Finite Domains and Exclusions as First-Class Citizens

Harold Boley

DFKI
Box 2080, 67608 Kaiserslautern, Germany
boley@informatik.uni-kl.de

Abstract. Languages based on logical variables can regard finite domains, finite exclusions, and, generally, types as values. Like a variable can be bound to a non-ground structure which can be later specialized through in-place assignment of some inner variables, it can also be bound to, say, a domain structure which can be specialized later through 'in-place deletion' of some of its elements (e.g. by intersection with other domain structures). While finite domains prescribe the elements of a disjunctive structure, the complementary finite exclusions forbid the elements of a conjunctive structure. Domains and exclusions can be values of variables or occur inside clauses as/in terms or within an occurrence-binding construct (useful to name arbitrary terms). In a relational-functional language (e.g., RELFUN) they can also be returned as values of functions. Altogether, domains and exclusions become first-class citizens. Because they are completely handled by an extended unification routine, they do not require delay techniques needed in (more expressive) constraint systems. Still, their backtracking-superseding 'closed' representation leads to smaller proof trees (efficiency), and abstracted, intensional answers (readability). Anti-unification (for generalization) exchanges the roles of domains and exclusions. The operational semantics of domains, exclusions, and occurrence bindings is specified by a RELFUN meta-`unify` function (and implemented in pure LISP).[1]

1 Introduction

Characteristic for logic programming (LP) is its uniform variable concept: the single construct of logical variables is usable in different *modes* (input, output, or mixed). However, mainly for efficiency (control) reasons, *committed-choice languages* have compromised this uniformity: they distinguish modes at the user level (e.g., 'read-only' annotations). Similarly, *finite domains*, which turned out to be most useful in constraint systems [16], can entail a compromised variable concept: they introduce 'domain' variables separately from logical variables, limiting which variables may be unified with which kind of term (e.g., domain variables must not be bound to logical variables).

[1] This research was supported by the BMFT under Grants ITW 8902 C4 and 413-5839-ITW9304/3.

The latter problem leads us to the issue of extending LP languages by a clean construct for finite domains (generally, types), deeply integrated with existing LP constructs. In other words, we come to this basic question: Is there a method of optional, predeclaration-free, variable domain restriction (generally, variable typing) fully in the spirit of logical variables? This can be answered affirmatively by applying the following principle: Instead of introducing a new kind of variable with an associated domain (type) **and** a possible value, regard the domain (type) **as** an initial value. A domain value can then be successively *constrained* or *specialized* (e.g. by intersecting it with other domain terms) until it ultimately fails or becomes an ordinary value. (The empty domain is identified with failure, the singleton domain with its single element.)

The 'type-as-value' principle will also be applied to a new type-like construct, namely *finite exclusions*, complementary to finite domains. [2] An exclusion term specifies the values that **cannot** be assigned to a variable. It becomes specialized on unification with other exclusions (here performing union!), fails when unified with one of its argument values, and transmutes to an ordinary value unequal to any of its arguments. (The empty exclusion is identified with success.)

On domain-exclusion unification the exclusion values are set-theoretically subtracted from the domain values. Thus, while a domain corresponds to a *disjunction of solved equalities*, an exclusion corresponds to a *conjunction of solved disequalities*, where 'solved' stands for single-variable constraints. General disequality constraints were introduced to LP by PROLOG II/III [4]. By considering only the special case of solved (dis)equalities we can regard constraints as typed logical variables: all their value specializations can be handled as part of the unification routine of LP languages, without need for the goal-delaying mechanisms on which constraint languages are often based.

After having established finite domains and exclusions as values of variables, we will show that they may also be used 'anonymously' anywhere a term can occur (e.g. as top-level arguments of clauses). The final step then is to allow domain and exclusion terms also as values returned by functions of functional LP extensions such as RELFUN [2]. Altogether, domains and exclusions become first-class citizens of cleanly extended relational, functional, and relational-functional languages.

2 Domain Terms

As the predefined term for finite domains we will use variable-length `dom` structures. They are built from an arbitrary finite number, n, of unordered, repetition-

[2] We will not expand much on further type-like constructs as values, but should note here that certain unary predicates p (e.g. `woman`) could be marked (with a "$"-prefix) as user-defined sorts $\$p$ (here `$woman`) that may be assigned to variables, where unification applies p to an ordinary value (e.g. `mary`) or looks up $\$p$'s *glb* (e.g. `$mother`) with another marked predicate (e.g. `$parent`) in a finite sort lattice.

free[3] constants, c_i:[4]

$$\texttt{dom[}c_1,\ldots,c_n\texttt{]}$$

In general, `dom` structures can be used like ordinary terms.

The empty and singleton domains reduce as follows (**unknown** indicates failure):

$$\texttt{dom[]} \longrightarrow \textbf{unknown}$$
$$\texttt{dom[}c\texttt{]} \longrightarrow c$$

In our RELFUN implementation, the behavior of `dom` structures is handled by an extension of the unification routine (cf. appendix A). This behavior will be described by employing RELFUN's generalized `is`-primitive for unification:

term **is** *expression*

unifies *term* (e.g. a variable) with the value of *expression* (e.g. another term).

For instance, the (left-to-right-ordered) conjunction

```
X is dom[1,2,3], X is dom[2,3,4,5]
```

initializes `X` with the three-element domain containing the integers `1`, `2`, and `3`, and then intersects it with the four-element domain containing `2`, `3`, `4`, and `5`, thus specializing the `X` value to the two-element domain `dom[2,3]`. Similarly, the conjunction

```
X is dom[1,2,3], X is dom[2,3,4,5], X is dom[1,3,5]
```

specializes `X` to a singleton domain, i.e. is equivalent to

```
X is 3
```

However,

```
X is dom[1,2,3], X is dom[2,3,4,5], X is dom[1,3,5],
                                          X is dom[1,2,4,8]
```

[3] In accordance with RELFUN's call-by-value semantics, we also permit active `dom` (and `exc`) *calls*, using round parentheses, which remove repetitions before constructing passive `dom` (and `exc`) *structures*, using square brackets.

[4] Unlike many finite-domain systems, we introduce no special treatment for integer domains here. Conversely, generalizing domain elements beyond arbitrary constants would entail complications in using finite domains: even ground structures as in `dom[f[a],f[b]]` would suggest that unification with `f[X]` be successful, nondeterministically binding `X` to `a` or `b`, where in fact the advantage of finite domains is their deterministic behavior, as in `dom[a,b]` unified with `X`, just binding `X` to the entire domain term. Rules for reducing a unification like `f[X] is dom[f[a],f[b]]` to the deterministic `X is dom[a,b]`, perhaps via `f[X] is f[dom[a,b]]`, would be a challenge for non-constant-element extensions of finite domains.

fails since `X` now degenerates to the empty domain.

Note that all orders of successive domain constraining are (result-)equivalent, including the usual left-to-right order of PROLOG's implementation of SLD resolution, which we could thus keep for our domain implementation: information about the current domain specialization can always immediately be stored as variable values, and goals need never be delayed.

There is an analogy between our finite-domain structures and the well-known non-ground structures of LP: binding a variable to a finite-domain structure corresponds to binding a variable to a non-ground structure. In both cases, when unified with another such variable, its value may become specialized:

1. Some elements of the domain structure may become deleted. (The domain structure can thus transmute to a single element.)
2. Some inner variables of the non-ground structure may become bound. (The non-ground structure can thus become a ground structure.)

