4 Record Constraints

In addition to finite domain constraints, DFKI Oz supports fine-grained constraints on records, called *record constraints* or *feature constraints*. A record in Oz is a rational tree with labeled edges and nodes. The edges are called *features* (they correspond to record fields) and the nodes are called *labels*. The basic idea is that one may calculate with a record that is partially known. Specifically, records may be constrained to have a given label and given features, without the record being determined (i.e., label and number of fields are known). As more is known about the record, more features can be added. An undetermined record is also called an "(open) feature structure".

Record constraints provide a simple logical foundation for feature structures as used in computational linguistics. Feature structures have been shown to be useful for natural language and knowledge representation tasks. An influential variety of feature structures called "psi-terms" was developed by Hassan Aït-Kaci [2] and used in the LIFE language [3]. The implementation of feature structures in Oz is based on foundational work on records for logic programming [15] that is inspired by the work on psi-terms. Psi-terms differ from Oz feature structures in important ways, but both have in common the explicit representation of partial information about records.

There are four primitive record constraints:

1. `X` is a record.
2. `X` has label `L`.
3. `X` has feature `F` with value `Y`.
4. `X` has a given set of features (i.e., *arity*), and no others.

DFKI Oz provides built-in procedures to support these constraints. The built-in solver is complete for these constraints when the label and features are known.

The following example shows how feature structures are used in Oz:

```
% X has feature ´age´ with value 25
X^age=25

% X has feature ´name´ with value "tom"
X^name="tom"
```

```
% X has label 'person'
{LabelC X person}
```

This constrains `X` to a record containing at least the label `person`, the feature `age` with value `25`, and the feature `name` with value `"tom"`. The Oz syntax for this partially known record is `person(age:25 name:"tom" ...)`. The above example could have been entered as:

```
X=person(age:25 name:"tom" ...)
```

with identical meaning.

An undetermined record becomes determined when it is unified with a determined record, for example:

```
X=person(age:_ name:_ sex:m)
```

`X` is now known to have the features `age`, `name`, and `sex`, and no others.

4.1 The Module `RecordC`

The module `RecordC` contains the predefined procedures for programming with feature structures. This section presents the procedures provided by `RecordC` and their semantics.

A variable `X` is said to be *determined* when its top level is known. There are three cases: `X` is determined to a constant, to a tuple, or to a record. The last case is relevant here: a variable is determined to a record if its label and its arity are determined, i.e., when the label is a literal (an atom or name) and all the features are known.

All procedures requiring an input record that are not described in this section will wait until the record is determined before proceeding. See the description of the module `Record` in the Oz Standard Modules.

The procedures in this section are efficient; all simple operations (such as `X^F=V`) execute in amortized constant-time and all others are quasi-linear (the same complexity as unification). The speed of operations on determined records is not affected by the existence of undetermined records.

`X^F=V`

Constrain `X` to a record; also wait until `F` is determined and then constrain `X` to have subtree `V` at feature `F`. The feature `F` must be a literal.

`{LabelC X L}`

Constrain `X` to a record; also wait until `L` is determined or `X` has a determined label and then constrain `X`'s label and `L` to be equal. The label must be a literal.

`{WidthC X W}`

Constrain `X` to a record and `W` to a finite domain. Also constrain the number of features of `X` to be equal to `W`. This built-in is a propagator and will disappear when `X` is determined.

`X=L(A:T1 B:T2 ...)`

Abbreviates `{LabelC X L} X^A=T1 X^B=T2`. The three dots "..." are part of the syntax. At least one feature must be given.

`X=L(A:T1 B:T2)`

Constrain `X` to a record with label `L` and features `A` and `B`, with values `T1` and `T2`. Constrain `X` to have no other features.

`{IsRecordC X}`
`{IsRecordCB X ?B}`

Constrain `X` to a record. Nothing else is implied about `X`, neither the label nor the number of features.

`{Show X}`

Show the constraints on `X` in the constraint store. If `X` is an undetermined record, the atomic features of `X` are shown in alphabetical order. The name features of `X` are shown in arbitrary order after the atomic features.

4.2 A Constrained Sort Hierarchy

This section shows how one can quickly and efficiently encode a constrained sort hierarchy, such as exists in the LIFE language [4, 3], into DFKI Oz. To encode the hierarchy we use both record constraints and finite domain constraints. The resulting terms behave exactly like LIFE's psi-terms.

4.2.1 Encoding a Sort Hierarchy

Each sort `s` is encoded as a record. The hierarchy is encoded by giving the record a field that contains a finite domain value representing the sort's place in the hierarchy. If `s` is a child of `t`, then `s`'s value is a subset of `t`'s value. For example, `s` can be encoded with the domain `[1]` and `t` with the domain `[1 2]`.

