanism of record construction. The $a \ll b$ expression yields a new record from a concatenation of two records a and b in a right-preferential order. That is to say, fields declared in b override the fields also declared in a. The correspondent translation into the basic λN syntax is the following:

$$\mathbf{T}(a \ll b) = a(sel \rightarrow \mathbf{T}(b))!$$

For instance, the following expression:

$\{x \doteq 1 \quad y \doteq 2\} \ll \{x \doteq 4 \quad z \doteq 5\}$

yields a new record with three fields x, y, z containing the values 4,2,5 respectively:

$\{x \doteq 4 \quad y \doteq 2 \quad z \doteq 5\}$

Recursion can be used to define recursive records. The syntax of records is enlarged for recursive records as follows:

$$\mu(rec)\{x_1 \doteq a_1 \ldots x_n \doteq a_n\}$$

For instance we can define the following recursive record

$$\begin{array}{lll} Seasons = \mu(rec) & \{ & spring \doteq \{name \doteq \text{“spring”} \quad next \doteq rec.summer\} \\ & & summer \doteq \{name \doteq \text{“summer”} \quad next \doteq rec.autumn\} \\ & & autumn \doteq \{name \doteq \text{“autumn”} \quad next \doteq rec.winter\} \\ & & winter \doteq \{name \doteq \text{“winter”} \quad next \doteq rec.spring\} \\ & \} & \end{array}$$

and, for example, $Seasons.summer.next.next.name$ yields “winter”.

When we define recursive records at different levels, the ‘\’ scope escape operator allows references to these different levels. Using the ‘\’ notation, we can compact the syntax for describing recursive record structures by removing the explicit reference to *rec*. For instance, the previous *Seasons* recursive record can be described using compact notation as follows:

$$\begin{array}{lll} Seasons = & \{ & spring \doteq \{name \doteq \text{“spring”} \quad next \doteq \backslash\backslash summer\} \\ & & summer \doteq \{name \doteq \text{“summer”} \quad next \doteq \backslash\backslash autumn\} \\ & & autumn \doteq \{name \doteq \text{“autumn”} \quad next \doteq \backslash\backslash winter\} \\ & & winter \doteq \{name \doteq \text{“winter”} \quad next \doteq \backslash\backslash spring\} \\ & \} & \end{array}$$

Given the basic notions about λN calculus, next we describe the translation rules from Noos descriptions to λN calculus.

5.2 Noos formal syntax

In this section we will use the λN formalization of extensible records as a basis to formalize Noos descriptions as feature terms. The Noos refinement mechanism will be formalized by means of the record concatenation operation.

There are two types of feature terms in Noos: *constant feature terms* and *evaluable feature terms*. Feature terms are defined on a signature Σ composed by a set of sort names, a set of method names, and a set of feature names. Formally,

Definition 5.1 *(Noos Signature)*
We define the Noos *signature as the tuple* $\Sigma = \{\langle \mathcal{S}, \leq_s \rangle, \langle \mathcal{M}, \leq_m \rangle, \mathcal{F}\}$ *such that:*

- $\mathcal{S}$ *is a finite set of sort symbols including* $\perp_s$ *and* $\leq_s$ *is an order relation such that* $\perp_s$ *is the smallest element.*
- $\mathcal{M}$ *is a finite set of method symbols including* $\perp_m$ *and* $\leq_m$ *is an order relation such that* $\perp_m$ *is the smallest element.*
- $\mathcal{F}$ *is a finite set of* feature symbols*;*

Given the Noos signature Σ, we define the set Γ of feature terms as a union of constant feature terms and evaluable feature terms:

$$\Gamma = \Gamma_c \cup \Gamma_m$$

In the next section we will introduce Noos formal syntax for constant feature term step to step. The complete description of the formal syntax of Noos is given in Figure 5.2. Evaluable feature terms will be introduced in Section 5.9.

5.2.1 Constant feature terms

Given the signature Σ, we define *constant feature terms* as follows[2]:

Definition 5.2 *A constant feature term* ψ *is an expression of the form:*

$$\psi \quad ::= [s\ f_1 \doteq \mathrm{fv}_1 \cdots f_n \doteq \mathrm{fv}_n]$$

where s *is a sort in* $\mathcal{S}$*,* $f_1, \cdots, f_n$ *are pairwise distinct features in* $\mathcal{F}$*,* $n \geq 0$*, and each* fv_i *is a feature value.*

A feature term is translated to λN syntax, using the translation rule $\mathbf{T}$, as a recursive record as follows:

$$\mathbf{T}(s\,[f_1 \doteq \mathrm{fv}_1 \cdots f_n \doteq \mathrm{fv}_n]) \;=\; \{f_1 \doteq \mathrm{fv}_1 \cdots f_n \doteq \mathrm{fv}_n\}$$

The sort of a feature term (noted $\tau(\psi) = s$) is a component not represented in the λN notation and will be explained in Section 5.2.2.

Feature values

There are three kinds of feature values:

$$\begin{array}{rcl} \mathrm{fv} & ::= & \psi \\ & | & \psi_1 \cdots \psi_n \\ & | & \mathrm{ref} \end{array}$$

where the first one is a term; the second is a set of terms; and the third is a path reference (described in Section 5.2.3). We have shown in Section 3.2.4 that feature values can also be described using closed methods, and in Section 5.9 we will extend the syntax for incorporating closed methods.

[2]We use two equivalent notations for feature terms: the first one is as an horizontal enumetarion of features (the notation used here). The second notation is the vertical enumeration of features (useful to represent feature terms inside feature terms).

5.2.2 The sort of a feature term

From the set Γ of feature terms we define a sorting function $\tau : \Gamma \rightarrow \mathcal{S} \cup \mathcal{M}$, assigning either a sort symbol or a method symbol to each feature term. A constant feature term has assigned a sort symbol. An evaluable feature term has assigned a method symbol. Specifically, given a feature term ψ defined as follows,

$$\psi \;=\; [s\ f_1 \doteq \mathrm{fv}_1 \cdots f_n \doteq \mathrm{fv}_n]$$

the sorting function τ for ψ is defined as $\tau(\psi) = s$. We will say that the sort of ψ is s.

5.2.3 Path references

A path reference has two components: a *head* and a *path*. A head is constructed using the backslash notation (for instance, '\\') A path is constructed as a concatenation of field selections using the dot notation (for instance '$f_1.f_2$'). The formal syntax for path references is the following:

$$\begin{array}{lcl} \mathrm{ref} & ::= & \mathrm{head} \mid \mathrm{head\ path} \\ \mathrm{head} & ::= & \backslash\mathrm{head} \quad \mid \quad \backslash \\ \mathrm{path} & ::= & f.\mathrm{path} \quad \mid \quad f \end{array}$$

An example of a path reference is $\backslash\backslash f_1.f_2$. A path reference may have only a sequence of backslashes (for instance \\). Path references are used to provide a formalization of Noos relative path references and a precise interpretation of the rules of scope and refinement (see Section 5.3).

Reference over sets

Since feature values can be sets, path references have to deal with feature values that are sets. The field selection operator over sets is defined as the pointwise extension of the λN field selection operator. The result of a field selection over a set is a new set. Therefore, we define the field selection operator over sets as follows:

$$S.f \equiv \bigcup_{s \in S} s.f$$

5.2.4 Refinement

Feature terms are always constructed in the Noos language by refinement. The definition of a feature term ψ by refinement of another feature term ψ' will be formalized as a record concatenation as follows:

$$\psi \;\;::=\;\; \psi' \ll [s\ f_1 \doteq \mathrm{fv}_1 \cdots f_n \doteq \mathrm{fv}_n]$$

$$
\begin{array}{llll}
\psi & ::= & [s\ f_1 \doteq \mathrm{fv} \cdots f_n \doteq \mathrm{fv}] & \text{; feature term} \\
 & | & \psi \ll [s\ f_1 \doteq \mathrm{fv} \cdots f_n \doteq \mathrm{fv}] & \text{; refinement} \\
 & & & \\
\mathrm{fv} & ::= & \psi & \text{; record} \\
 & | & \psi_1 \cdots \psi_n & \text{; set of records} \\
 & | & \mathrm{ref} & \text{; reference} \\
 & & & \\
\mathrm{ref} & ::= & \mathrm{head} \mid \mathrm{head\ path} & \text{; path reference} \\
 & & & \\
\mathrm{head} & ::= & \backslash\mathrm{head} \quad | \quad \backslash & \text{; outer reference} \\
 & & & \\
\mathrm{path} & ::= & f.\mathrm{path} \quad | \quad f & \text{; path}
\end{array}
$$

Figure 5.2. Formal syntax of Noos using the λN approach.

$$
\begin{array}{lll}
\Upsilon(\mathtt{desc}) & = & \Upsilon_d(\mathtt{desc}, 0) \\
 & & \\
\Upsilon_d((\mathtt{define\ (const)} & = & \Upsilon(\mathtt{const}) \ll [\mathit{const}\ \Upsilon_f(\mathtt{fdesc}_1, \ell) \\
\quad \mathtt{fdesc}_1 \cdots \mathtt{fdesc}_n), \ell) & & \quad \cdots \Upsilon_f(\mathtt{fdesc}_n, \ell)] \\
 & & \\
\Upsilon_f((\mathtt{f\ desc}), \ell) & = & \mathtt{f} \doteq \Upsilon_d(\mathtt{desc}, \ell + 1) \\
\Upsilon_f((\mathtt{f\ desc}_1 \cdots \mathtt{desc}_n), \ell) & = & \mathtt{f} \doteq \Upsilon_d(\mathtt{desc}_1, \ell + 1) \cdots \Upsilon_d(\mathtt{desc}_n, \ell + 1) \\
\Upsilon_f((\mathtt{f\ ref}), \ell) & = & \mathtt{f} \doteq \Upsilon_r(\mathtt{ref}, \ell + 1) \\
 & & \\
\Upsilon_r((\mathtt{>>\ f}_1 \cdots \mathtt{f}_n), \ell) & = & \underbrace{\backslash \cdots \backslash}_{\ell} f_n . \cdots . f_1
\end{array}
$$

Figure 5.3. Translation rules from Noos syntax to λN syntax.

where ψ is defined with sort s ($\tau(\psi) = s$) and by extending ψ' with $f_1, \ldots, f_n$ features.

Note that concatenation operation $\ll$ is a right-preferential order operation. This means that all the definitions of features $f_1, \ldots, f_n$ also defined in ψ' override the definitions given in ψ'.

5.3 Translation rules from Noos to λN

We have introduced the formal syntax of Noos that is fully described in Figure 5.2. Now, we will introduce the translation rules Υ from Noos syntax to Noos formal syntax. There are three kinds of translation rules: translation rules for descriptions Υ_d, translation rules for features Υ_f, and translation rules for references Υ_r. We will explain these translation rules incrementally. A complete description of translation rules is given in Figure 5.3.