This extension thus preserves the 'specializing-assignment' property of logic programming (a given value can be subsequently specialized, while arbitrary reassignment of a variable leads to failure). [5]

Two conjunctions exhibit the analogy:

```
X is dom[1,2,3], Y is dom[2,3,4,5], X is Y
```

deletes `1` from `X`, `4` and `5` from `Y`, assigning `dom[2,3]` to `X` and `Y`.

```
X is f[A,B,3,4,5], Y is f[1,B,3,D,E], X is Y
```

binds `A` to `1`, `D` and `E` to `4` and `5`, respectively, assigning `f[1,B,3,4,5]` to `X` and `Y`.

Note that the final (right-most) result of domain specializations need not be a single value such as `3` but can still be a domain value such as `dom[2,3]`, because such an 'intensional answer' is perfectly legitmate in our language; lack of further specialization possibilities does not lead to 'floundering' goals.

We can carry the analogy one step further. Instead of being assigned to a variable, a non-ground structure can occur directly everywhere a term can occur in a formula (e.g., within another structure). Such 'anonymous use' can also be permitted for finite-domain structures. An anonymous non-ground structure or domain structure has the same advantages as an anonymous variable: by eliminating variable names, 'single-occurrence' and 'back-substitutable' variables

[5] Of course, assigning type-like (e.g. domain or non-ground) structures to variables as initial 'non-terminal' values and specializing them to 'terminal' values after successful (unifying) type checks is only possible for specializing-assignment (LP) languages: in reassignment (imperative) languages, a variable has to preserve its original type 'value' – in a separate 'slot' – when assigning a terminal value to it because the type will be needed unchanged on reassigning further terminal values. This prevention of the type-as-value principle, and consequently of type 'first-classness', can be construed as one more disadvantage of imperative languages.

(non-ground structures, domain structures) can be immediately identified as such, programs become more concise, and no spurious bindings will be created.

For instance, since the variables `X` and `Y` are only used as intermediate stores, the above conjunctions via back-substitution become single expressions:

```
dom[1,2,3] is dom[2,3,4,5]
```

succeeds, bindingless, with the intersection domain `dom[2,3]`.

```
f[A,B,3,4,5] is f[1,B,3,D,E]
```

succeeds, not creating spurious bindings (just `A` = 1, `D` = 4, and `E` = 5), with the most general common non-ground structure `f[1,B,3,4,5]`.

3 Exclusion Terms

While finite domains *prescribe some constant of a disjunction*, finite exclusions *forbid every constant of a conjunction.* Thus the constants in an exclusion structure are implicitly 'negative'. If a variable is constrained by an exclusion and a domain assignment (in any order), both possibly singleton, the constants of the exclusion delete equal constants of the domain (set difference). If a variable is constrained by two exclusion assignments, their constants are taken together (set union), which **specializes** the original values.

Our predefined term for finite exclusions will be variable-length `exc` structures. They are again built from an arbitrary finite number, n, of unordered, repetition-free constants, c_i:

$$\texttt{exc}[c_1, \ldots, c_n]$$

In general, also `exc` structures can be used like ordinary terms.

The empty exclusion reduces as follows (the anonymous variable, "_", indicates success):

$$\texttt{exc}[] \longrightarrow \_$$

A singleton exclusion cannot be reduced context-freely since its element represents a single 'negative' constant, which has to await a unification partner.

In RELFUN, `exc` structures are again handled by an extension of the unification routine (cf. appendix A).

For instance, these conjunctions show three principal unifications of `exc` structures:

```
X is exc[1,2,3], Y is dom[2,3,4,5], X is Y
X is dom[1,2,3], Y is exc[2,3,4,5], X is Y
X is exc[1,2,3], Y is exc[2,3,4,5], X is Y
```

The first binds `X` to an exclusion of `1`, `2`, and `3`, `Y` to `dom[2,3,4,5]`, and then subtracts the former from the latter, specializing both `X` and `Y` to `dom[4,5]`. The second symmetrically 'excludes' `2` through `5` from `dom[1,2,3]`, ultimately binding `X` and `Y` to `dom[1]` or `1`. The third leads to `X` and `Y` being bound to the united exclusion `exc[1,2,3,4,5]`.

Note that an exclusion can result from unification only if both respective unification partners are `exc` structures. If one partner is a `dom` structure or a constant, either of these kinds of terms also appears in successful results; `exc` structures "subtract and disappear". Thus, the first result, `dom[4,5]`, is a – sufficiently specialized – finite domain ("Only constants `4` or `5` are allowed"), while, say, `exc[1,2,3,6,...]` would not be a – sufficiently specialized – finite exclusion ("All constants but `1` and `2` and `3` and `6` and ... are allowed").

Like for domains, we can choose any order of exclusion constraining, and thus keep the left-to-right order: the negative information of exclusions is also stored as part of the variable substitution, not with goals, which, again, need never be delayed. Also, if only exclusions are involved, the right-most result of exclusion specializations still is a 'negative answer' such as `exc[1,2,3,4,5]`; if all intermediate values are identical singleton exclusions, a 'negative singleton answer' such as `exc[3]` arises.

Exclusions can also be used anonymously, with the same advantages as mentioned for anonymous domains (see end of section 2). For instance, shortening the above conjunctions, the expressions

```
exc[1,2,3] is dom[2,3,4,5]
dom[1,2,3] is exc[2,3,4,5]
exc[1,2,3] is exc[2,3,4,5]
```

succeed bindingless with, respectively, the difference domain `dom[4,5]`, the difference constant `1`, and the united exclusion `exc[1,2,3,4,5]`.

Summarizing the domain and exclusion constructs, a 'domain assignment'

$$X = \texttt{dom}[c_1, \ldots, c_n]$$

corresponds to the disjunction of X-solved equalities

$$X = c_1 \vee \ldots \vee X = c_n \tag{1}$$

with "=" being used like RELFUN's "`is`", while an 'exclusion assignment'

$$X = \texttt{exc}[c_1, \ldots, c_n]$$

corresponds to the conjunction of X-solved disequalities (where (2) = ¬(1) shows that exclusions are negated domains)

$$X \neq c_1 \wedge \ldots \wedge X \neq c_n \tag{2}$$

with "≠" having no direct analogue in RELFUN. However, since in such conjunctions (in RELFUN written with "," instead of "∧") exclusion values become united, the equivalent n-ary exclusion assignment

$$X \texttt{ is exc}[c_1, \ldots, c_n]$$

naturally corresponds to the following conjunction of n unary ones:

$$X \texttt{ is exc[}c_1\texttt{]}, \ \ldots, \ X \texttt{ is exc[}c_n\texttt{]}$$

Thus, finite exclusions express negative information as **values** ('object-centered') that can be simply passed around and unified like positive information, while LP extensions via a "$\neq$" **connective** (symmetric) suggest two-variable constraints like $X \neq Y$, normally entailing another layer of complexity such as the need to delay a disequality until a variable becomes bound. (A possible non-ground extension of exclusions for representing two-variable constraints will be discussed in section 9.)

4 Occurrence Bindings

Let us further introduce a generally useful construct for binding a variable to some (initial) value(s) at one or more of its occurrences in arbitrary formulas. If this is a type-like value, e.g. a non-ground structure or a domain or an exclusion, it can become specialized by subsequent unification.

Occurrence bindings are written as binary `bnd` structures built from a variable, v, and a term, t: [6]

$$\texttt{bnd[}v, t\texttt{]}$$

In general, `bnd` structures can be used as terms.

Taking a non-ground-structure example,

```
bnd[X,f[A,B,3,4,5]] is f[1,B,3,D,E]
```

binds `X` to `f[A,B,3,4,5]`, which is then unified with `f[1,B,3,D,E]`, binding `A` to `1`, `D` and `E` to `4` and `5`, respectively, thus specializing the `X` value to `f[1,B,3,4,5]`.