A simple algorithm to encode a complete hierarchy is to start by giving each sort a domain consisting of a unique integer, and then to propagate the integers upwards in the hierarchy, including each integer in the domains of all higher sorts.

For example, take the following hierarchy:

```
student <| person.
employee <| person.
staff <| employee.
faculty <| employee.
workstudy <| student.
workstudy <| staff.
```

The notation `student <| person` is from LIFE and means that `student` inherits from `person`. The hierarchy is encoded as:

```
fun {PersonCode}    {FD.dom [1 2 3 4 5 6]} end
fun {StudentCode}   {FD.dom [1       5  ]} end
fun {EmployeeCode}  {FD.dom [1 2 3 4    ]} end
fun {StaffCode}     {FD.dom [1   3      ]} end
fun {FacultyCode}   {FD.dom [  2        ]} end
fun {WorkstudyCode} {FD.dom [1          ]} end
```

The top element in this hierarchy is `person`, and both `faculty` and `workstudy` are leaf elements. The important property of the encoding is that finding the greatest lower bound of two sorts can be done by unifying their codes.

4.2.2 Constraining the Sorts

The next step is to constrain the sorts. This can be done using a conditional. For example, let's assume that the sorts `student` and `faculty` are constrained:

```
:: student(advisor => faculty, roommate => person).
:: faculty(secretary => staff, assistant => person).
```

The notation `:: student(advisor => faculty)` is from LIFE and means that every term with sort `student` has a field `advisor` whose value is a term of sort `faculty`. To add this constraint at the moment a sort inherits from `student`, we use a conditional equality test:[10]

```
proc {StudentDemon X}
   if X^code={StudentCode} then
      X=sort(advisor:{NewFaculty} roommate:{NewPerson} ...)
   else true fi
end

proc {FacultyDemon X}
   if X^code={FacultyCode} then
      X=sort(secretary:{NewStaff} assistant:{NewPerson} ...)
   else true fi
end
```

[10] This works because equality of finite domain variables takes the whole domain into account, not just the bounds.

```
proc {Demons X}
   {FacultyDemon X}
   {StudentDemon X}
end
```

Calling `{StudentDemon X}` will do nothing until `X` inherits from `student`. To create a term, it must be given the right code and `Demons` must be called:

```
proc {NewPerson X}    X=sort(code:{PersonCode} ...)    {Demons X} end
proc {NewStudent X}   X=sort(code:{StudentCode} ...)   {Demons X} end
proc {NewEmployee X}  X=sort(code:{EmployeeCode} ...)  {Demons X} end
proc {NewStaff X}     X=sort(code:{StaffCode} ...)     {Demons X} end
proc {NewFaculty X}   X=sort(code:{FacultyCode} ...)   {Demons X} end
proc {NewWorkstudy X} X=sort(code:{WorkstudyCode} ...) {Demons X} end
```

4.2.3 An Example Query

We create a new employee and a new student with:

```
E={NewEmployee}
S={NewStudent}
```

This results in two constrained sorts:

```
E = sort(code: _ {1..4} ...)
S = sort(code: _ {1 5}
         advisor: sort(code: 2
                       assistant: sort(code: _ {1..6} ...)
                       secretary: sort(code: _ {1 3} ...) ...)
         roommate: sort(code: _ {1..6} ...) ...)
```

Constraining these to be equal with `E=S` results in:

```
E = sort(code: 1
         advisor: sort(code: 2
                       assistant: sort(code: _ {1..6} ...)
                       secretary: sort(code: _ {1 3} ...) ...)
         roommate: sort(code: _ {1..6} ...) ...)
```

The result has sort `workstudy`.

4.3 A Head-Driven Phrase Structure Grammar (HPSG)

This section presents in detail an HPSG parser written using record constraints. This gives a nontrivial example of how record constraints may be used for natural-language parsing. The parse tree is defined with record constraints and the parser is executed with encapsulated search.

The HPSG parser is presented here in three parts. The first part define the grammar by means of constraints. The second part gives the lexicon. The third part defines a simple bottom-up parser. We present and explain each part separately. The complete program consists of all program fragments concatenated together, and can be found in the demo file `grammar-hpsg.oz` provided with the system.

The parse tree consists of leaves (i.e., external nodes) `W` and internal nodes `P`, with the following structure:

```
W = w(phon:_ cat:_ subcat:_)
P = p(phon:_ cat:_ subcat:_ headDtr:_ compDtr:_)
```

A word `W` has a phonology `phon` (a list of input words), a category `cat` (e.g., `noun`, `adjective`, `determiner`, or `verb`), and a subcategory `subcat` (a list of categories). An internal node `P` has in addition two children: the head daughter `headDtr` and the complement daughter `compDtr`, which both refer to other nodes.