The Noos language allows the use of names for describing feature values. In Noos formal syntax feature terms have no name. Therefore, before translating a given description D to the formal syntax, we have to perform a preprocess where each name reference is substituted by either the description that it denotes or by a path reference to another feature with the same name reference as feature value. The specification of the same name reference as the feature value of different features requires the satisfaction of the *path equality* property (see Section 3.2.3). In order to preserve this path equality only one occurrence of a name reference will be translated to the description that it denotes. In the rest of occurrences the name reference will be translated to a path reference to the first occurrence.

Translation rules have two parameters. The first parameter contains a description (the text to be translated). The second parameter holds and integer indicating the depth level of the description according to the root description. The depth level will be used to translate path references.

A Noos description `desc` is translated with a description rule starting with depth level zero.

$$\Upsilon(\texttt{desc}) \;=\; \Upsilon_d(\texttt{desc}, 0)$$

Since a feature term is built by refinement of another feature term, a description is translated as the concatenation of two records applying the following rule:

$$\begin{array}{lcl} \Upsilon_d(\texttt{(define (const)} & = & \Upsilon(\texttt{const}) \ll [\mathit{const}\ \Upsilon_f(\texttt{fdesc}_1, \ell) \\ \quad\quad \texttt{fdesc}_1 \cdots \texttt{fdesc}_n\texttt{)}, \ell) & & \quad\quad\quad\quad \cdots \Upsilon_f(\texttt{fdesc}_n, \ell)] \end{array}$$

where the feature term is built concatenating (refining) the feature term denoted by `const` ($\Upsilon(\texttt{const})$) with a record composed by the features defined in the body of the description translated using translation rules for features Υ_f.

Given the description `fdesc` of a feature as the pair `(f v)`, translation rules Υ_f define a field of a record with name `f` and with value that obtained from the translation of `v` to formal syntax increasing the depth level by one. There is one translation rule Υ_f for each kind of feature value:

$$\begin{array}{lcl} \Upsilon_f\,(\texttt{(f desc)}, \ell) & = & \texttt{f} \doteq \Upsilon_d(\texttt{desc}, \ell + 1) \\ \Upsilon_f\,(\texttt{(f desc}_1 \cdots \texttt{desc}_n\texttt{)}, \ell) & = & \texttt{f} \doteq \Upsilon_d(\texttt{desc}_1, \ell + 1) \cdots \Upsilon_d(\texttt{desc}_n, \ell + 1) \\ \Upsilon_f\,(\texttt{(f ref)}, \ell) & = & \texttt{f} \doteq \Upsilon_r(\texttt{ref}, \ell + 1) \end{array}$$

New (sub)descriptions are translated using Υ_d rules. Path references are translated using the following Υ_r rule:

$$\Upsilon_r\,(\texttt{(>> f}_1 \cdots \texttt{f}_n\texttt{)}, \ell) \;=\; \underbrace{\backslash \cdots \backslash}_{\ell} f_n. \cdots .f_1$$

Note that a Noos path reference such that `(>> father mother)` is built as a concatenation of field selections in reverse order, i.e. `mother.father`, and containing in the head as many backslashes as the depth level.

Absolute path references are not allowed in the formal language. This means that they have to be translated to relative path references. There are two ways of translating absolute path references:

```
(define (Person :id Paul)
  (spouse Mary)
  (child (>> child of Mary)))

(define (Person :id Mary)
  (spouse Paul)
  (child (define (person)
           (age 3))))

(define (Person)
  (spouse (define (person)
            (spouse (>>))
            (child (define (person)
                     (age 3)))))
  (child (>> child of (>> spouse))))
```

Figure 5.4. Substitution of name references. Given two descriptions of `Paul` and `Mary`, `Paul` description is translated to the last description where the name references to `Mary` are substituted by its description in feature `spouse` and by a relative path reference in the absolute path reference `(>> child of Mary)` in feature `child`.

- A first case is when an absolute path reference `(>> F of D)` has a name reference `D` that it occurs as a feature value in another feature `F'`. In this case, the name reference `D` can be substituted to a relative path to the `F'` feature. Figure 5.4 shows an example of a substitution of an absolute path reference.

- A second is when an absolute path reference `(>> F of D)` has a name reference `D` that it does not occur as a feature value in another feature. In this situation, this name reference cannot be substituted by a path reference. We can extend the description with a new feature that has the name reference as value. Consequently, the name reference in the absolute path can now be substituted to a relative path to the new feature.

We are now ready to show, using an example, how a description in Noos syntax is translated to Noos formal syntax. Taking as example the previous description of `Peter`:

```
(define (person id:  Peter)
  (age 28)
  (drives (define (Car)
            (owner (>>))
            (complaint does-not-start)
            (gas-level-in-tank full))))
```

where Peter is defined by refinement of Person and the car that Peter is driving is defined by refinement of Car. Moreover, Car is defined as follows

```
(define Car
  (owner (define (person)))
  (gas-level-in-tank level)
  (gas-gauge-reading (>> gas-level-in-tank)))
```

and person, does-not-start, full, and level are defined with no features.

Then, the feature term that represents Peter description is built from the two following concatenations[3]:

$$\Upsilon(\texttt{Peter}) = \Upsilon(\texttt{Person}) \ll \begin{bmatrix} \textsf{Person} & & \\ \mathit{age} & \doteq & \mathit{28} \\ \mathit{drives} & \doteq \Upsilon(\texttt{Car}) \ll & \begin{bmatrix} \textsf{Car} & & \\ \mathit{owner} & \doteq & \backslash\backslash \\ \mathit{complaint} & \doteq & \mathit{does\text{-}not\text{-}start} \\ \mathit{gas\text{-}level\text{-}in\text{-}tank} & \doteq & \mathit{full} \end{bmatrix} \end{bmatrix}$$

Next, applying the operation of record concatenation to the description of that car we will obtain gas-gauge-reading? and empty-level? feature descriptions from the feature term car. Thus, after the concatenation operation, and since person is a feature term without features, we will obtain the following term:

$$\Upsilon(\texttt{Peter}) = \begin{bmatrix} \textsf{Person} & & \\ \mathit{age} & \doteq & \mathit{28} \\ \mathit{drives} & \doteq & \begin{bmatrix} \textsf{Car} & & \\ \mathit{owner} & \doteq & \backslash\backslash \\ \mathit{complaint} & \doteq & \mathit{does\text{-}not\text{-}start} \\ \mathit{gas\text{-}level\text{-}in\text{-}tank} & \doteq & \mathit{full} \\ \mathit{gas\text{-}gauge\text{-}reading} & \doteq & \backslash \mathit{gas\text{-}level\text{-}in\text{-}tank} \end{bmatrix} \end{bmatrix}$$

Notice that the reference in the owner feature is to Peter (two backslashes), and the reference in feature gas-gauge-reading is to the feature gas-level-in-tank in car (one backslash).

An additional notion that will be used later is the notion of *subterm* of a feature term:

Definition 5.3 *(Subterm)*
A feature term ψ_i is a subterm of a feature term ψ if ψ_i can be accessed from ψ following a path $f_1.\cdots.f_n$.

For instance, in the previous example of Peter feature term the car feature term is a subterm accessible from Peter by the path constituted by the drives feature.

[3]We write the sort assigned to a feature term in the head of the term in sans serif font, like Person or Car in the example.

5.4 Using variables in feature terms

Feture terms can also be represented using variables instead of path references. Variables, as described in [Aït-Kaci and Podelski, 1993], provide a notation based on tags. The motivation of using this new representation for feature terms is that variables provide a more adequate notation for defining the semantical interpretation of feature terms.

A feature term can be represented using variables by means of assigning a different variable to each feature term and replacing path references by the variable assigned to the referenced feature term. Specifically, a feature term, using variables, is defined as follows:

Definition 5.4 *Given the signature Σ and a set $\mathcal{V}$ of variables, we define a feature term ψ as an expression of the form:*

$$\psi \quad ::= \quad X : s\,[f_1 \doteq \Psi_1 \cdots f_n \doteq \Psi_n]$$

where X is a variable in $\mathcal{V}$; s is a sort in $\mathcal{S}$; $f_1, \cdots, f_n$ are pairwise distinct features in $\mathcal{F}$; $n \geq 0$; each Ψ_i is a set of feature terms and variables; and at most one occurrence of each variable is sorted.

We call the variable X in the above feature term the *root* of ψ (noted $Root(\psi) = X$), and say that X is *sorted* by the sort s (noted $\tau(X) = s$). The set of variables and the set of features occurring in ψ are noted respectively as $\mathcal{V}_\psi$ and $\mathcal{F}_\psi$.

The use of variables instead of path references in feature values implies that path equality will be represented as equalities of variables.

For instance, the `Peter` feature term can be described using variables instead of path references as follows:

$$\text{Peter} = X : \textit{Person}\begin{bmatrix} \textit{age} & \doteq X_1 : \textit{28} \\ \textit{drives} & \doteq X_2 : \textit{Car}\begin{bmatrix} \textit{owner} & \doteq X \\ \textit{complaing} & \doteq X_{21} : \textit{does-not-start} \\ \textit{gas-level-in-tank} & \doteq X_{22} : \textit{full} \\ \textit{gas-gauge-reading} & \doteq X_{22} \end{bmatrix} \end{bmatrix}$$

Note that the `Peter` term has a path equality between features `gas-level-in-tank` and `gas-gauge-reading` and another path equality between the `owner` feature and `Peter` (null path).

5.5 Semantics

We have presented the formal syntax for Noos and the set of translation rules from descriptions syntax to feature terms syntax. Next, we will describe the semantical interpretation of feature terms. We have argued our approach to construct terms embodying partial information. In fact, we have said that feature terms are interpreted as partial descriptions.

The semantics of Noos is constructed to capture this notion of feature terms as partial descriptions denoting sets of individuals in a given domain. First of all, we define an interpretation $\mathcal{I}$ over the Noos signature Σ.

Definition 5.5 *(Interpretation)*
We define an interpretation $\mathcal{I}$ over the signature $\Sigma = \{\langle \mathcal{S}, \leq_s \rangle, \langle \mathcal{M}, \leq_m \rangle, \mathcal{F}\}$ as the structure

$$\mathcal{I} = \{\mathbb{S}, \mathbb{F}, \mathbb{M}\}$$

such that:

1. $\mathbb{S}$ *is a non-empty set, called* domain *of $\mathcal{I}$ (or, universe);*
2. $\mathbb{F}$ *is a set of total unary functions $f : \mathbb{S} \mapsto \mathcal{P}(\mathbb{S})$;*
3. $\mathbb{M}$ *is a set of total functions $m : \mathcal{P}(\mathbb{S}) \times \cdots \times \mathcal{P}(\mathbb{S}) \mapsto \mathcal{P}(\mathbb{S})$;*
4. $\forall s \in \mathcal{S} : [\![s]\!]^{\mathcal{I}} \subseteq \mathbb{S}$ *and* $\forall s, s' \in \mathcal{S} : s \leq s' \Leftrightarrow [\![s]\!]^{\mathcal{I}} \supseteq [\![s']\!]^{\mathcal{I}}$*;*
5. $\forall f \in \mathcal{F} : [\![f]\!]^{\mathcal{I}} \subseteq \mathbb{F}$*;*
6. $\forall m \in \mathcal{M} : [\![m]\!]^{\mathcal{I}} \subseteq \mathbb{M}$*;*
7. $[\![\bot_s]\!]^{\mathcal{I}} = \mathbb{S}$*; and*
8. $[\![\bot_m]\!]^{\mathcal{I}} = \mathbb{M}$.