An analogous finite-domain example,

```
bnd[X,dom[1,2,3]] is dom[2,3,4,5]
```

binds `X` to `dom[1,2,3]`, which is then unified with `dom[2,3,4,5]`, thus specializing the `X` value to `dom[2,3]`.

A complementary finite-exclusion example,

```
bnd[X,exc[1,2,3]] is dom[2,3,4,5]
```

binds `X` to `exc[1,2,3]`, which is then unified with `dom[2,3,4,5]`, thus specializing the `X` value to `dom[4,5]`.

If the unification partner of an occurrence binding is directly given, here as the `is`-rhs (right-hand side), the `bnd` structure can always be equivalently replaced by an initializing ('pre-typing') `is` call:

[6] One could also use an infix notation like $v : t$ for increased conciseness. If t was the sort-marked predicate $\$p$, bnd$[v, \$p]$ would then shorten to $v : \$p$. The current implementation still has restrictions wrt the t's allowed in bnds. Section 5.2 will detail on the elimination of occurrence bindings.

```
X is f[A,B,3,4,5], X is f[1,B,3,D,E]
X is dom[1,2,3],   X is dom[2,3,4,5]
X is exc[1,2,3],   X is dom[2,3,4,5]
```

For bnds in clause heads, however, the unification partner is not directly given, as will be illustrated by the relational examples in section 5.2.

The binding construct, pairing a variable with a value, can again be assigned to a variable. Actually, in our implementation it is generated from dom/exc-bound variables at the end of reference chains to keep track of domain/exclusion specializations (while non-ground structures can be specialized via direct in-place assignments).

5 Domains/Exclusions in Relation Definitions

5.1 Facts and dom/exc Reductions

Starting with domains, the fact with a single-occurrence variable X,

```
likes(john,bnd[X,dom[ann,mary,susan]]).
```

is equivalent to the fact using the domain anonymously (regard "X" as "_"):

```
likes(john,dom[ann,mary,susan]).
```

Both can be equivalently queried by ("%" precedes comments)

```
likes(john,mary)  % success
likes(john,peggy) % failure
likes(john,Whom)  % success: Whom = dom[ann,mary,susan]
likes(john,dom[mary,peggy,susan])  % success
likes(john,bnd[Whom,dom[mary,peggy,susan]])
                  % success: Whom = dom[mary,susan]
likes(john,exc[mary,peggy])  % success
likes(john,bnd[Whom,exc[mary,peggy]])
                  % success: Whom = dom[ann,susan]
```

We can reduce the dom fact, obtaining the three 'multiplied out' facts

```
likes(john,ann).
likes(john,mary).
likes(john,susan).
```

Note that the queries would be answered equivalently. However, 'intensional' answers (delivering one closed dom structure) would become 'extensional' answers (enumerating several constants); so the bnd/dom query, instead of binding Whom to dom[mary,susan], would first bind Whom to mary, and then, via backtracking, to susan.

If we let $cls_{i_1,\ldots,i_k}(x)$ denote a clause with term x at some position $i_1,\ldots,i_k$ ($i_1 = 0$ being the head, $i_1 = 1$ the first premise, ...,[7] $i_2 = 0$ being i_1's operator/constructor, $i_2 = 1$ its first argument, ..., etc.) and $cls_\sim(x)$ a clause not having the term x at any position, then a general *multout* algorithm can be defined recursively via an equation schema (treating queries as answer-head rules):

$$multout(cls_\sim(\texttt{dom[}c_1,\ldots,c_n\texttt{]})) = cls_\sim(\texttt{dom[}c_1,\ldots,c_n\texttt{]})$$

$$multout(cls_{i_1,\ldots,i_k}(\texttt{dom[}c_1,\ldots,c_n\texttt{]})) = \begin{cases} multout(cls_{i_1,\ldots,i_k}(c_1)) \\ \ldots \\ multout(cls_{i_1,\ldots,i_k}(c_n)) \end{cases}$$

For example, $multout(\texttt{likes(john,dom[ann,mary,susan])})$ matches the second equation via the instantiation $multout(cls_{0,2}(\texttt{dom[ann,mary,susan]}))$, whose rhs's through the first equation lead to the three **dom**less facts shown above.

Continuing with exclusions, the fact with a single-occurrence variable X,

```
likes(john,bnd[X,exc[mary,claire,linda]]).
```

is equivalent to the fact using the exclusion anonymously (since "X" is "_"):

```
likes(john,exc[mary,claire,linda]).
```

Both can be interchangeably queried by

```
likes(john,mary)  % failure
likes(john,peggy) % success
likes(john,Whom)  % success: Whom = exc[mary,claire,linda]
likes(john,dom[mary,peggy,susan])  % success
likes(john,bnd[Whom,dom[mary,peggy,susan]])
                  % success: Whom = dom[peggy,susan]
likes(john,exc[mary,peggy])  % success
likes(john,bnd[Whom,exc[mary,peggy]])
                  % success: Whom = exc[peggy,mary,claire,linda]
```

If we have a 'closed universe' of a finite number, say 8, of individuals, e.g. $\{ann, claire, john, linda, mary, peggy, susan, tina\}$, we could reduce the **exc** fact, obtaining the five 'complemented out' facts

```
likes(john,ann).
likes(john,john).
likes(john,peggy).
likes(john,susan).
likes(john,tina).
```

where the bnd/dom query would now first bind Whom to peggy, then, via backtracking, to susan. (These facts are also the multiplied out form of a **dom** fact.)

If the 'non-Horn' extension of a (classic, strong) negation construct is available for facts, e.g. via **false**-valued functions in RELFUN, one could also approximate the **exc** fact in an 'open universe', with infinite complements, by

[7] Since the last premise may constitute the value of a functional clause, the *multout* algorithm below will also work for function definitions.

```
likes(john,dom[mary,claire,linda]) :- ! & false.
likes(john,X).
```

Queries as shown above could now bind a second argument Whom to the dom by (successfully!) returning false, but would, e.g., also return a bindingless false for mary (rather than yielding unknown due to unification failure). The impurity of the cut-protected 'catch-all' fact seems to favor our proposal to express such special cases of negation by the special-purpose construct exc directly in clause heads, permitting non-Horn clauses as "Horn clauses + exclusions".

5.2 Clauses and bnd-to-is Reductions

A typed version of a well-known PROLOG program contains a rule with a non-single-occurrence variable X, whose head occurrence is domain-bound:

```
likes(john,bnd[X,dom[ann,mary,susan]]) :- likes(X,wine).
likes(dom[mary,peggy,susan],wine).
```

The query

```
likes(john,Whom)
```

here binds Whom to dom[mary,susan]. The query (indefinite even wrt john)

```
likes(dom[fred,john],bnd[Whom,dom[ann,susan,tina]])
```

binds Whom to susan (not selecting fred or john from the anonymous dom).

A 'negatively' typed version of the program again contains a rule with a non-single-occurrence variable X, whose head occurrence is exclusion-bound:

```
likes(john,bnd[X,exc[mary,claire,linda]]) :- likes(X,wine).
likes(exc[mary,peggy,susan],wine).
```

The query

```
likes(john,Whom)
```

now binds Whom to exc[peggy,susan,mary,claire,linda]. The query

```
likes(dom[fred,john],bnd[Whom,dom[ann,susan,tina]])
```

binds Whom to dom[ann,tina] (again leaving "fred or john" anonymous).

A binding construct bnd[v,t] in a clause head can always be replaced by v by introducing a new premise v is t. If v is t is further transformed to $t'(v)$, applying a unary predicate t' corresponding to t, the entire reduction is similar to the reduction of a sorted logic to an unsorted one.

