Oz : records, tuples, lists

Record

MakeRecord {Record.make +L +LIs ?R}
returns a new record with label L, features LIs, and fresh variables at every field.

Width {Record.width +R ?I}
returns the width of R in I.

Arity {Record.arity +R ?LIs}
returns the list of features (arity) LIs of R.
For example, {Arity a(nil 7 c: 1 b: c)} yields [1 2 3 c] as output.

toList {Record.toList +R ?Xs}
binds Xs to list of all fields of R in the order as given by Arity.
For example, {Record.toList f(a a: 2 b: 3)} yields [a 2 3] as output.

map {Record.map +R1 +P ?R2}
For example, {Record.map a(12 b: 13 c: 1) IntToFloat} yields the record a(12.0 b: 13.0 c: 1.0) as output.

mapInd {Record.mapInd +R1 +P ?R2}
the ternary procedure P is applied with the index as first actual argument.
For example, {Record.mapInd a[1: d 3: a f: e] fun {$ I A} A(I) end}
yields the record a[1: d(1) 3: a(3) f: e(f)] as output.

forall {Record.forAll +R +PO}
For example, {Record.forAll 01#02#03 proc {$ O} {O do ()} end} sends the message do() to the objects 01, 02, and 03.

Tuple

MakeTuple {Tuple.make +L +I ?T}
binds T to new tuple with label L and fresh variables at features 1 through I.

append {Tuple.append +T1 +T2 ?T3}
returns a tuple with same label as T2.

List

MakeList {List.make +I ?Xs}
returns a list of length I. All elements are fresh variables.

Append {List.append +Xs Y ?Zs}
For example, {Append [1 2] [3 4]} returns the list [1 2 3 4].

Length {List.length +Xs ?I}
returns the length of Xs.

Nth {List.nth +Xs +I ?Y}
returns the Ith element of Xs (counting from 1).

Reverse {List.reverse +Xs ?Ys}
returns the elements of Xs in reverse order.

Flatten {List.flatten +Xs ?Ys}
binds Ys to the result concatenating all sublists of Xs recursively.

Map {List.map +Xs +P ?Ys}
returns the list obtained by applying P to each element of Xs.
For example, {Map [12 13 1] IntToFloat} returns [12.0 13.0 1.0].

forall {List.forAll +Xs +PO}
For example, {List.forAll 01 02 03 proc {$ O} {O do ()} end} sends the message do() to the objects 01, 02, and 03.
Oz : constraints

Telling Domains
:: ?D::+Spec {FD.int +Spec ?D}
tells the constraint store that D is an integer in Spec.
::: Dv:::+Spec {FD.dom +Spec ?Dv}
tells the constraint store that Dv is a vector of integers in Spec.
Waits until Dv is constrained to a vector.
list {FD.list +I +Spec ?Ds}
tells the constraint store that Ds is a list of integers in Spec of length I.
tuple {FD.tuple +L +I +Spec ?Dt}
tells the constraint store that Dt is a tuple of integers in Spec of width I and label L.
record {FD.record +L +Ls +Spec ?Dr}
tells the constraint store that Dr is a record of integers in Spec with features Ls and label L.
decl {FD.decl ?D}
Abbreviates {FD.int 0#FD.sup D}.

Propagators
distinct {FD.distinct *Dv}
All elements in Dv are pairwise distinct.
distinctOffset {FD.distinctOffset *Dv +Iv}
For all i,j : Dv.i + Iv.i = Dv.j + Iv.j and I has a value.
distance {FD.distance *D1 *D2 +A +D3}
creates a propagator for |D1-D2| A D3, A being '=:', '>:' etc.
sum {FD.sum *Dv +A *D}
creates a propagator for D1+...+Dn A D, A being '=:', '>:' etc.
sumC {FD.sumC +Iv *Dv +A +D}
creates a propagator for the scalar product of the vectors Iv and Dv:
I1*D1+...+In*Dn A D, A being '=:', '>:' etc, and I has a value.

Reified constraints
reified.int {FD.reified.int +Spec *D1 D2}
reifies {FD.int Spec D1} into D2.
reified.dom {FD.reified.dom +Spec Dv D}
reifies {FD.dom Spec Dv} into D.
reified.distance {FD.reified.distance *D1 *D2 +A +D3 D4}
reifies {FD.distance D1 D2 A D3} into D4.
reified.sum {FD.reified.sum *Dv +A *D1 D2}
reifies {FD.sum Dv A D1} into D2.
reified.sumC {FD.reified.sumC +Iv *Dv +A *D1 D2}
reifies {FD.sumC Iv Dv A D1} into D2.

0/1 Propagators
conj {FD.conj $D1 $D2 $D3}
D3 is the conjunction of D1 and D2.
disj {FD.disj $D1 $D2 $D3}
D3 is the disjunction of D1 and D2.
nega {FD.nega $D1 $D2}
D2 is the negation of D1.