From interpretation $\mathcal{I}$, constant feature terms are interpreted as partial descriptions denoting sets of elements in the domain $\mathbb{S}$ under all possible valuations of its variables $\mathcal{V}_\psi$ in $\mathbb{S}$.

Specifically, the interpretation of a feature term under a specific valuation α of its variables is given as:

Definition 5.6 *(Denotation of a constant feature term under a valuation α)*
Given the interpretation $\mathcal{I}$, the denotation $[\![\psi]\!]^{\mathcal{I},\alpha}$ of a constant feature term ψ, under the valuation $\alpha : \mathcal{V}_\psi \rightarrow \mathbb{S}$ is defined by:

Let $\psi = X : s[f_1 \doteq \Psi_1 \cdots f_n \doteq \Psi_n]$

$$[\![\psi]\!]^{\mathcal{I},\alpha} = \begin{cases} \{\alpha(X)\} & \textit{when } \alpha(X) \in [\![s]\!]^{\mathcal{I}} \textit{ and } \forall_{i=1,\ldots,n} : [\![f_i]\!]^{\mathcal{I}}(\alpha(X)) \in [\![\Psi_i]\!]^{\mathcal{I},\alpha} \\ \emptyset & \textit{otherwise} \end{cases}$$

expressing that the element assigned to the root variable X (noted $\alpha(X)$) has to belong to the set of elements denoted by its sort $[\![s]\!]^{\mathcal{I}}$, and for each feature f_i the value of the function that denotes $[\![f_i]\!]^{\mathcal{I}}(\alpha(X))$ has to belong to the denotation of the corresponding subterm $[\![\Psi_i]\!]^{\mathcal{I},\alpha}$ under the same valuation.

Since feature values Ψ_i are sets of feature terms, we have to define the interpretation $[\![\psi_1 \cdots \psi_m]\!]^{\mathcal{I},\alpha}$ of sets of feature terms. Sets are interpreted on $\mathcal{P}(\mathbb{S})$ as follows:

Definition 5.7 *(Semantics of sets)*
Given a set $\Psi = \psi_1 \ldots \psi_m$, the denotation $[\![\Psi]\!]^{\mathcal{I},\alpha}$ under the valuation α is given by:

$$[\![\psi_1 \ldots \psi_m]\!]^{\mathcal{I},\alpha} = \{w \in \mathcal{P}(\mathbb{S}) \mid \exists e_1 \in w \cdots \exists e_m \in w(\forall_{i=1,\ldots,m} : e_i \in [\![\psi_i]\!]^{\mathcal{I},\alpha} \text{ and } \forall_{i,j=1,\ldots,m} : i \neq j \Rightarrow e_i \neq e_j)\}$$

where for all $\psi_i = X \in \mathcal{V}_\psi$ then $[\![X]\!]^{\mathcal{I},\alpha} = \{\alpha(X)\}$

Finally, constant feature terms are interpreted as the union of domain elements denoted by all the valuations of $\mathcal{I}$ (noted $Val(\mathcal{I})$), as follows:

Definition 5.8 *(Semantics of a constant feature term)*
Given the interpretation $\mathcal{I}$, the denotation $[\![\psi]\!]^{\mathcal{I}}$ of a constant feature term ψ is given by:

$$[\![\psi]\!]^{\mathcal{I}} = \bigcup_{\alpha \in Val(\mathcal{I})} [\![\psi]\!]^{\mathcal{I},\alpha}$$

Using this semantical interpretation of feature terms, it is legitimate to establish an order relation between terms. Given two terms ψ and ψ', we will be interested in determine when $[\![\psi]\!]^{\mathcal{I}} \subset [\![\psi']\!]^{\mathcal{I}}$. In other words, we want to determine when a feature term ψ is more specific (contains more information) than another feature term ψ'.

5.6 Term subsumption

We have just seen that the semantical interpretation of feature terms allows to define an ordering relation between feature descriptions. We call this ordering relation *subsumption*. The intuitive meaning of subsumption is that of *informational ordering*. We say that a feature term ψ subsumes another feature term ψ' (noted $\psi \sqsubseteq \psi'$) when all information in ψ is also contained in ψ'. Formally,

Definition 5.9 *(Subsumption)*
Given two feature terms ψ and ψ', ψ subsumes ψ', $\psi \sqsubseteq \psi'$, if there is a total mapping function $\upsilon : \mathcal{V}_\psi \to \mathcal{V}_{\psi'}$ such that :

1. $\upsilon(Root(\psi)) = Root(\psi')$,

and $\forall x \in \mathcal{V}_\psi$

2. $\tau(x) \leq \tau(\upsilon(x))$,
3. *for every $f_i \in \mathcal{F}$ such that $x.f_i \doteq \Psi_i$ is defined, then $\upsilon(x).f_i \doteq \Psi'_i$ is also defined, and*
   (a) $\forall \psi_k \in \Psi_i$ *either* $\exists \psi'_k \in \Psi'_i$ *such that* $\upsilon(Root(\psi_k)) = Root(\psi'_k)$
   or $\exists x' \in \Psi'_i$ *such that* $\upsilon(Root(\psi_k)) = x'$,

(b) $\forall x \in \Psi_i$ *either* $\exists \psi'_k \in \Psi'_i$ *such that* $\upsilon(x) = Root(\psi'_k)$
or $\exists x' \in \Psi'_i$ *such that* $\upsilon(x) = x'$,

(c) $\forall \psi_k, \psi'_k \in \Psi_i \ (\psi_k \neq \psi'_k \Rightarrow \upsilon(Root(\psi_k)) \neq \upsilon(Root(\psi'_k)))$,

(d) $\forall x, \psi'_k \in \Psi_i \ (\upsilon(x) \neq \upsilon(Root(\psi'_k)))$,

(e) $\forall x, y \in \Psi_i \ (x \neq y \Rightarrow \upsilon(x) \neq \upsilon(y))$.

For instance, given the following two feature terms:

$$\mathrm{T1} = X : Person \begin{bmatrix} name & \doteq X_1 : Name \begin{bmatrix} first \doteq X_{11} : Michael \\ last \doteq X_{12} : Smith \end{bmatrix} \\ lives\text{-}at & \doteq X_2 : Address\big[city \doteq X_{21} : NYCity\big] \\ father & \doteq X_3 : Person\big[name \doteq X_{31} : Name\big[last \doteq X_{12}\big]\big] \end{bmatrix}$$

$$\mathrm{T2} = Y : Person \begin{bmatrix} name & \doteq Y_1 : Name\big[last \doteq Y_{11} : family\text{-}name\big] \\ father & \doteq Y_2 : Person\big[name \doteq Y_{21} : Name\big[last \doteq Y_{11}\big]\big] \end{bmatrix}$$

where `Smith` is a kind of `family-name` (`family-name` $\leq$ `Smith`), we have that T2 $\sqsubseteq$ T1 since T1 contains all the information given in T2 (including the fact that the last name of a person is a `family-name` and is the same that the last name of her father). Moreover, T1 specializes T2 specifying a partial description of her home address, specifying her first name and a specific family-name.

Notice that definition 5.9 of subsumption provides a concrete interpretation of the subsumption between two sets: given two sets of feature terms Ψ, Ψ' we say that $\Psi \sqsubseteq \Psi'$ if the terms provided in Ψ are extended and refined in Ψ'; formally,

Definition 5.10 *(Set subsumption)*
Given two sets of feature terms Ψ, Ψ' *we say that* $\Psi \sqsubseteq \Psi'$ *if for each* $\psi \in \Psi$, *there is a different* $\psi' \in \Psi'$ *such that* $\psi \sqsubseteq \psi'$.

From the definition of subsumption of two feature terms ψ and ψ', it can be easily proved that each subterm ψ_i of ψ also subsumes its correspondent ψ'_i of ψ'. Formally,

Lemma 5.1 *Given two feature terms* ψ *and* ψ', *such that* ψ *subsumes* ψ', *if* $\psi.f_1 \cdots f_n$ *is defined then:*

$$\psi.f_1 \cdots f_n \sqsubseteq \psi'.f_1 \cdots f_n$$

Proof: Let υ *a total mapping function satisfying the subsumption requirements for* $\psi \sqsubseteq \psi'$ *and* $\psi.f_1 \cdots f_n = \Psi$; *since* $\mathcal{V}_\Psi \subseteq \mathcal{V}_\psi$ *we can take the same mapping function restricted to* $\mathcal{V}_\Psi$ *to prove the result.*

The subsumption operation captures the notion of informational ordering. In fact, using subsumption we want to capture the notion of semantical inclusion. When we say that a feature term ψ subsumes another feature term ψ' ($\psi \sqsubseteq \psi'$), we understand that the denotation of ψ ($[\![\psi]\!]^{\mathcal{I}}$) includes the denotation of ψ' ($[\![\psi']\!]^{\mathcal{I}}$). Formally,

Theorem 5.1

$$\psi \sqsubseteq \psi' \Longrightarrow [\![\psi]\!]^{\mathcal{I}} \supseteq [\![\psi']\!]^{\mathcal{I}}$$

Proof:

Since $\psi \sqsubseteq \psi'$, there is a mapping function $\upsilon : \mathcal{V}_{\psi} \rightarrow \mathcal{V}_{\psi'}$ satisfying the subsumption requirements.

Then, for all non empty valuation α' of ψ' we can construct a non empty valuation α of ψ as follows:

$$\alpha(x) = \alpha'(\upsilon(x))$$

Having this valuation, and using Definition 5.9.2, for all $x \in \mathcal{V}_{\psi}$ then $\tau(x) \leq \tau(\upsilon(x))$ is satisfied.

Then, Definition 5.5.4 assures that $[\![\tau(x)]\!]^{\mathcal{I},\alpha} \supseteq [\![\tau(\upsilon(x))]\!]^{\mathcal{I},\alpha}$. In particular, we have that $\alpha(Root(\psi)) \in [\![\tau(Root(\psi))]\!]^{\mathcal{I},\alpha}$.

Next,

1. *if ψ has no features we have that $[\![\psi]\!]^{\mathcal{I}} \supseteq [\![\psi']\!]^{\mathcal{I}}$ and the theorem is proved.*
2. *otherwise for all f such that $\psi.f \doteq \Psi$ is defined, using Definition 5.9.3, we have that $\psi'.f \doteq \Psi'$ is also defined.*

   Finally, since Lemma 5.1 assures that $\Psi \sqsubseteq \Psi'$, by induction we have that $f^{\mathcal{I}}(\alpha(Root(\psi))) \in [\![\Psi]\!]^{\mathcal{I},\alpha}$ and then the theorem is also proved.