Thus, the bnd/dom rule is equivalent to

```
likes(john,X) :- X is dom[ann,mary,susan], likes(X,wine).
```

and, with $t' =$ ann-mary-or-susan, to

```
likes(john,X) :- ann-mary-or-susan(X), likes(X,wine).
ann-mary-or-susan(dom[ann,mary,susan]).
```

Also, the bnd/exc rule is equivalent to

```
likes(john,X) :- X is exc[mary,claire,linda], likes(X,wine).
```

and, with $t' =$ `not-mary-claire-and-linda`, to

```
likes(john,X) :- not-mary-claire-and-linda(X), likes(X,wine).
not-mary-claire-and-linda(exc[mary,claire,linda]).
```

The reduced form can perform 'type' checking only **after** unification, once the former bnd variable is bound. Unlike the transformation (in section 4) of

```
bnd[X,dom[1,2,3]] is dom[2,3,4,5]      % fact p(bnd[X,dom[1,2,3]]).
bnd[X,exc[1,2,3]] is dom[2,3,4,5]      % fact p(bnd[X,exc[1,2,3]]).
```

to the 'pre-typing' (domain/exclusion-initializing, not possible for clause heads as indicated by the "%"-comments)

```
X is dom[1,2,3], X is dom[2,3,4,5]  % X is dom[1,2,3] not in p(X).
X is exc[1,2,3], X is dom[2,3,4,5]  % X is exc[1,2,3] not in p(X).
```

the above bnd-to-is reduction thus performs 'post-typing' (domain/exclusion-specializing, generally applicable), as in

```
X is dom[2,3,4,5], X is dom[1,2,3] % rule p(X) :- X is dom[1,2,3].
X is dom[2,3,4,5], X is exc[1,2,3] % rule p(X) :- X is exc[1,2,3].
```

Unfortunately, post-typed clauses no longer permit the selectivity of typed (e.g. domain-constrained or sorted) unification and WAM-indexing and of typed anti-unification (for generalization, see section 7). Also, at least if compared with the ":"-infix syntax of bnd as usable for our versions of the PROLOG example,

```
likes(john,X:dom[ann,mary,susan]) :- likes(X,wine).
likes(john,X:exc[mary,claire,linda]) :- likes(X,wine).
```

the is-reduced formulations are less readable.

Combining post-typing with the reformulation of an is-assigned exclusion as a conjunction of solved disequalities (cf. (2) in section 3), we can repeatedly transform any n-ary-exc-head clause

$$p(..., X : \texttt{exc}[c_1, \ldots, c_n], ...) \texttt{ :- } q_1(...), \ \ldots, \ q_z(...).$$

to an equivalent unary-exc-body clause

$$p(..., X, ...) \texttt{ :- } X \texttt{ is exc}[c_1], \ \ldots, \ X \texttt{ is exc}[c_n], \ q_1(...), \ \ldots, \ q_z(...).$$

(for anonymous exclusions we choose a new variable for "X"), [8] representing

$$p(..., X, ...) \texttt{ :- } X \neq c_1, \ \ldots, \ X \neq c_n, \ q_1(...), \ \ldots, \ q_z(...).$$

where the non-Horn-clause character engendered by the exc terms is revealed by the "$\neq$" constraints preceding the ordinary premises.

[8] While we may also combine post-typing with the reformulation of an is-domain as a disjunction of solved equalities (cf. (1) in section 3), we can directly apply the

6 Finite-Domain/Exclusion Functional Programming

Having introduced finite domains and exclusions into relational programming as terms that can be values of logical variables, we now transfer them to functional programming as terms that can be arguments and values of functions. (Similarly, the binding construct can be employed in function arguments and values.)

Domains and exclusions thus become first-class citizens of relational-functional languages such as RELFUN.

6.1 Domains/Exclusions as Function Arguments

The use of finite domains as *arguments* of functions works like their use in relations. For instance, the two directed equations (":-&" is a left-to-right directed "=")

```
separates(dom[canada,mexico,usa],japan) :-& pacific.
separates(dom[canada,mexico,usa],
 dom[denmark,france,germany,italy,spain,sweden,uk]) :-& atlantic.
```

use 'anonymous' `dom` arguments for compactly defining a `separates` function. (They could be multiplied out to 24 domless equations, analogous to the 24 facts in section 7.)

The query

```
separates(bnd[Source,dom[canada,usa,panama]],
          Destination)
```

binds `Source` to `dom[canada,usa]`, `Destination` to `japan`, and returns `pacific`; on backtracking it rebinds `Destination` to the European subdomain and returns `atlantic`.

Analogously, finite exclusions act as function arguments as was shown for relation arguments. For instance, the `safe-divide` function

```
safe-divide(Nominator,bnd[Denominator,exc[0]]) :-&
                                   /(Nominator,Denominator).
```

or, using a post-typing function definition (":-" and "&" permit intervening relational premises),

```
safe-divide(Nominator,Denominator) :- Denominator is exc[0] &
                                   /(Nominator,Denominator).
```

---

multout algorithm (cf. section 5.1) to any n-ary-dom-head clause

$$p(..., X{:}\mathtt{dom}[c_1, \ldots, c_n], ...) \text{ :- } q_1(...), \ \ldots, \ q_z(...).$$

to obtain n equivalent domless clauses

$$p(..., X{:}c_1, ...) \text{ :- } q_1(...), \ \ldots, \ q_z(...). \quad \ldots \quad p(..., X{:}c_n, ...) \text{ :- } q_1(...), \ \ldots, \ q_z(...).$$

(for anonymous domains we just omit "X:").

'excludes' `Denominator`-named arguments which would lead to division by zero.

Thus, the query

```
safe-divide(8,4)
```

returns 2 because $4 \neq 0$ is true. On the other hand, the query

```
safe-divide(8,0)
```

yields **unknown** (rather than an error from the "/"-built-in) because $0 \neq 0$ is false.

Many function definitions, e.g. factorial and `fibonacci` (below) over the naturals, become more declarative than in PROLOG by **excluding**, in a defining clause, arguments of earlier clauses: the definition thus needs no cut and in fact has disjoint, order-independent ('OR-parallel') clauses. The `fib` definition can even be shortened to two clauses via complementary `dom` and `exc` arguments:

```
fib(dom[0,1]) :-& 1.
fib(bnd[N,exc[0,1]]) :-& +(fib(-(N,2)),fib(-(N,1))).
```

6.2 Functions with Domain/Exclusion Values

The use of finite domains as *values* of functions works as follows. Like any other term, a domain term can be specified as (part of) the returned value in a function definition. Such a function then returns the finite domain to its caller as a 'closed' term representing a finite number of non-deterministic values, which without domain terms available would typically be enumerated via backtracking.

For instance, the directed equations

```
direction(old) :-& dom[east,west].
direction(new) :-& dom[north,south].
direction(all) :-& dom[north,west,south,east].
```

use `dom` values for compactly defining a `direction` function. The first clause, e.g., can be regarded as a 'closed' form of the non-deterministic, two-clause function definition produced by *multout* (section 5.1):

```
direction(old) :-& east.
direction(old) :-& west.
```

A main call unifies returned domain terms just like for anonymously specified domains. For instance, using the variable-length `tup` function for list building,

```
tup(direction(old),direction(new))
```

just like

```
tup(dom[east,west],dom[north,south])
```

returns `[dom[east,west],dom[north,south]]`.

In particular, a domain functionally returned to the top-level gives the user a more compact representation of results than their enumeration, much like a domain assigned to a relational request variable.

We may also call domain-valued functions within `is`-calls. For example, while the query

```
D is direction(old), D is direction(new)
```

fails (the domains are disjoint), the query

```
D is direction(all), D is direction(new)
```

succeeds, temporally binding `D` to `dom[north,west,south,east]`, but then specializing it to `dom[north,south]`.