4.3.1 Principles of the Grammar

The parse tree is characterized, in HPSG terminology, by giving the *principles* on which it is based. The principles define the grammar by giving the properties that any correct parse tree must satisfy. The principles are given as record constraints, which are defined *locally* between each internal node and its two daughters. `{Phrase P}` constrains the internal node `P` to satisfy the three principles that define our grammar:

```
proc {Phrase P}
   H C
in
   P = p(cat:_ subcat:_ phon:_ headDtr:H compDtr:C)

   % 1. subcat principle
   H^subcat = C^cat | P^subcat

   % 2. saturated complements principle
   C^subcat = nil

   % 3. head-feature principle
   P^cat = H^cat
end
```

The subcat principle states how `P`'s subcategory relates to that of its daughters (`H^subcat=C^cat|P^subcat`). The saturated complements principle states that the complement daughter must have an empty subcategory (`C^subcat=nil`). The head-feature principle states that a node inherits its category from its head daughter (`P^cat=H^cat`).

In addition to these principles, we need to express the nondeterminism that a node may be built in two different ways from its two children. This is expressed by the procedure `{Rules P L R}`:

```
proc {Rules P L R}
   {Phrase P}
   P^phon = {Append L^phon R^phon}
   or P^subcat = nil
      P^headDtr = R
      P^compDtr = L
   [] P^subcat = [_]
      P^headDtr = L
      P^compDtr = R
   ro
end
```

The parser will of course choose an order that satisfies the constraints.

4.3.2 The Lexicon

The lexicon is defined by the relation `{Word W}`, which behaves as a database of facts. A word `W` is a leaf of the parse tree (not to be confused with an *input* word, which is raw input to the parser).

```
proc {Word W}
   P C S
in
   W = w(phon:[P] cat:C subcat:S)
   or P=mary          C=noun        S=nil
   [] P=john          C=noun        S=nil
   [] P=girl          C=noun        S=[determiner]
   [] P=nice          C=adjective   S=nil
   [] P=pretty        C=adjective   S=nil
   [] P=the           C=determiner  S=nil
   [] P=laughs        C=verb        S=[noun]
   [] P=meets         C=verb        S=[noun noun]
   [] P=kisses        C=verb        S=[noun noun]
   [] P=embarrasses   C=verb        S=[noun noun]
   [] P=thinks        C=verb        S=[verb noun]
   [] P=is            C=verb        S=[adjective noun]
   [] P=met           C=adjective   S=nil
   [] P=kissed        C=adjective   S=nil
   [] P=embarrassed   C=adjective   S=nil
   ro
end
```

4.3.3 The Bottom-up Parser

The bottom-up parser is defined by the function `{Parse Phon}`, which takes a phonology and returns a parse tree:

```
fun {Parse Phon}
   {Parse1 {Map Phon proc {$ W F} F^phon=[W] {Word F} end}}
end

fun {Parse1 Fs}
   case Fs of [F] then F
   else {Parse1 {Move Fs}} end
end

proc {Move Fs ?Gs}
   case Fs of F|(G|Fr=T) then
      or X in {Rules X F G}  Gs=X|Fr
      [] Gs = F|{Move T}
      ro
   else false end
end
```

`Parse` first converts the phonology into a list of leaves using `Map`. For example, the phonology `[the girl]` is converted into the list:

```
[w(phon: [the] cat: determiner subcat: nil)
 w(phon: [girl] cat: noun subcat: [determiner])]
```

The elements in this list are the leaves of the parse tree (i.e., the *words*). The parser scans this list and tries to combine two adjacent elements with `Rules` (see `Move`). It repeats this action, creating a new list with one less element in each scan, until only one element is left (see `Parse1`). This element is the parse tree.

4.3.4 An Example Query

The procedure `SParse` calls the parser using encapsulated search:

```
proc {SParse Phon}
   {Search query(fun {$} {Parse Phon} end)}
   {Search next}
end
```

By default, this will search for the first depth-first enumeration of the disjunctions that satisfies all constraints. Entering the query:

```
{SParse [the girl]}
```

will display the following parse tree in the browser window:

```
p(cat: noun
  compDtr: w(cat: determiner phon: [the] subcat: nil)
  headDtr: w(cat: noun
             phon: [girl]
             subcat: [determiner])
  phon: [the girl]
  subcat: nil)
```

Additional valid parses (if any exist) may be obtained with the query `{Search next}`.