From the notion of subsumption we define following additional notion of *equivalence*:

Definition 5.11 *(Equivalence)*
Given two feature terms ψ and ψ', we say that they are syntactic variants if and only if $\psi \sqsubseteq \psi'$ and $\psi' \sqsubseteq \psi$.

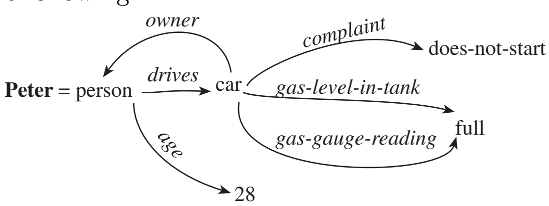
The intuition about the notion of equivalence is that two feature terms are equivalent when contain the same information.

5.7 Representing feature terms as labeled graphs

Feature terms can have an equivalent form of representation as labeled directed graphs [Carpenter, 1992] [Aït-Kaci and Podelski, 1993] [Backofen and Smolka, 1995]. This representation is interesting because it offers an intuitive and visual syntax.

A feature term can be represented as a labeled directed graph that has, for each variable $X : s$, a node q labeled with sort s, and having an arc from q to another node q' labeled by f, written $q \xrightarrow{f} q'$, for each feature f defined in q with feature value q'.

For instance, the graph representation of the feature term of `Peter` from Section 5.4 is the following:



Formally, given the signature Σ, having a set of sort symbols $\mathcal{S}$ and a set of feature symbols $\mathcal{F}$ we define a feature term ψ as a labeled graph as follows:

Definition 5.12 *(Labeled graph representation of feature terms)*
A feature term is a tuple $\psi = < \mathcal{Q}, \bar{q}, \theta, \delta >$ *where:*

- $\mathcal{Q}$ *: is a finite set of nodes rooted at* $\bar{q}$
- $\bar{q}$ *: is the root node*
- $\theta : \mathcal{Q} \rightarrow \mathcal{S}$ *: a total node typing function.*
- $\delta : \mathcal{F} \times \mathcal{Q} \rightarrow 2^{\mathcal{Q}}$ *: is a partial feature value function*

where θ determines the labels of nodes, and $\delta(f, q) = q'$ implies that there is an arc labeled by f from q to another node q'.

In this labeled graph representation path references are defined as follows:

Definition 5.13 *(Paths)*
A path π *is composed of a sequence of features. Let* ϵ *the empty path, we extend* δ *to paths as follows:*

- $\delta(\epsilon, q) = q$
- $\delta(f\pi, q) = \delta(\pi, \delta(f, q))$

That is to say, if $\pi = f_1 \ldots f_n$ then $\delta(\pi, q_0) = q_n$ if $q_0 \xrightarrow{f_1} q_1 \xrightarrow{f_2} q_2 \cdots \xrightarrow{f_n} q_n$. The set $\{\delta(\pi, q) | \pi \text{ is a path}\}$ is the set of nodes *reachable* from q by some path. The root of a feature term is a node $\bar{q}$ that satisfies $Q = \{\delta(\pi, \bar{q}) | \pi \text{ is a path}\}$, i.e. the node from which all nodes in Q are reachable.

We say that there is a path equality when two different paths π_1 and π_2 lead from a same node q to the same node q' ($\delta(\pi_1, q) = q' = \delta(\pi_2, q)$).

Labeled graph representation of feature terms has been the basis for developing graphical browsers for the Noos development environment (see Appendix A).

5.8 Understanding feature terms as clauses

Feature terms can be also understood as conjunctions of clauses [Aït-Kaci and Podelski, 1993]. This clausal representation is useful and more usual for understanding learning methods. There are two kinds of atomic clauses: sort clauses ($X : s$) and feature clauses ($f(X, Y)$). A given feature can be represented also as a conjunction of these two kind of atomic clauses.

Thus, we associate each feature term $\psi = X : s[f_1 \doteq \psi_1 \cdots f_n \doteq \psi_n]$ with a clause $\phi(\psi)$ as follows:

$$\phi(\psi) = X : s \wedge f_1(X, Y_1) \wedge \phi(\psi_1) \wedge \cdots \wedge f_n(X, Y_n) \wedge \phi(\psi_n)$$

where $Y_1, \cdots, Y_n$ are roots of $\psi_1, \cdots, \psi_n$ respectively.

For instance, the `Peter`'s feature term can be represented as a clause in the following way:

$$\begin{array}{llll}
X : \text{Person} & \wedge\ \text{age}(X, 28) & & \\
 & \wedge\ \text{drives}(X, Y)\ \wedge\ Y : \text{Car} & \wedge\ \text{owner}(Y, X) & \\
 & & \wedge\ \text{complaint}(Y, Z)\ \wedge\ Z : \text{does-not-start} & \\
 & & \wedge\ \text{gas-level-in-tank}(Y, W)\ \wedge\ W : \text{full} & \\
 & & \wedge\ \text{gas-gauge-reading}(Y, W) &
\end{array}$$

5.9 Evaluable feature terms

Noos methods are defined as *evaluable feature terms*. Evaluable feature terms, like constant feature terms, are formalized as recursive records.

Given the signature $\Sigma = \{\langle \mathcal{S}, \leq_s \rangle, \langle \mathcal{M}, \leq_m \rangle, \mathcal{F}\}$ we define evaluable feature terms as follows:

Definition 5.14 *An evaluable feature term ψ is an expression of the form:*

$$\psi \quad ::= [m\ f_1 \doteq \Psi_1 \cdots f_n \doteq \Psi_n]$$

where m is a sort in $\mathcal{M}$, $f_1, \cdots, f_n$ are pairwise distinct features in $\mathcal{F}$, $n \geq 0$, and each Ψ_i is a set of feature terms and variables.

Notice that we said the feature value can be (any) feature term; in particular it can be a constant feature term or an evaluable feature term.

Below, we will describe how Noos built-in methods are formalizated as evaluable feature terms. Then, we will show how new methods are built by refinement.

Built-in methods

The set of Noos built-in methods is formalized as a collection of predefined *evaluable feature terms*. A built-in method is modelled as a recursive extensible record linked to a lambda abstraction. We call "required" features of the evaluable feature term those feature labels defined as the parameters of the lambda abstraction. For instance, the `subtract` built-in method is defined as a recursive extensible record linked to the $subtract_{def}$ lambda abstraction with two named parameters (called `amount` and `minus`) that encodes the usual subtraction operation of two numbers:

$$\begin{array}{lcl} \texttt{subtract} & = & [subtract\ amount \doteq number \quad minus \doteq number] \\ subtract_{def} & = & \lambda(amount\ minus)\ amount - minus \end{array}$$

Defining methods

New methods can be defined by refinement of built-in methods (or other methods). Refinement is modelled as record concatenation. For instance, we can define the method `minus-one` by refinement of the `subtract` method as the following record concatenation:

$$\texttt{minus-one} \ = \ \texttt{subtract} \ll [subtract\ minus \doteq 1]$$

Then, we can define another method by refinement of `minus-one`, binding the remaining `amount` parameter with a specific value obtaining a closed method amenable to be evaluated.

The complete list of built-in methods with their parameter names is described in Appendix D.

5.9.1 Defining methods in features

Closed methods can be evaluated to infer feature values. The evaluation of a method m in a feature f to infer its feature value is indicated with the '#' token using the following syntax:

$$f \doteq m\#$$

The syntax $m\#$ is translated to basic λN syntax using the translation rule $\mathbf{T}$ as follows:

$$\mathbf{T}(m\#) \ = \ m(sel \rightarrow \lambda_{def})!!$$

where λ_{def} is the lambda abstraction linked to the built-in method b such that $b \leq_m m$. That is to say, the evaluation of a method m is translated as the functional application of λ_{def} taking as values for the arguments of λ_{def} the feature values of the features with same name defined in m.

For instance, the following expression

$$[subtract\ amount \doteq 7 \quad minus \doteq 3]\#$$

is translated by the translation rule $\mathbf{T}$ to λN basic syntax as follows

$$\lambda(sel)sel(amount \rightarrow 7)(minus \rightarrow 3)(\lambda(amount\ minus)\ amount - minus)!!$$

that is reduced after a first step to

$$\lambda(amount\ minus)\ amount - minus\ (amount \rightarrow 7)(minus \rightarrow 3)!$$

and eventually yields 4 as result.

We say that a feature is *reduced* when the method incorporated to infer its feature value has been evaluated.

Using the notion of reduced features we introduce the notion of *normal form* for feature terms:

Definition 5.15 *(Normal feature term)*
A feature term ψ is in normal form when all their features ($\mathcal{F}_\psi$) are reduced (there is no feature with a method not yet evaluated).

5.9.2 Semantics of evaluable feature terms

Evaluable feature terms are interpreted, under an interpretation $\mathcal{I} = \langle \mathbb{S}, \mathbb{F}, \mathbb{M} \rangle$, as partial descriptions denoting sets of functions in $\mathbb{M}$. Analogously to constant feature terms, the interpretation of an evaluable feature term is given as the union of functions denoted by all the valuations of $\mathcal{I}$ ($Val(\mathcal{I})$) as follows:

Definition 5.16 *(Semantics of an evaluable feature term)*
Given the interpretation $\mathcal{I}$, the denotation $[\![\psi]\!]^{\mathcal{I}}$ of an evaluable feature term ψ is given by:

$$[\![\psi]\!]^{\mathcal{I}} = \bigcup_{\alpha \in Val(\mathcal{I})} [\![\psi]\!]^{\mathcal{I},\alpha}$$

and for each valuation α in $Val(\mathcal{I})$ its interpretation is given as follows:

Definition 5.17 *(Denotation of an evaluable feature term under a valuation α)*
Given the interpretation $\mathcal{I}$, the denotation $[\![\psi]\!]^{\mathcal{I},\alpha}$ of an evaluable feature term ψ, under a valuation $\alpha : \mathcal{V}_\psi \rightarrow \mathbb{M} \cup \mathbb{S}$ is inductively given by:

Let $\psi = X : m[f_1 \doteq \psi_1 \cdots f_n \doteq \psi_n]$

$$[\![\psi]\!]^{\mathcal{I},\alpha} = \begin{cases} \{\alpha(X)\} & \textit{when } \alpha(X) \in [\![m]\!]^{\mathcal{I}} \textit{ and } \forall_{i=1,\ldots,n} : [\![f_i]\!]^{\mathcal{I}}(\alpha(X)) \in [\![\Psi_i]\!]^{\mathcal{I},\alpha} \\ \emptyset & \textit{otherwise} \end{cases}$$

5.10 Preferences

Preferences are modeled by partially ordered sets (also called *posets*). A partially ordered set is a pair $\langle S, \prec \rangle$ composed by a set of elements S and a binary relation $\prec$ defined on S. We demand $\prec$ to satisfy the reflexive and transitive properties. In fact, we are asking $\langle S, \prec \rangle$ to form a pre-order. Formally,

Definition 5.18 *(Preference)*
A preference *is a pair $\langle S, \prec \rangle$, where S is a set of alternatives and $\prec$ is a binary relation over S such that it is reflexive ($a \prec a$) and transitive ($a \prec c$ when $a \prec b$ and $b \prec c$). When $a \prec b$ we say that a is* preferred *to b, and when both $a \prec b$ and $b \prec a$ we say that a and b are* equally preferred[4].