The `is`-embedded non-ground functional query

```
[new,dom[west,north]] is tup(Which,direction(Which))
```

succeeds by binding, as its second attempt, `Which` to `new` and building the list `[new,dom[north,south]]`, whose *most general* 'instantiation' in common with the `is`-lhs (left-hand side) is the domless ground list `[new,north]`.

Analogously, an exclusion term can be (part of) the returned value of a function. For instance, the definition

```
permitted(butcher-shop) :-& exc[dog].
permitted(pet-shop) :-& exc[cat,dog].
```

prohibits certain entries to butcher and pet shops: the non-ground call

```
permitted(Where)
```

enumerates the exclusion values `exc[dog]`, binding `Where` to `butcher-shop`, and `exc[cat,dog]`, binding `Where` to `pet-shop`.

Two such `permitted` calls may be embedded into an `is`-call:

```
[cat,dom[kid,dog]] is tup(permitted(Where),permitted(Where))
```

This succeeds by specializing the `is`-lhs to `[cat,kid]`, consistently binding `Where` to `butcher-shop`.

Finally, a function can also return a mix of domains and exclusions. For example, the `dish`es (dis)liked by several people may be defined thus:

```
dish(john) :-& dom[chilli,pizza,sushi,chop-suey].
dish(mary) :-& exc[sushi].
dish(fred) :-& exc[spaghetti,pizza].
dish(tina) :-& dom[sushi,chop-suey,hamburger].
```

For constraining the set of candidate restaurants, they could perform intersection-difference operations equivalent to

```
[D,D,D,D] is tup(dish(john),dish(mary),dish(fred),dish(tina))
```

binding `D` to the (fortunately unique) solution `chop-suey`.

7 Domain and Exclusion Anti-Unification

In section 5.1 we have defined the *multout* algorithm for 'multiplying out' finite domains from clauses into an extensional form, and noted that the general reduction of finite exclusions would involve a strong form of negation.

Conversely, the automatic generation of intensional, domain/exclusion-using clauses from ordinary ones constitutes an interesting generalization task. In particular, a set of 'similar' clauses can often be generalized by individually generating a finite domain in each distinguishing argument position, thus 'compressing' the clauses' information. Generalizing more than one argument position at a time (giving rise to new combinations when multiplying out) amounts to 'inducing' new information from the clauses.

For instance, inverting two *multout* transformations, the 24 relational(ized) `separates` facts

```
separates(pacific,canada,japan).
separates(pacific,mexico,japan).
separates(pacific,usa,japan).
separates(atlantic,canada,denmark).
separates(atlantic,canada,france).
separates(atlantic,canada,germany).
separates(atlantic,canada,italy).
separates(atlantic,canada,spain).
separates(atlantic,canada,sweden).
separates(atlantic,canada,uk).
separates(atlantic,mexico,denmark).
separates(atlantic,mexico,france).
separates(atlantic,mexico,germany).
separates(atlantic,mexico,italy).
separates(atlantic,mexico,spain).
separates(atlantic,mexico,sweden).
separates(atlantic,mexico,uk).
separates(atlantic,usa,denmark).
separates(atlantic,usa,france).
separates(atlantic,usa,germany).
separates(atlantic,usa,italy).
separates(atlantic,usa,spain).
separates(atlantic,usa,sweden).
separates(atlantic,usa,uk).
```

can be generalized (compressed) to the two facts[9]

[9] If some (interactive/automatic) analyzer notices that a certain domain such as `dom[canada,mexico,usa]` occurs repeatedly in a program, it may be useful to have it defined more globally as a predicate (with a user-provided name) such as `america(dom[canada,mexico,usa])` and replace the domain by the predicate name used as a "$"-marked sort, e.g. in the clause `separates(pacific,$america,japan)`.

```
separates(pacific,dom[canada,mexico,usa],japan).
separates(atlantic,dom[canada,mexico,usa],
 dom[denmark,france,germany,italy,spain,sweden,uk]).
```

which are relationalized versions of the **separates** function in section 6.1. [10]

A simple method for this (least general) generalization is pairwise *domain anti-unification* of the input facts. For ease of presentation we will assume that clauses are represented as structures, e.g. regarding an atom (fact) as a structure whose constructor stands for the predicate. Domain anti-unification of two structures works like classic anti-unification [11] (in our implementation, [5], (nested) structures having different constructors or arities yield a new variable) with the following modifications. For a (named or anonymous) variable and a domain it yields a variable in the manner classic anti-unification handles variable/constant pairings. For different constants it yields a **dom** term containing these constants, not a (sometimes overly general) new variable. (For a constant and a structure it has to yield a new variable since currrent **dom** terms cannot contain structures.) Generally (constants can be treated as singleton domains), domain anti-unification of two **dom** terms yields their union (unification: intersection). Identical **dom** (later: **exc**) terms can directly yield one copy unchanged, short-cutting spurious unions (later: intersections).

The complementary *exclusion anti-unification* for a (named or anonymous) variable and an exclusion yields a variable in the manner classic anti-unification handles variable/constant pairings. It yields the intersection (unification: union) of two **exc** terms. For an exclusion and a constant (singleton domain) it yields the **exc** term minus the constant. Generally, the *domain-exclusion anti-unification* of a **dom** and an **exc** term, in any order, yields the **exc** term with the elements of the **dom** term set-theoretically subtracted (unification: domain with exclusion subtracted). An empty-exclusion outcome, as usual, represents the always successful anonymous variable. Altogether, the domain/exclusion complementarity commutes nicely with the unification/anti-unification duality.

Let us start an example for domain anti-unification with, say, the first two input facts:

```
separates(pacific,canada,japan).
separates(pacific,mexico,japan).
```

A comparison of the equivalent notations '**dom**[. . .]' and '$...' reveals our convention that domains/exclusions do not carry a 'typing symbol' such as the "$" for sorts: their **dom**/**exc**-constructor marks them as types with 'built-in' unification behavior; on the other hand, "$"-less predicate names are just constants unifying with themselves. Domains/exclusions exhibit their built-in properties in all places they are permitted as first-class citizens. Making them passively passable data structures (without list-coding as in appendix A), e.g. for amalgamated object/meta-level programming, is as hard as for logical variables, requiring a kind of quote operator.

[10] In RELFUN the **relationalize** algorithm can be used to make relational/functional knowledge more accessible to such inductive-LP methods, which we study wrt efforts in knowledge Validation and Exploration by Global Analysis (VEGA).

Anti-unification generalizes them via a domain in the second argument:

```
separates(pacific,dom[canada,mexico],japan).
```

This intermediate result domain-anti-unified with the third input fact,

```
separates(pacific,usa,japan).          % usa = dom[usa]
```

leads to the completely generalized `pacific` fact above. Similarly, the remaining input facts, via three groups of textually ordered domain-anti-unification steps, generalize their third argument to a common domain:

```
separates(atlantic,canada,
 dom[denmark,france,germany,italy,spain,sweden,uk]).
separates(atlantic,mexico,
 dom[denmark,france,germany,italy,spain,sweden,uk]).
separates(atlantic,usa,
 dom[denmark,france,germany,italy,spain,sweden,uk]).
```

The completely generalized `atlantic` fact above is then obtained as for the `pacific` side. (Equivalently, the second argument could be generalized first.)