[4]We will note (a_i, a_j) the pairs such that $a_i \prec a_j$.

As we have shown in Section 3.4, there are two kinds of basic operations over preferences in Noos: *preference methods*, that take a set of source elements and an ordering criterion and build a preference (a partially ordered set), and *preference combination methods*, that take two preferences and a combination criterion and build a new preference.

Before to present the formalization of the preference operations we will introduce some preliminary definitions. Then, we will present preference methods, preference combination methods, and finally we show a set of properties about these methods useful for implementing applications in Noos.

Preliminary definitions

Definition 5.19 *(Restriction)*
Given a preference defined by the pair $\langle A, \prec \rangle$ and the set $B \subset A$, we define the preference restricted to the set B as

$$\langle A, \prec \rangle_{|B} = \langle B, \prec \cap \{B \times B\} \rangle$$

Definition 5.20 *(Extension)*
Given a preference defined by the pair $\langle A, \prec \rangle$ and $A \subset B$, we define the preference extension to B as

$$\langle A, \prec \rangle_{extB} = \langle B, \prec \cup \{b \times b\} \rangle$$

Definition 5.21 *(Transitive Closure)*
Given a binary relation $\prec$ defined on a set A we define its transitive closure $\overline{\prec}$ as

$$\overline{\prec} = \{(a_i, a_j) | \exists a_1, \ldots, a_n \in A : a_i \prec a_1 \prec \cdots a_n \prec a_j \}$$

5.10.1 Preference methods

A preference method takes a set of source elements and an ordering criterion and builds a preference. There are several built-in preference methods in Noos. Each preference method implements a different ordering criteria. The complete list of preference methods is provided in Appendix D. Below, we will describe two preference methods as example: `increasing-preference` and `subsumption-preference`.

The `increasing-preference` method can be described as a method that given a feature name f and a set of source elements S builds a preference $\langle S, \prec \rangle$ such that:

$$\prec = \{(s_i, s_j) | s_i.f \doteq v_i, \; s_j.f \doteq v_j, \; v_i > v_j \}$$

The `subsumption-preference` method can be described as a method that given a set of source elements S builds a preference $\langle S, \prec \rangle$ such that:

$$\prec = \{(s_i, s_j) | s_i \sqsubseteq s_j, \; s_j \not\sqsubseteq s_i \}$$

5.10.2 Preference combination methods

A preference combination method takes one or two preferences (created either by preference methods or by other preference combination methods) and builds a new preference combining them in a specific manner. There are five preference combination methods in Noos: `inversion`, `T-intersection`, `C-intersection`, `T-union`, and `C-union`.

The `inversion` method takes a preference $\langle A, \prec\rangle$ and builds a new preference on the same set A inverting the order relations as follows:

Definition 5.22 *(Inversion)*
Given a preference defined by the pair $\langle A, \prec\rangle$ *we define its preference inversion as*

$$\langle A, \prec\rangle^{-1} = \langle A, \{(a_i, a_j) | (a_j, a_i) \in \prec\}\rangle$$

The other four preference combination methods take two preferences $\langle A, \prec_1\rangle$ and $\langle B, \prec_2\rangle$ and build a new preference $\langle S, \prec\rangle$ combining them. Since sets A and B can be different, we will define first two basic combination operations from preferences on the same set of elements. Then, we will explain how combination methods extend these two operations for combining preferences on different sets.

We define two basic operations for combining two preferences $\langle A, \prec_1\rangle$ and $\langle A, \prec_2\rangle$, defined on the same set A. The first one builds a new preference mantaining only the pairs $a \prec b$ common to both $\prec_1$ and $\prec_2$ as follows[5]:

Definition 5.23 *(Basic intersection)*
Given two preferences $\langle A, \prec_1\rangle, \langle A, \prec_2\rangle$ *defined on the same set* A *we define their* intersection preference *as*

$$\langle A, \prec_1\rangle \cap \langle A, \prec_2\rangle = \langle A, \prec_1 \cap \prec_2\rangle$$

The second basic operation builds a new preference gathering all the pairs $a \prec b$ from either $\prec_1$ or $\prec_2$, and performing a transitive closure as follows[6]:

Definition 5.24 *(Basic Transitive Union)*
Given two preferences $\langle A, \prec_1\rangle, \langle A, \prec_2\rangle$ *defined on the same set* A *we define their* transitive union preference *as*

$$\langle A, \prec_1\rangle \uplus \langle A, \prec_2\rangle = \langle A, \overline{\prec_1 \cup \prec_2}\rangle$$

Given two preferences $P1 = \langle A, \prec_1\rangle$ and $P2 = \langle B, \prec_2\rangle$ defined on two different sets A and B, we have considered two alternatives in order to be able to use the basic combination operations. The first one is by extending the preferences $P1$ and $P2$ to the union set $A \cup B$. The second alternative considered is to restrict the preferences $P1$ and $P2$ to the intersection set $A \cap B$ (assuming that $A \cap B \neq \emptyset$). Since we have two alternatives and two basic combination operations, we have developed four combination methods.

[5] Note that the intersection of two pre-orders is also a pre-order.

[6] Transitive closure is performed because of the union of two orders is not always an order.

The `T-intersection` method combines two preferences by restricting preferences to the elements of the intersection and, then, performing the transitive union operation on the resulting preferences as follows:

Definition 5.25 *(T-Intersection)*
Given two preferences $\langle A, \prec_1 \rangle$, $\langle B, \prec_2 \rangle$ *we define their t-intersection* $\overline{\sqcap}$ *using definitions (5.19) and (5.24) as*

$$\langle A, \prec_1 \rangle \overline{\sqcap} \langle B, \prec_2 \rangle = \langle A, \prec_1 \rangle_{|A \cap B} \uplus \langle B, \prec_2 \rangle_{|A \cap B}$$

The `C-intersection` method combines two preferences by restricting preferences to the elements of the intersection and, then, performing the intersection operation on the resulting preferences as follows:

Definition 5.26 *(C-Intersection)*
Given two preferences $\langle A, \prec_1 \rangle$, $\langle B, \prec_2 \rangle$ *we define their c-intersection* $\underline{\sqcap}$ *using definitions (5.19) and (5.23) as*

$$\langle A, \prec_1 \rangle \underline{\sqcap} \langle B, \prec_2 \rangle = \langle A, \prec_1 \rangle_{|A \cap B} \cap \langle B, \prec_2 \rangle_{|A \cap B}$$

The `T-union` method combines two preferences by extending preferences to the elements of the union and, then, performing the transitive union operation on the resulting preferences as follows:

Definition 5.27 *(T-union)*
Given two preferences $\langle A, \prec_1 \rangle$, $\langle B, \prec_2 \rangle$ *we define their t-union* $\overline{\uplus}$ *using definitions (5.20) and (5.24) as*

$$\langle A, \prec_1 \rangle \overline{\uplus} \langle B, \prec_2 \rangle = \langle A, \prec_1 \rangle_{Ext A \cup B} \uplus \langle B, \prec_2 \rangle_{Ext A \cup B}$$

Finally, the `C-union` method combines two preferences by extending preferences to the elements of the union and, then, performing the intersection operation on the resulting preferences as follows:

Definition 5.28 *(C-union)*
Given two preferences $\langle A, \prec_1 \rangle$, $\langle B, \prec_2 \rangle$ *we define their c-union* $\underline{\uplus}$ *using definitions (5.20) and (5.23) as*

$$\langle A, \prec_1 \rangle \underline{\uplus} \langle B, \prec_2 \rangle = \langle A, \prec_1 \rangle_{Ext A \cup B} \cap \langle B, \prec_2 \rangle_{Ext A \cup B}$$

5.10.3 Higher order preferences

Higher order preferences are preference combination methods that build preferences from preferences over preferences. In the current version of Noos we have developed one higher order preference method called `H-union` (hierarchical union). `H-union` takes two preferences $P1$ and $P2$, and constructs a new preference preserving all the set of order relations specified in $P1$ and adding the subset of order relations from $P2$ that are not in conflict with $P1$.

In a similar way to combination methods, we will first define a basic hierarchical union operation from preferences on the same set of elements. Then, we will explain how `H-union` extends the basic operation for combining preferences on different sets.

Definition 5.29 *(Basic Hierarchical Union)*
Given two preferences $\langle A, \prec_1\rangle, \langle A, \prec_2\rangle$ defined on the same set A we define their basic hierarchical union *preference as*

$$\langle A, \prec_1\rangle \bullet \langle A, \prec_2\rangle = \langle A, \overline{\prec_1 \cup \{(a_i, a_j) \in \prec_2 | (a_j, a_i) \notin \overline{\prec_1 \cup \prec_2}\}}\rangle$$

The `H-union` method combines two preferences by extending preferences to the union of sets as follows:

Definition 5.30 *(Hierarchical Union)*
Given two preferences $\langle A, \prec_1\rangle$, $\langle B, \prec_2\rangle$ we define their hierarchical union using definitions (5.20) and (5.29) as

$$\langle A, \prec_1\rangle \bullet \langle B, \prec_2\rangle = \langle A, \prec_1\rangle_{ExtA\cup B} \bullet \langle B, \prec_2\rangle_{ExtA\cup B}$$

5.10.4 Properties

Below we present some properties of preference combination operations that are useful in the development of Noos applications. Particularly, we are interested in properties regarding combinations of preferences—such as commutativity and associativity.

1. The inversion of the inversion of a given preference A yields itself: $(A^{-1})^{-1} = A$
2. Inversion is a morphism with respect to the intersection $(A \cap B)^{-1} = A^{-1} \cap B^{-1}$ and to the union $(A \uplus B)^{-1} = A^{-1} \uplus B^{-1}$.
3. $\uplus$ and $\cap$ operations are associative and commutative.
4. Given two orders A, B (being reflexive, transitive, and antisymmetric) their transitive union $A \uplus B$ is not always an order (may be only a pre-order).
5. Given two orders $\langle A, \prec_1\rangle, \langle A, \prec_2\rangle$, $\langle A, \prec_1\rangle \bullet \langle A, \prec_2\rangle$ has finer granularity than $\langle A, \prec_1\rangle$.
6. Given two orders $\langle A, \prec_1\rangle, \langle A, \prec_2\rangle$ such that $\langle A, \prec_1\rangle$ has finer granularity than $\langle A, \prec_2\rangle$,
$$\langle A, \prec_1\rangle \bullet \langle A, \prec_2\rangle = \langle A, \prec_2\rangle \bullet \langle A, \prec_1\rangle = \langle A, \prec_1\rangle$$
7. Hierarchical union method (5.29) is not commutative.
Example: given $A = \{a, b\}$
$$\langle A, a \prec b\rangle \bullet \langle A, b \prec a\rangle = \langle A, a \prec b\rangle \neq \langle A, b \prec a\rangle = \langle A, b \prec a\rangle \bullet \langle A, a \prec b\rangle$$

8. Hierarchical union method (5.29) is not associative.

   Example: given $A = \{a, b, c\}$

$$(\langle A, a \prec b\rangle \bullet \langle A, b \prec c\rangle) \bullet \langle A, c \prec a\rangle = \langle A, a \prec b \prec c\rangle \bullet \langle A, c \prec a\rangle =$$
$$= \langle A, a \prec b \prec c\rangle \neq \langle A, a \prec b\rangle =$$
$$= \langle A, a \prec b\rangle \bullet \langle A, b \prec c \prec a\rangle = \langle A, a \prec b\rangle \bullet (\langle A, b \prec c\rangle \bullet \langle A, c \prec a\rangle)$$

5.11 Perspectives

Perspectives is a mechanism to describe declarative biases for retrieval in the Noos episodic memory. Given a source problem situation expressed by a feature term, and given a syntactic pattern, the `perspective` method constructs a new feature term that is a partial description of the problem describing the relevant aspects of the problem situation. *Perspectives* are constructed in Noos using the `perspective` built-in method (see Section 4.3).