Suppose we have one additional input fact,[11]

```
separates(atlantic,panama,denmark).
```

For group formation on the third argument, domain anti-unification would leave this fact as a singleton group since `denmark` is the only European partner specified for `panama`. Now, the four resulting groups differ in two arguments, not just in one. Still domain-anti-unifying them would generalize the second argument and 'absorb' `denmark` into the domain of the third argument:

```
separates(atlantic,dom[canada,mexico,usa,panama],
 dom[denmark,france,germany,italy,spain,sweden,uk]).
```

This generalized `atlantic` fact expresses more information than the input facts, namely an induction from Denmark to the other European countries (which happens to be empirically true); again multiplying out the result makes these induced facts explicit:

```
separates(atlantic,panama,france).
. . .
separates(atlantic,panama,uk).
```

However, since (domain) anti-unification can find a generalization for each pair of structures, its use most be controlled. An example of overgeneralization would result from further domain-anti-unifying the completely generalized `pacific` and `atlantic` facts above, generating a single fact expressing much more than the 24 inputs via geographically vacuous Pacific/Atlantic and Japan/Europe domains.

An example for exclusion anti-unification can take two versions of a fact from section 5.1 as input:

[11] Such a `separates` enrichment was proposed by Manfred Meyer and Knut Hinkelmann. Thanks also to Otto Kühn, Michael Sintek, and Panagiotis Tsarchopoulos.

```
likes(X,exc[mary,claire,linda]).  % Everybody likes all except MCL
likes(john,exc[mary,tina]).       % John likes all except Mary & Tina
```

Anti-unification generalizes them via an intersection of the exclusions in the second argument:

```
likes(X,exc[mary]).                % Everybody likes all except Mary
```

This is the least general generalization of the input facts since exactly the subexclusion common to both facts is kept. In cases where we have a closed universe, say $\{ann, claire, john, linda, mary, peggy, susan, tina\}$ of section 5.1, the inputs can be rewritten as complementary domain facts:

```
likes(X,dom[ann,john,peggy,susan,tina]).   %  (*)
likes(john,dom[ann,claire,john,linda,peggy,susan]).
```

Domain anti-unification via union generalizes them to

```
likes(X,dom[ann,claire,john,linda,peggy,susan,tina]).
```

which is the complement of the exclusion-anti-unification result above.

Finally, domain-exclusion anti-unification of the input facts

```
likes(X,exc[mary,claire,linda]).
likes(john,dom[mary,tina]).   %  (**)
```

via subtraction generalizes them to

```
likes(X,exc[claire,linda]).
```

Here, the exclusion is minimally weakened (its extension being minimally enlarged) to accomodate what is specified by the domain. This can again be illustrated for the case of a closed universe: anti-unify (*) with (**) and recomplement the result. Such least general generalizations by domain-exclusion anti-unification thus remove **dom-exc** contradictions in a set of clauses, e.g. about John's liking of Mary in the above input facts; similarly, exclusion anti-unification removes the less obvious **exc-exc** contradictions concerning constants that occur in only one of the exclusions, e.g. about John's liking of, say Claire, in the previous input facts. This may be exploited for 'theory revision' [12] of knowledge bases containing exclusion terms.

8 Operational Semantics

Since all user-defined relations and functions are invoked through unification, we were able to handle the relational-functional domain extensions in a uniform, efficient manner by building our first-class **domain** and **exclusion** notions, as well as the larger part of our **bnds**, into the (pure LISP) unification routine **unify** of the definitional interpreter of RELFUN. (A smaller, less interesting part of occurrence bindings is built into the term-instantiation routine, not treated

here.) In appendix A we use a meta-interpreter approach for specifying the operational semantics of the extended **unify** via RELFUN clauses only relying on non-extended unification. This will contain enough detail both to document the actual RELFUN implementation and to permit transfers to other LP languages.

While constants will stand for themselves, non-constant terms will be coded as ground lists as shown by the table below, where "'" indicates recursive coding.

constant	*constant*
Identifier	[**vari**, *identifier*]
*Identifier***level*	[**vari**, *identifier*, *level*]
$[a_1, \ldots, a_n]$	[**tup**, $a'_1, \ldots, a'_n$]
constructor$[a_1, \ldots, a_n]$	[*constructor'*, $a'_1, \ldots, a'_n$]
dom$[c_1, \ldots, c_n]$	[**dom**, $c_1, \ldots, c_n$]
exc$[c_1, \ldots, c_n]$	[**exc**, $c_1, \ldots, c_n$]
bnd$[v, t]$	[**bnd**, v', t']

Substitutions will be represented as lists of pair lists of variables and their values of the form $[[v'_1, t'_1], \ldots, [v'_n, t'_n]$, `[bottom]]`, i.e. the empty substitution becomes `[[bottom]]` (not `[]`, see below).

For instance, the call

```
unify( [bnd,[vari,x],[exc,a,b,c]], [dom,b,c,d,e], [ [bottom] ] )
```

successfully returns the substitution `[ [[vari,x],[dom,d,e]], [bottom] ]`.

In appendix A, the **unify** function takes two terms **X** and **Y** and a substitution **Environment** (initially often empty), and returns the substitution extended by the mgu of **X** and **Y** in **Environment** (on success) or **[]** (on failure). It calls **unify-ua** with **ultimate-assoc**-dereferenced **X/Y** arguments for case analysis. This workhorse decomposes one or two **bnds** into their variable and expression parts for **unify-bnd**, where a missing **bnd** (variable) is indicated by **[]**. Mixed **dom/exc** arguments are handed to **dom-exc**, performing (set-as-list) subtraction. Homogeneous **doms** are handed to **dom-intersection** for (set-as-list) intersection. In both cases (only) the non-emptiness of the result list is checked (so this can be optimized). Homogeneous **excs** are successful in any case. Plain partner arguments to **doms** and **excs** are checked via **member** calls simplifying earlier cases with singleton **doms** reduced to the plain argument. The last **unify-ua** clause does **unify** on constructors (incl. **tup**) and calls **unify-args** (not expanded here) for corecursive processing of their arguments. The **unify-bnd** function essentially parallels the **dom** and **exc** cases of **unify-ua**, but hands subtraction, intersection, and union results to **unify-bnd-env** for extension of the **Environment** argument, using the variable(s) of the **bnd**(s).[12] Such **bnds** for **dom/exc**-variable

[12] Thus, while the update of non-ground structures in relational languages leads to bindings of free inner variables, the update of **dom** and **exc** structures leads to bindings shadowing previous ones, as known from function calls and **let** blocks in interpreters for functional languages. In a (WAM) compiler implementation we could get the efficiency of in-place assignment via real in-place deletion/addition of elements of **dom/exc** structures allocated on the heap.

updates may be generated by the function `ultimate-assoc`: it returns the dereferenced value of a variable in `Environment`, except if the value is a `dom` or an `exc`, in which case it creates a `bnd` pair of the variable immediately preceding in the reference chain and of the `dom` or `exc` expression.

While RELFUN's generalized `is`-primitive also automatically profits from the `dom`/`exc`-enhanced unification, for ordinary built-in relations and functions the actual arguments that are finite domains have to be 'multiplied out' (built-in calls cannot have exclusion arguments); for built-in (constant-valued) functions the values then have to be recollected into a new domain structure.

As we have seen in section 5.1, the *multout* transformation could be performed statically for user-defined operations, too, thus eliminating the domain extension for a non-enhanced LP implementation. However, this would lose the combinatorial efficiency advantage of finite domains. Also, their complementarity with finite exclusions, not allowing this treatment, would become occluded.

For a model-theoretic characterization [9] of programs containing first-class finite domains, the *multout* transformation could also be exploited semantically. Of course, a characterization via a domain-extended Herbrand base would be more 'direct'. And again, leaving domains in the semantic kernel would allow to exploit the domain/exclusion complementarity.

9 Conclusions

Let us briefly summarize our notion of finite domains and exclusions:

- They are useful even without constraint (delay!) techniques because their backtracking-superseding 'closed' representation leads to
  - smaller proof trees (efficiency),
  - abstracted, intensional answers (readability).
- We have generalized them to first-class citizens (values of logical variables and of functions, usable anonymously as arguments and inside structures, no 'floundering' for non-singleton domain results).
- Their complementarity wrt unification (most general specialization) 'changes signs' wrt anti-unification (least general generalization).
- Their operational semantics and interpreter implemention is given by extensions of the unification routine of LP languages (specified here via meta-unification).

The examples of this paper have indicated ways of employing our finite domain/exclusion concept for the compact representation of first-order knowledge. In RELFUN, domain/exclusion terms can also be used in the operator position, thus permitting a higher-order notation for knowledge like "Functions `factorial`, `fibonacci`, or `exponential` applied to 0 return 1" (domain anti-unification also generalizing operators/constructors could extract this from three multiplied out functional clauses):

```
dom[fac,fib,exp](0) :-& 1.   % F:dom[exp,sin](0) gives 1, F=exp
```

It will be instructive to observe which particular use of our domain/exclusion extension of LP is most profitable for a real-world representation task, e.g. in the areas of materials engineering [3] or calendar management (e.g. just unify two agents' restrictions, "All dates except May 12 and 23" and "Only May 9-13": `exc[12-may,23-may] is dom[9-may,...,13-may]`).

An area for further theoretical work would be the extension of Herbrand models for finite domains and, more demanding (perhaps via $T_P \downarrow \omega$ [9]), finite exclusions. Concerning domain/exclusion anti-unification, it will be interesting to see how further inductive-LP or machine-learning methods based on classic anti-unification may profit from the domain/exclusion extension, using our recent LISP implementation [5] of the rules introduced in section 7. On the unification side, an efficient WAM compiler/emulator extension for our (variable-length!) finite domains and exclusions should be written, building on the RELational/FUNctional machine [1], FIDO III [7, 15], and FLIP [14], all in COMMON LISP: WAM instructions for unifying constants such as **`get_constant`** would need a membership/non-membership test case for **`dom/exc`** structures, new instructions **`get_dom/get_exc`** could unify **`dom/exc`** structures, performing, e.g., intersection/union for other **`dom/exc`** structures (perhaps maintaining canonically ordered elements), etc. Also, it could be studied how our specialized finite domains/exclusions could be fruitfully **characterized** as a CLP($\mathcal{FD}$)-like instance of the constraint-logic programming scheme [8], and if they could be usefully **combined** with our RELFUN-implemented finite-domain constraints FINDOM [13] or those in FIDO [10], or with concrete domains [6], or other, more general constraint formalisms.

Finally, let us explore a possible non-ground extension of the treatment of solved disequations, e.g. $\mathtt{X} \neq \mathtt{1}$, as exclusion bindings, e.g. `X is exc[1]`, if only to confirm that ground exclusions in fact constitute the 'local optimum' suggested by section 3: Can we treat unsolved disequations, e.g. $\mathtt{X} \neq \mathtt{Y}$, as exclusion bindings with non-ground rhs's, e.g. `X is exc[Y]` and/or `Y is exc[X]`? Well, we could store both binding directions, but let us choose one direction, say `X is exc[Y]`, and put this into the substitution. If further computation instantiates `Y` to a constant, say `1`, perhaps via a binding chain, the disequation reduces to a solved form, `X is exc[1]`, treated as usual. If `X` thus specializes to a constant, `1`, we can 'swap' the disequation to a solved form, `Y is exc[1]`, within the substitution. For an added disequation, say the unsolved `X is exc[Z]`, the two bindings may be simplified to one, here `X is exc[Y,Z]`. For `Y is exc[Z]`, after swapping, they can be joined to `Y is exc[X,Z]`; this avoids (possibly circular) instantiations like `X is exc[exc[Z]]`, non-equivalent to `X is exc[Z]` because "$\neq$" is not transitive. If any variable of such a (generated) non-singleton, non-ground exclusion becomes instantiated, this exclusion becomes partially solved, now constraining unifiable values (e.g. `is`-lhs's). For example, `X is exc[Y,Z]`, `Z is 2` or `X is exc[Y,2]` excludes the binding `X is 2`. If such non-ground exclusions (generally, types) can treat a larger class of constraints as bindings directly put into the substitution, unlike constraints as delayed goals, they will thus require very careful substitution updates and uses.