Formally, the signature of syntactic patterns Σ_{ext} is an extension of the signature Σ of feature terms that incorporates a set of feature variables $\mathcal{L}$ as follows:

$$\Sigma_{ext} = \{\langle \mathcal{S}, \leq_s\rangle, \langle \mathcal{M}, \leq_m\rangle, \mathcal{F} \cup \mathcal{L}\}$$

Using this extension of the signature Σ_{ext} and a set $\mathcal{V}$ of variables, we define syntactic patterns as second order feature terms as follows:

Definition 5.31 *A syntactic pattern ω is an expression of the form:*

$$\omega \quad ::= \quad X : s\,[f_1 \doteq \Omega_1 \cdots f_n \doteq \Omega_n]$$

where X is a variable in $\mathcal{V}$, s is a sort in $\mathcal{S}$, $f_1, \cdots, f_n$ are pairwise features in $\mathcal{F} \cup \mathcal{L}$, $n \geq 0$, and each Ω_i is a set of syntactic patterns and variables.

Given the previous definition of syntactic patterns, we define formally a perspective P as follows,

Definition 5.32 (Perspective)
Given a problem case C and a declarative bias defined by means of a syntactic pattern S, a S-perspective *of C is defined as a feature term P such that there is a total bijective function $\beta : \mathcal{V}_P \to \mathcal{V}_S$, a total mapping function $\delta : \mathcal{V}_S \to \mathcal{V}_C$, and an instantiation function $\rho : \mathcal{F}_S \to \mathcal{F}_P$ satisfying:*

1. $\rho(f) = f \qquad \forall f \in \mathcal{F}$
2. $\beta(Root(P)) = Root(S)$, $\delta(Root(S)) = Root(C)$

and $\forall x \in \mathcal{V}_P$

3. $Sort(\beta(x)) \leq Sort(\delta(\beta(x)))$,
4. $Sort(x) = Sort(\delta(\beta(x)))$,
5. *for every $f_i \in \mathcal{F}$ such that $x.f_i \doteq \Psi_i$ is defined, we have that*

(a) $\exists f_j \in \mathcal{F}_S : \rho(f_j) = f_i$,

(b) both $\beta(x).f_j \doteq \Psi'_i$ *and* $\delta(\beta(x)).f_i \doteq \Psi''_i$ *have to be defined,*

and $\forall \psi_k \in \Psi_i$ *:*

(c) $\exists \psi'_k \in \Psi'_i$ *such that* $\beta(Root(\psi_k)) = Root(\psi'_k)$

(d) $\exists \psi''_k \in \Psi''_i$ *such that* $\delta(\beta(Root(\psi_k))) = Root(\psi''_k)$.

Remark that a perspective P is constructed as a partial description of a problem case C. In other words, this implies that $P \sqsubseteq C$. Another important remark is that several perspectives satisfying the definition can be obtained. This implies that the implementation of the perspective mechanism has to provide a way to obtain all of them (for instance, by providing a backtracking based mechanism).

5.12 Descriptive Dynamic Logic

The goal of Descriptive Dynamic Logic (DDL) [Sierra et al., 1996] is to provide a common logical framework to describe and identify the formal characteristics of Multi-Language Architectures (MLA). In this way, DDL can be understood as a formal basis to describe and compare different multi-language architectures.

In general, a MLA allows to build knowledge systems as a set of units with initial local theories written in possibly different languages. Each unit is also usually allowed to have its own intra-unit deductive system. Moreover, the whole knowledge system is equipped with an additional set of deductive rules, called *bridge rules*, to control the information flow among the different units of the knowledge system. Thus, the DDL approach is very useful to model reflective systems based on the use of several units containing local theories (or meta-theories acting upon theories) that influence and/or modify each other.

In order to model knowledge systems using DDL two levels have to be considered: At the first level the *Multi-Language Logical Architecture* (MLA) is defined as the concept representing the most general characteristics of target systems. In this level languages, inference rules, and allowed topologies are described.

At the second level a particular *knowledge system* is represented. Particular theories are built determining a subset of unit identifiers, the languages and inference rules used in each unit, the set of interconnections among units and their corresponding bridge rules, concrete signatures, and finally initial theories for each unit.

For a full description of DDL see [Sierra et al., 1996]. Here we only present some basic definitions of DDL in order to make easier understanding the Noos formalization.

Definition 5.33 *A Multi-Language Logical Architecture is a 4-tuple* $MLA = (L, \Delta, S, T)$*, where:*

1. $L = \{L_j\}_{j \in J}$ *is a set of logical languages,*

2. $\Delta = \{\Delta_{j_1,j_2}\}_{j_1,j_2 \in J}$ *is a set of (instances of) inference rules between pairs of languages, i.e.* $\Delta_{j_1,j_2} \subseteq 2^{L_{j_1}} \times L_{j_2}$. *In particular, when* $j_1 = j_2$, Δ_{j_1,j_2} *denotes a set of inference rules of the corresponding language; otherwise it denotes a set of bridge rules between two different languages,*

3. S *is a finite set of symbols for unit identifiers,*

4. T *is the set of possible topologies. Each topology is determined by a set of directed links between symbols from* S, *i.e.* T *is a subset of* $2^{S \times S}$.

Notice that DDL is focused only on finite languages as it is the usual case in knowledge systems where some limitative rules are imposed on the generation of formulas.

Definition 5.34 *A Multi-Language Knowledge System MKS for a given MLA is a 7-tuple* $MKS = (MLA, U, M_L, M_\Delta, B, M_\Sigma, M_\Omega)$ *where:*

1. *MLA is a Multi-Language Logical Architecture,*

2. U *is a set of unit identifiers, i.e.* U *is a subset of* S,

3. M_L *assigns a language to each unit identifier, i.e.* $M_L \longrightarrow L$,

4. M_Δ *assigns a set of inference rules to each unit identifier, i.e* $M_\Delta : U \longrightarrow \bigcup_{i \in J} 2^{\Delta_{ii}}$ *such that if* $M_L(u) = L_j$, *for some* $j \in J$, *then* $M_\Delta(u) \subseteq \Delta_{jj}$,

5. B *is a mapping that assigns a set of directed bridge rules to pairs of different units, i.e.* $B : U \times U \longrightarrow \bigcup_{i,j \in J} 2^{\Delta_{ij}}$, *in accordance with the allowed topologies in* MLA,

6. M_Σ *assigns a concrete signature to each unit identifier,*

7. M_Ω *assigns a set of formulas (initial local theory) to each unit identifier, i.e.* $M_\Omega : U \longrightarrow \bigcup_{i \in J} 2^{L_i}$ *such that if* $M_L(u) = L_k$ *then* $M_\Omega(u) \subseteq L_k$.

Definition 5.35 *The set* Φ_0 *of atomic formulas of DDL will be defined as the set of "quoted" formulas from the languages* L *in* MLA, *indexed by the unit identifiers in* U.

$$\Phi_0 = \{u : \lceil \varphi \rceil | u \in U, \varphi \in M_L(u)\}$$

Definition 5.36 *The set* Π_0 *of atomic programs of DDL is defined as the union of intra-unit inference rules* Π_0^{Intra} *and the inter-unit rules* Π_0^{Inter}

$$\Pi_0 = \left(\bigcup_{k \in K} \Pi_{0_{kk}}^{Intra} \right) \cup \left(\bigcup_{k \neq j \in K} \Pi_{0_{kl}}^{Inter} \right)$$

being $\Pi_{0_{kk}}^{Intra}$ *the set of all quoted deduction steps allowed according the unit language* $M_L(u_k)$ *and the inference rules determined by* $M_\Delta(u_k)$

and $\Pi_{0_{kl}}^{Inter}$ *the set of all quoted deduction steps allowed according the unit languages* $M_L(u_k)$ *and* $M_L(u_l)$ *and the inference rules determined by* $B(u_k, u_l)$*.*

Given the set Φ_0 of atomic formulas and the set Π_0 of atomic programs, the set Φ of compound formulas and the set Π of compound programs is constructed following the propositional dynamic logic composition rules defined as:

1. $true \in \Phi$, $false \in \Phi$, $\Phi_0 \subseteq \Phi$,

2. if $\varphi, \psi \in \Phi$ then $\neg\varphi \in \Phi$ and $(\varphi \vee \psi) \in \Phi$,

3. if $\varphi \in \Phi$ and $\alpha \in \Pi$ then $\langle\alpha\rangle\varphi \in \Phi$, denoting the possibility that after the execution of α the formula φ to be true.

4. $\Pi_0 \subseteq \Pi$,

5. if $\alpha, \beta \in \Pi$ then the sequential concatenation $(\alpha; \beta) \in \Pi$,

6. if $\alpha, \beta \in \Pi$ then the indeterministic union $(\alpha \cup \beta) \in \Pi$,

7. if $\alpha \in \Pi$ then the self-iteration $\alpha^* \in \Pi$,

8. if $\varphi \in \Phi$ then $\varphi? \in \Phi$, denoting the program that evaluates whether a given formula φ is true.

$[\alpha]\varphi$ is the usual modal abbreviation for $\neg\langle\alpha\rangle\neg\varphi$. Also $\wedge$, $\rightarrow$ and $\leftrightarrow$ are abbreviations with the standard meaning.

Definition 5.37 *The DDL semantics, following the PDL semantics, is defined relative to a structure* M *of the form* $M = (W, \tau, \rho)$*, where* W *is a set of states,* τ *a mapping* $\tau : \Phi \rightarrow 2^W$ *assigning to each formula* φ *the set of states in which* φ *is true, and* ρ *a mapping* $\rho : \Pi \rightarrow 2^{W \times W}$ *which assigns to each program a set of pairs* (s, t) *representing transitions between states.*

After introducing the basic concepts of DDL, we are ready to formalize the metalevel inference in Noos.