A The RELFUN Meta-unify

Since this RELFUN unification meta-specification in RELFUN is deterministic (fortunately), there are many cuts (unfortunately) [2], which are, however, not needed for obtaining the first (and only) solution, just for preventing (meaningless) attempts to search for more solutions. Using RELFUN's **relationalize** command, this **unify** function would become a relation, also runnable in PROLOG, binding an additional first argument to the result substitution.

```
unify(X,Y,Environment) :-&
        unify-ua(ultimate-assoc(X,Environment),
                 ultimate-assoc(Y,Environment),
                 Environment).

unify-ua([bnd,Xvar,Xexpr],[bnd,Yvar,Yexpr],Environment) :-
        !& unify-bnd(Xexpr,Yexpr,Xvar,Yvar,Environment).
unify-ua([bnd,Xvar,Xexpr],Y,Environment) :-
        !& unify-bnd(Xexpr,Y,Xvar,[],Environment).
unify-ua(X,[bnd,Yvar,Yexpr],Environment) :-
        !& unify-bnd(X,Yexpr,[],Yvar,Environment).
unify-ua(X,Y,Environment) :- equal(X,Y) !& Environment.
unify-ua([vari|Namel],Y,Environment) :-
        !& [[[vari|Namel],Y]|Environment].
unify-ua(X,[vari|Namel],Environment) :-
        !& [[[vari|Namel],X]|Environment].
unify-ua([dom|Delem],[exc|Eelem],Environment) :-
        !& conjn(dom-exc([dom|Delem],[exc|Eelem]),Environment).
unify-ua([exc|Eelem],[dom|Delem],Environment) :-
        !& conjn(dom-exc([dom|Delem],[exc|Eelem]),Environment).
unify-ua([dom|Xdelem],[dom|Ydelem],Environment) :-
        !& conjn(dom-intersection([dom|Xdelem],[dom|Ydelem]),
                 Environment).
unify-ua([exc|Xeelem],[exc|Yeelem],Environment) :- !& Environment.
unify-ua([dom|Delem],Y,Environment) :-
        !& conjn(membern(Y,Delem),Environment).
unify-ua(X,[dom|Delem],Environment) :-
        !& conjn(membern(X,Delem),Environment).
unify-ua([exc|Eelem],Y,Environment) :-
        !& conjn(negn(membern(Y,Eelem)),Environment).
unify-ua(X,[exc|Eelem],Environment) :-
        !& conjn(negn(membern(X,Eelem)),Environment).
unify-ua(X,Y,Environment) :- atom(X) !& [].
unify-ua(X,Y,Environment) :- atom(Y) !& [].
unify-ua([Xfirst|Xrest],[Yfirst|Yrest],Environment) :-
        ! New-environment is unify(Xfirst,Yfirst,Environment) &
        conjn(New-environment,unify-args(Xrest,Yrest,New-environment)).

unify-args([],[],Environment) :- !& Environment.
unify-args([],Y,Environment) :- !& [].
unify-args(X,[],Environment) :- !& [].
```

```
% vertical-bar treatment omitted: generate list from "|"-rest
unify-args([Xfirst|Xrest],[Yfirst|Yrest],Environment) :-
        ! New-environment is unify(Xfirst,Yfirst,Environment) &
        conjn(New-environment,unify-args(Xrest,Yrest,New-environment)).

unify-bnd([dom|Delem],[exc|Eelem],Xvar,Yvar,Environment) :-
        ! Differ is dom-exc([dom|Delem],[exc|Eelem]) &
        conjn(Differ,unify-bnd-env(Differ,Xvar,Yvar,Environment)).
unify-bnd([exc|Eelem],[dom|Delem],Xvar,Yvar,Environment) :-
        ! Differ is dom-exc([dom|Delem],[exc|Eelem]) &
        conjn(Differ,unify-bnd-env(Differ,Xvar,Yvar,Environment)).
unify-bnd([dom|Xdelem],[dom|Ydelem],Xvar,Yvar,Environment) :-
        ! Inter is dom-intersection([dom|Xdelem],[dom|Ydelem]) &
        conjn(Inter,unify-bnd-env(Inter,Xvar,Yvar,Environment)).
unify-bnd([exc|Xeelem],[exc|Yeelem],Xvar,Yvar,Environment) :-
        !& unify-bnd-env(exc-union([exc|Xeelem],[exc|Yeelem]),
                         Xvar,
                         Yvar,
                         Environment).
unify-bnd([dom|Delem],Y,Xvar,Yvar,Environment) :-
        neq([vari|Namel],Y) !&
        conjn(membern(Y,Delem),unify-bnd-env(Y,Xvar,Yvar,Environment)).
unify-bnd(X,[dom|Delem],Xvar,Yvar,Environment) :-
        neq([vari|Namel],X) !&
        conjn(membern(X,Delem),unify-bnd-env(X,Xvar,Yvar,Environment)).
unify-bnd([exc|Eelem],Y,Xvar,Yvar,Environment) :-
        neq([vari|Namel],Y) !&
        conjn(negn(membern(Y,Eelem)),
              unify-bnd-env(Y,Xvar,Yvar,Environment)).
unify-bnd(X,[exc|Eelem],Xvar,Yvar,Environment) :-
        neq([vari|Namel],X) !&
        conjn(negn(membern(X,Eelem)),
              unify-bnd-env(X,Xvar,Yvar,Environment)).
unify-bnd([vari|Namel],Y,Xvar,Yvar,Environment) :-
        ! New is unify([vari|Namel],Y,Environment) &
        conjn(New,unify-bnd-env([vari|Namel],Xvar,Yvar,New)).
unify-bnd(X,Y,Xvar,Yvar,Environment) :-
        ! New is unify(X,Y,Environment) &
        conjn(New,unify-bnd-env(Y,Xvar,Yvar,New)).

unify-bnd-env(Val,[vari|Xvarnamel],[vari|Yvarnamel],Environment) :-
        !& appfun(conjn(negn(equal([vari|Xvarnamel],[vari|Yvarnamel])),
                        [[[vari|Xvarnamel],[vari|Yvarnamel]]]),
                  [[[vari|Yvarnamel],Val]|Environment]).
unify-bnd-env(Val,Xvar,Yvar,Environment) :-
        !& appfun(appfun(conjn(Xvar,[[Xvar,Val]]),
                         conjn(Yvar,[[Yvar,Val]])),
                  Environment).
```

```
dom-intersection([dom|Xdelem],[dom|Ydelem]) :-&
        mk-dom(intersection(Xdelem,Ydelem)).
exc-union([exc|Xeelem],[exc|Yeelem]) :-& mk-exc(union(Xeelem,Yeelem)).
dom-exc([dom|Delem],[exc|Eelem]) :-& mk-dom(set-difference(Delem,Eelem)).

ultimate-assoc([vari|Namel],Environment) :-
        !& ultimate-assoc-binding([vari|Namel],
                                  assoc([vari|Namel],Environment),
                                  Environment).
ultimate-assoc(X,Environment) :- !& X.

ultimate-assoc-binding([vari|Namel],[],Environment) :- !& [vari|Namel].
ultimate-assoc-binding([vari|Namel],
                       [[vari|Namel],[dom|Delem]],
                       Environment)
     :- !& [bnd,[vari|Namel],[dom|Delem]].
ultimate-assoc-binding([vari|Namel],
                       [[vari|Namel],[exc|Eelem]],
                       Environment)
     :- !& [bnd,[vari|Namel],[exc|Eelem]].
ultimate-assoc-binding([vari|Namel],[[vari|Namel],Y],Environment) :-
        !& ultimate-assoc(Y,Environment).

mk-dom([]) :- !& [].
mk-dom([D]) :- !& D.
mk-dom([D|Ds]) :-& [dom,D|Ds].

mk-exc([]) :- !& _.
mk-exc(Eelem) :-& [exc|Eelem].

neq(X,X) :- !& false.
neq(X,Y).

negn([]) :- !.
negn(X) :-& [].

membern(E,[]) :- !& [].
membern(E,[E|Rest]) :- !& [E|Rest].
membern(X,[Y|Rest]) :-& membern(X,Rest).

assoc(N,[]) :- !& [].
assoc(N,[[N,V]|Ar]) :- !& [N,V].
assoc(N,[Af|Ar]) :-& assoc(N,Ar).

% conjn(X,Y)   acts like   if neq([],X) then Y else []
% appfun   is the normal functional append
% equal, intersection, union, set-difference   are built-ins: ground args
```

References

1. Harold Boley. A Relational/Functional Language and its Compilation into the WAM. Technical Report SEKI SR-90-05, University of Kaiserslautern, Department of Computer Science, April 1990.
2. Harold Boley. Extended Logic-plus-Functional Programming. In *Proceedings of the 2nd International Workshop on Extensions of Logic Programming, ELP '91, Stockholm 1991*, volume 596 of *LNAI*. Springer, 1992.
3. Harold Boley, Ulrich Buhrmann, and Christof Kremer. Towards a Sharable Knowledge Base on Recyclable Plastics. November 1993. To appear in: TMS'94 Symposium on Knowledge-Based Applications in Material Science and Engineering, Feb/Mar 1994, San Francisco, USA, TMS, Warrendale PA.
4. Alain Colmerauer. Introduction to Prolog III. In *ESPRIT '87*, pages 611–629. North Holland, 1987.
5. Cornelia Fischer. PANTUDE — An Anti-Unification Algorithm for Expressing Refined Generalizations. DFKI Kaiserslautern, February 1994.
6. Philipp Hanschke. A Declarative Integration of Terminological, Constraint-based, Data-driven, and Goal-directed Reasoning. Research Report RR-93-46, DFKI Kaiserslautern, October 1993.
7. Hans-Günter Hein. Propagation Techniques in WAM-based Architectures — The FIDO-III Approach. DFKI Technical Memo TM-93-04, DFKI Kaiserslautern, October 1993.
8. Joxan Jaffar and Jean-Louis Lassez. Constraint Logic Programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages (POPL), Munich, Germany*, pages 111–119. ACM, January 1987.
9. John W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, Heidelberg, New York, 1987.
10. Manfred Meyer and Jörg Müller. Solving Configuration Tasks Efficiently Using Finite Domain Consistency Techniques. *International Journal of Applied Intelligence*, 1994. To appear.
11. Gordon D. Plotkin. A Note on Inductive Generalization. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 5, pages 153–163. Elsevier North-Holland, New York, 1970.
12. Luc De Raedt. *Interactive Theory Revision - An Inductive Logic Programming Approach.* Academic Press, London, 1992.
13. Michael Sintek. FINDOM — Finite Domains in RELFUN Via Simulated Reassignment Variables. DFKI Kaiserslautern, June 1992.
14. Michael Sintek. FLIP: Functional-plus-Logic Programming on an Integrated Platform. 3rd Workshop on Functional Logic Programming, Schwarzenberg, Germany, January 1994.
15. Werner Stein. Nutzung globaler Analysetechniken in einem optimierenden Compiler für die Constraint-Logic-Programming-Sprache FIDO III. Diplomarbeit, Universität Kaiserslautern, FB Informatik, Juli 1993.
16. Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, Cambridge, Ma., 1989.