5.13 Modeling Noos inference using DDL

In order to model Noos inference in DDL we have to define the set of unit languages, the set of inta-unit and inter-unit inference rules, the possible topologies, and finally, the inference process as a set of compound programs;

5.13.1 Noos unit languages

Formally, every feature term is represented as a DDL unit. Every unit has a different identifier. There are three kinds of unit languages: *concept languages*, *method languages* and *metalevel languages*. Concept languages pertain to units representing concepts (called concept units). Method languages pertain to units representing methods (called method units) and are an extension of concept languages. Metalevel languages pertain to units representing metalevels (called metalevel units) and are also an extension of concept languages.

Unit languages are built from the set of feature names $\mathcal{F}$ and the set of unit identifiers U. We note U_m the subset of method unit identifiers from U.

Before presenting Noos unit languages we will define a notational equivalence that simplifies their definition:

Definition 5.38 *A feature value c is considered equivalent to the singleton set that contains as element this feature value c.*

$$c \equiv \{c\}$$

Given the definition above, we describe feature values and the set of inference rules for query-methods directly working on sets.

The language L_c of a concept unit c representing a feature term ψ contains formulas describing the feature values pertaining to each feature of ψ and formulas describing the method pertaining to each feature of ψ. Since we have shown in Section 3.3.6 that path references can be also viewed as query-methods, we will represent all the path references as query-methods in order to simplify the DDL model of Noos. Specifically, the set of formulas Φ_c of a given concept unit c is described as:

$$\begin{array}{ll} f \in \mathcal{F}, u \in 2^U, m \in U_m : & f \doteq u \in \Phi_c \\ & f \doteq m\# \in \Phi_c \end{array}$$

The language L_μ of a metalevel unit μ is a concept unit language extended with formulas describing the set of formulas about features contained in its referent unit. Thus, the set of formulas Φ_μ of a metalevel unit μ contains, in addition, formulas describing for each feature in the referent unit the method and value (referent) pertaining to the feature as follows:

$$f \in \mathcal{F}; m \in U_m : method(f) \doteq m \in \Phi_\mu$$

$$f \in \mathcal{F}, u \in 2^U : referent(f) \doteq u \in \Phi_\mu$$

The language L_m of a method unit m contains a set of formulas Φ_m describing feature values and feature methods like concept units. Moreover, Φ_m contains a set of formulas describing the result of the method evaluation. These formulas are described as follows:

$$u \in 2^U : result(m) \doteq u \in \Phi_m$$

As we have shown, query-methods are a special kind of methods that provide metalevel capabilities of reasoning about feature values. Query-methods are also represented as DDL units. Their languages are method languages enriched with formulas containing the feature values of features in other units. Thus, the set of formulas Φ_m of a given query-method unit m is enriched by

$$f \in \mathcal{F}, c_i \in U, u \in 2^U : c_i.f \doteq u \in \Phi_m$$

Another special kind of methods are eval-methods. Their languages are method languages enriched with formulas containing the evaluation results of other methods.

$$m_i \in U_m, u \in 2^U : result(m_i) \doteq u$$

5.13.2 Inference rules

The elementary inference inference steps in DDL are represented as a collection of inference rules. There are two kinds of inference rules: *intra-unit* inference rules for modeling the inference within a unit, and *inter-unit* inference rules for modeling the communication among the different units. Only methods and meta-level units have intra-unit inference rules. Inter-unit inference rules may connect a unit with any other unit following the Noos topology (see Section 5.13.3).

Intra-Unit Inference Rules

Only methods and metalevel units have intra-unit inference rules. Inference rules in metalevel units select one method for a given feature f from a set of alternative methods S; reflection rules (represented as inter-unit rules) will add this selected method to its referent unit. The inference rule for method selection is the following:

$$\delta_f^{select} = \frac{\begin{array}{c} f \doteq S \\ m \in S \end{array}}{method(f) \doteq m}$$

Inference rules in methods code the built-in definition of the evaluation of the method. There is one inference rule for each type of built-in method provided by Noos. We have only to translate the previously defined λN lambda abstractions to the DDL syntax. For instance, the inference rule for a `subtract` method m previously defined as

$$subtract_{def} = \lambda(amount\ minus)\ amount - minus$$

is translated to

$$\delta_m^{subs} = \frac{\begin{array}{c} amount \doteq c_1 \\ minus \doteq c_2 \\ \text{“}c' = c_1 - c_2\text{”} \end{array}}{result(m) \doteq c'}$$

where a new formula $result(m) \doteq c'$ is added with the result of the difference between the feature values given in *amount* and *minus*.

A more interesting inference rule is the inference rule for query-methods that allows to reason about feature values of other units. The inference rule δ_m^{iv} for a query-method unit m is defined as

$$\delta_m^{iv} = \frac{\begin{array}{c} feature \doteq f \\ domain \doteq \{c_1 \cdots c_n\} \\ c_1.f \doteq s_1 \\ \cdots \\ c_n.f \doteq s_n \end{array}}{result(m) \doteq \cup s_i}$$

Another important inference rule is the rule for eval-methods that allows to reason about method units. The inference rule δ_m^{ne} for an Eval-method unit m is defined as

$$\delta_m^{ne} = \frac{\begin{array}{c} methods \doteq \{m_1 \cdots m_n\} \\ result(m_1) \doteq s_1 \\ \cdots \\ result(m_n) \doteq s_n \end{array}}{result(m) \doteq \cup s_i}$$

Inter-Unit Inference Rules

There are four kinds of inter-unit inference rules: *Reification rules*, *Reflection rules*, *Reduction rules*, and *Translation rules*. *Reification rules* specify the representation that a metalevel unit has about its corresponding base-level unit. *Reflection rules* specify the changes that a metalevel unit may perform upon its corresponding base-level unit. *Reduction rules* add to a unit the result of the evaluation of one of its methods. Finally, *Translation rules* specify how formulas may be transported from a unit to another one. The collection of inter-unit inference rules are determined by the allowed topologies in Noos (see Section 5.13.3).

Reification rules $\delta_{c\mu}^{up}$ add to the metalevel unit μ the set formulas about the feature values known in the unit c

$$\delta_{c\mu}^{up} = \frac{f \doteq u}{referent(f) \doteq u}$$

Reflection rules $\delta_{\mu cf}^{down}$ add to the base-level unit c a formula about the feature method selected by the metalevel unit μ

$$\delta^{down}_{\mu c f} = \frac{method(f) \doteq m}{f \doteq \#m}$$

Reduction rules δ^{red}_{mcf} add to a unit c the formula for a feature f with the result of the evaluation of one method unit m.

$$\delta^{red}_{mcf} = \frac{result(m) \doteq u}{f \doteq u}$$

Translation rules δ^{tquery}_{cmf} add from a unit c to a query-method unit m the formula for a feature f

$$\delta^{tquery}_{cmf} = \frac{f \doteq u}{c.f \doteq u}$$

Translation rules $\delta^{teval}_{m_1 m_2}$ add from a method unit m_1 to a method unit m_2 a formula with the evaluation result of m_1

$$\delta^{teval}_{m_1 m_2} = \frac{result(m_1) \doteq u}{result(m_1) \doteq u}$$

5.13.3 Topology

The set of possible topologies in Noos is formed by three kinds of relations among units: *reference relations*, *feature method relations* and *metalevel relations*. The set of reference relations of a unit c with other units c' is determined by the set of formulas $f \doteq c'$ contained in c. The set of feature method relations of a unit c with other method units m is determined by the set of formulas $f \doteq m\#$ contained in c. Metalevel relations are determined by explicit `meta` relations from Noos descriptions. Metalevel relations are exclusive relations: one unit can be the metalevel of only another unit (called *referent*), one unit can only have one metalevel unit, and cycles are forbidden. Note that a metalevel unit can be the referent of another (meta)metalevel unit. Figure 5.5 shows an example of a specific Noos topology.

5.13.4 Programs

Inference in Noos is modelled by means of four kinds of programs: *task programs* formalizing the inference of feature values, *metalevel programs* formalizing the metalevel inference, *query programs* formalizing the inference performed by query methods, and *eval programs* formalizing the inference performed by eval-methods.

The formalization of the Noos inference is presented without taking into account preferences. Then, we will extend the formalization for incorporating preferences.

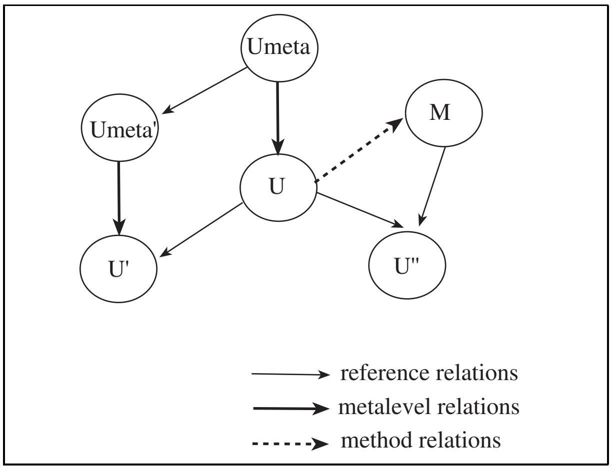


Figure 5.5. An example of an allowed topology.

Task programs

Every unit c has a set Π_c^{Intra} of task programs $\pi_{c.f}$ for features f defined in c. Specifically,

$$\Pi_c^{Intra} = \bigcup_{f \in \mathcal{F}} \pi_{c.f}$$

where $\pi_{c.f}$ is defined as follows

$$\pi_{c.f} = (\pi^{\mu}_{c.f} \cup true?); \bigcup_{m \in U_m} \left((f \doteq m\#)?; \pi_m; \delta^{red}_{mcf}\right)$$

The task program $\pi_{c.f}$ for a feature f of unit c is defined as the sequential concatenation of three programs: (i) the metalevel inference program $\pi^{\mu}_{c.f}$; (ii) the evaluation program π_m of the method m; and (iii) the inference rule δ^{red}_{mcf} that adds a new formula with the result of the method evaluation. The indeterministic union $\pi^{\mu}_{c.f} \cup true?$ expresses the possibility to skip the metalevel inference step when there is a method defined in the unit c.

Metalevel inference programs

Metalevel inference $\pi^{\mu}_{c.f}$ for a feature f of a unit c is defined as the following sequential concatenation:

$$\pi^{\mu}_{c.f} = \delta^{up}_c; \pi_{\mu.f}; \delta^{select}_f; \delta^{down}_{\mu cf}$$

The metalevel inference program $\pi^{\mu}_{c.f}$ starts with a reification inference rule δ^{up}_c, then engages the task program $\pi_{\mu.f}$ for a feature f at the metalevel unit μ, selects one of the methods obtained in the previous step (δ_f^{select}), and reflects down this selected method to the referent unit ($\delta_{\mu c f}^{down}$).

Evaluation programs

Evaluation of methods in Noos are also formalized as DDL programs having the following scheme:

$$\pi_m = \pi_{m.f_1}; \cdots; \pi_{m.f_n}; \delta_m$$

The evaluation program π_m of a method m is composed by the sequence of task programs to infer the values for their required subtasks $(f_1, \cdots, f_n)$ followed by the intra-unit inference rule of m that combines the values of subtasks into a final value. For instance, the evaluation program π_m^{sub} of a subtraction method m is the sequence of task programs to compute the operands and the intra-unit rule δ_m^{sub} that combines them.

$$\pi_m^{sub} = \pi_{m.amount}; \pi_{m.minus}; \delta_m^{sub}$$

Query-methods

Query-methods have also their own evaluation programs. The evaluation program of a query-method is the reification of inference in Noos. A query-method m involves the subtask *feature* (for obtaining a feature name f); the subtask *domain* (for obtaining a unit or a set of units s); and the tasks of inferring the feature value of feature f of all units in s. We use s as a shorthand of $\{c_1 \cdots c_n\}$.

The first query-method is *Infer-value* method. The compound program π_m^{iv} for an infer-value method is the following:

$$\pi_m^{iv} = \pi_{m.feature}; \pi_{m.domain}; \bigcup_{s,f \in L_m} (R_m^{iv}; \delta_m^{iv})$$

where

$$R_m^{iv} = ((\mathit{feature} \doteq f) \wedge (\mathit{domain} \doteq \{c_1 \cdots c_n\}))?; \pi_{c_1.f}; \delta_{c_1 m f}^{tquery}; \cdots; \pi_{c_n.f}; \delta_{c_n m f}^{tquery}$$

The evaluation program π_m^{iv} of an *Infer-value* method m first engages the computation of a feature name f ($\pi_{m.feature}$), next computes a unit or set of units s ($\pi_{m.domain}$), and finally the task f is performed to all units in s (R_m^{iv}) and the results are combined by δ_m^{iv}.

The evaluation program π_m^{ev} of an *Exists-value* method m determines if any solution to a task f for a set of units in s exists, and returns a boolean value accordingly:

$$\pi_m^{ev} = \underline{\text{if}} \langle \pi_m^{iv} \rangle ? \ \underline{\text{then}} \ \delta_m^{true} \ \underline{\text{else}} \ \delta_m^{false}$$

where δ_m^{true} and δ_m^{false} are just intra-unit inference rules assigning *true* and *false* respectively.

The evaluation program π_m^{kv} of a *Known-value* method m determines if the solution to a task f for a set of units in s have already been computed, and returns a boolean value accordingly:

$$\pi_m^{kv} = \pi_{m.feature}; \pi_{m.domain}; \bigcup_{s,f \in L_m} R_m^{kv}$$

where

$$\begin{aligned} R_m^{kv} &= ((\mathit{feature} \doteq f) \wedge (\mathit{domain} \doteq \{c_1 \cdots c_n\}))?; \\ & \quad \underline{\text{if}}\ (\delta_{mc_1 f}^{tquery}; \cdots; \delta_{mc_n f}^{tquery})?\ \underline{\text{then}}\ \delta_m^{true}\ \underline{\text{else}}\ \delta_m^{false} \end{aligned}$$

Notice that the R_m^{kv} program is composed only by translation rules. Thus, the evaluation of *Known-value* methods will yield *true* only when all feature values have been already inferred.

Finally, evaluation program π_m^{av} of an *All-values* method m determines the set of all inferrable values to a task f for a set of units in s:

$$\pi_m^{av} = \pi_{m.feature}; \pi_{m.domain}; \bigcup_{s,f \in L_m} (R_m^{av}; \delta_m^{av})$$

where

$$\begin{aligned} R_m^{av} &= ((\mathit{feature} \doteq f) \wedge (\mathit{domain} \doteq \{c_1 \cdots c_n\}))?; \\ & \quad (\pi_{c_1.f}; \delta_{mc_1 f}^{tquery})^c; \cdots; (\pi_{c_n.f}; \delta_{mc_n f}^{tquery})^c \end{aligned}$$

and where α^c represents the closure of program α—that is, this program will lead to a state in which no different state is reachable by another application of program α. Specifically, a closure $(\pi_{c_i.f}; \delta_{mc_i f}^{tquery})^c$ produces all possible values of task program $\pi_{c_i.f}$. Next δ_m^{av} rule puts together all the values.

Query methods deal with the methods of a specific task and determine which of the four kinds of inference is engaged by that task. In order to deal with a specific method unit, Noos uses four kinds of eval-methods.

Eval-methods

The evaluation program of an eval-method m involves the subtask *methods* that engages in the computation of the methods to be evaluated. Next, the evaluation programs of these methods are performed.

The first eval-method is *Noos-eval*. The evaluation program π_m^{ne} of a *Noos-eval* method m is the following:

$$\pi_m^{ne} = \pi_{m.methods}; \bigcup_{s \in L_m} (R_m^{ne}; \delta_m^{ne})$$

where

$$R_m^{ne} = (methods \doteq \{m_1 \cdots m_n\})?; \pi_{m_1}; \delta_{m_1 m}^{ne}; \cdots; \pi_{m_n}; \delta_{m_n m}^{ne}$$

The evaluation program π_m^{ne} of an eval-method m first engages the computation of the methods to be evaluated ($\pi_{m.methods}$), next performs the evaluation programs of these methods (R_m^{ne}), and finally the results are combined (δ_m^{ne}).

The evaluation program of an *Exists-eval* method m checks whether any solution to evaluation exists, and returns a boolean accordingly:

$$\pi_m^{ee} = \underline{\text{if}}\langle\pi_m^{ne}\rangle?\ \underline{\text{then}}\ \delta_m^{true}\ \underline{\text{else}}\ \delta_m^{false}$$

The evaluation program of a *Known-eval* method m is analogous to the previous evaluation program for determining that methods have already been evaluated:

$$\pi_m^{ke} = \pi_{m.methods}; \bigcup_{s \in L_m} R_m^{ke}$$

where

$$R_m^{ke} = (methods \doteq \{m_1 \cdots m_n\})?; \underline{\text{if}}\ (\delta_{m_1 m}^{ne}; \cdots; \delta_{m_n m}^{ne})?\ \underline{\text{then}}\ \delta_m^{true}\ \underline{\text{else}}\ \delta_m^{false}$$

Finally, the evaluation program of a *All-eval* method m infers all the possible values:

$$\pi_m^{ae} = \pi_{m.methods}; \bigcup_{s \in L_m} (R_m^{ae}; \delta_m^{ae})$$

where

$$R_m^{ae} = (methods \doteq \{m_1 \cdots m_n\})?; (\pi_{m_1}; \delta_{m_1 m}^{ne})^c; \cdots; (\pi_{m_n}; \delta_{m_n m}^{ne})^c$$

and δ_m^{ae} rule puts together all the values.

Inference in Noos starts when the user poses a query to the system by means of a *query expression*. There are two kinds of query expressions: *path references* and *eval expressions*. A path reference `(>> f of d)` will start the task program $\pi_{d.f}$. An eval expression `(noos-eval m)` will start the eval program π_m.

5.13.5 Adding preferences

Several alternative solutions can be yielded in solving a problem task `T`, since several methods can be defined for solving a subtask of `T`. The DDL-based formalism for modeling Noos inference presented in previous section yields a solution for a problem task `T` in an indeterministic way. Preferences are used in Noos for specifying a preference order on the set of alternative methods defined for solving a task. This preference order between methods implicitly define a preference order on the values inferred by methods. Thus, preferences define a preference order on solution values for a problem task T.

Preferences can be added to the previous presented formalism as a postcondition constraining the results yielded for a problem task. Specifically, given a problem task $f(d)$ we can define the task program $\pi^*_{d.f}$ taking into account the preference information as the following contatenation:

$$\pi^*_{d.f} = \pi_{d.f}; (Maximal\_Solution)?$$

where first $\pi_{d.f}$ engages the indeterministic task program for inferring a solution value; and $Maximal\_Solution$ is a conjunction of DDL formulas that ensures that the solution is maximal with respect to preferences.

The DDL expression for $Maximal\_Solution$ is a conjunction of formulas satisfying the following definition:

Definition 5.39 *(Maximal solution)*
Given the set of achieved subtasks $t_1, t_2 \cdots t_n$, that form the task decomposition of a problem task T, given the set of partial orders $\prec_1, \cdots \prec_n$ over the alternative methods for these subtasks, and given the set of methods $m_1, m_2 \cdots m_n$ engaged respectively to these subtasks, a solution of T is maximal *if there is no other combination of methods $m'_1 \prec_1 m_1$, $m'_2 \prec_2 m_2, \cdots m'_n \prec_n m_n$ (where at least one $m'_i \neq m_i$) that achieves a solution for T.*

5.14 Summary

This chapter presented the formal description of the Noos language. We presented the Noos formal syntax based on feature terms, its semantics, and the formal model of the Noos inference process.

We used λN calculus to provide a syntax for Noos feature terms. Moreover, λN calculus capabilities for modeling extensible knowledge are used for modeling the refinemement mechanism of Noos. λN calculus lexical scoping is used for modeling path references and path equality.

We adopted a related approach to the semantical interpretation of ψ-terms in order to provide a semantical interpretation of Noos feature terms. Following the *ψ-term* formalism, feature terms are interpreted as partial descriptions. This semantical interpretation of feature terms brings an ordering relation among them. We call this ordering relation *subsumption*.

ψ-terms allows three equivalent representations: terms, clauses, and graphs. We presented a graph representation of feature terms. The graph representation is the basis for developing graphical browsers for the Noos development environment (see Appendix A). We also presented the clausal representation. clausal representation is useful for comparing Noos learning methods with other existing learning methods.

We follow the work on *ψ-terms* for providing a formalism for Noos. Nevertheless, other formalisms such as description logics [Nebel, 1990] (also known as terminological logics) are also close to ψ-*terms*. Examples of languages based on description logics are LOOM [MacGregor, 1991] and CLASSIC [Brachman et al., 1991].

We used Descriptive Dynamic Logic to describe the inference process involved in solving a specific problem task. The DDL model of Noos is defined by means of a collection of units with three kinds of unit languages: concept languages, method languages and metalevel languages. Every feature term is represented as a DDL unit. The elementary inference inference steps in Noos are represented as a collection of inference rules. There are two kinds of inference rules: intra-unit inference rules for modeling the inference within a unit, and inter-unit inference rules for modeling the communication among the different units. Only methods and metalevel units have intra-unit inference rules. Inter-unit inference rules may connect a unit with any other unit following the Noos topology. Then, combining the inference rules, inference in Noos is modeled by means of four kinds of programs: task programs formalizing the inference of feature values, metalevel programs formalizing the metalevel inference, query programs formalizing the inference performed by query methods, and eval programs formalizing the inference performed by eval-methods.

Finally, we defined formally two specific elements of Noos: preferences and perspectives. Preferences are a declarative mechanism for decision making about sets of alternatives. Reasoning with preferences is modeled by partially ordered sets with a set of operations for constructing new preferences and combining them.

Perspectives are a mechanism to describe declarative biases for retrieval in the Noos episodic memory. Using feature terms, perspectives are formalized as second order feature terms that denote sets of terms.