Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

Research Report
RR-92-03

Extended Logic-plus-Functional Programming

Harold Boley

January 1992

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Postfach 20 80
D-6750 Kaiserslautern, FRG
Tel.: (+49 631) 205-3211/13
Fax: (+49 631) 205-3210

Stuhlsatzenhausweg 3
D-6600 Saarbrücken 11, FRG
Tel.: (+49 681) 302-5252
Fax: (+49 681) 302-5341

Deutsches Forschungszentrum für Künstliche Intelligenz

The German Research Center for Artificial Intelligence (Deutsches Forschungszentrum für Künstliche Intelligenz, DFKI) with sites in Kaiserslautern and Saarbrücken is a non-profit organization which was founded in 1988. The shareholder companies are Atlas Elektronik, Daimler-Benz, Fraunhofer Gesellschaft, GMD, IBM, Insiders, Mannesmann-Kienzle, SEMA Group, Siemens and Siemens-Nixdorf. Research projects conducted at the DFKI are funded by the German Ministry for Research and Technology, by the shareholder companies, or by other industrial contracts.

The DFKI conducts application-oriented basic research in the field of artificial intelligence and other related subfields of computer science. The overall goal is to construct *systems with technical knowledge and common sense* which - by using AI methods - implement a problem solution for a selected application area. Currently, there are the following research areas at the DFKI:

- Intelligent Engineering Systems
- Intelligent User Interfaces
- Computer Linguistics
- Programming Systems
- Deduction and Multiagent Systems
- Document Analysis and Office Automation.

The DFKI strives at making its research results available to the scientific community. There exist many contacts to domestic and foreign research institutions, both in academy and industry. The DFKI hosts technology transfer workshops for shareholders and other interested groups in order to inform about the current state of research.

From its beginning, the DFKI has provided an attractive working environment for AI researchers from Germany and from all over the world. The goal is to have a staff of about 100 researchers at the end of the building-up phase.

Friedrich J. Wendl
Director

Extended Logic-plus-Functional Programming

Harold Boley

DFKI-RR-92-03

To appear in:

Lars-Henrik Eriksson, Lars Hallnäs, Peter Schroeder-Heister (eds.):
Proceedings of the second workshop on extensions of logic programming. ELP '91,
SICS, Stockholm, Sweden, January 1991,
Springer, Lecture Notes in Artificial Intelligence, 1992.

This work has been supported by a grant from The Federal Ministry for Research and Technology (FKZ ITW-8902 C4).

Contents

Extended Logic-plus-Functional Programming

Harold Boley
Deutsches **F**orschungszentrum für **K**ünstliche **I**ntelligenz
Box 2080, D-6750 Kaiserslautern, F. R. Germany
boley@informatik.uni-kl.de

Abstract

Extensions of logic and functional programming are integrated in RELFUN. Its valued clauses comprise Horn clauses ('true'-valued) and clauses with a distinguished 'foot' premise (returning arbitrary values). Both the logic and functional components permit LISP-like varying-arity and higher-order operators. The DATAFUN sublanguage of the functional component is shown to be preferable to relational encodings of functions in DATALOG. RELFUN permits non-ground, non-deterministic functions, hence certain functions can be inverted using an 'is'-primitive generalizing that of PROLOG. For function nestings a strict call-by-value strategy is employed. The reduction of these extensions to a relational sublanguage is discussed and their WAM compilation is sketched. Three examples ('serialise', 'wang', and 'eval') demonstrate the relational/functional style in use. The list expressions of RELFUN's LISP implementation are presented in an extended PROLOG-like syntax.

1 Introduction

Many approaches are possible for combining logic and functional programming, as illustrated by the collection [DL86]. These can be preclassified in two principal dimensions. (1) The combination may start with a model-theoretic semantics which is then refined (via proof theory) for practical programming or, it may start with an implemented operational semantics which is tuned in practice and then abstracted for model-theoretic foundation. (2) A quite separate distinction is whether one is interested in a loosly coupled hybrid system or, whether one strives for a tightly integrated logic/functional language.

With RELFUN we have been pursuing the latter alternatives of these dimensions: it is an operationally defined, highly integrated language (cf. [Bol86]).

The language's operational spirit stems from its origin as a pure-LISP-based interpreter. Also the present version is both implemented in, and can access precoded functionality from (a subset of) COMMON LISP. Besides the definitional interpreter this implementation consists of a WAM compiler/emulator system. The RELFUN-in-LISP implementation runs all the examples to be presented here, where the speed is acceptable except, understandably, for the LISP-in-RELFUN example.

RELFUN's integrating concept is valued clauses, encompassing both PROLOG-style Horn clauses (for defining relations) and directed conditional equations (for defining functions). While the former start off from Horn logic, the idea for the latter is to regard a function definition like

$$signum(x) = \begin{cases} -1 & if \quad x < 0 \\ 0 & if \quad x = 0 \\ 1 & if \quad x > 0 \end{cases}$$

not as clauses of a logic with equality (shown on the left) but as clauses that **return** the right-hand sides of the directed equations via a ("`&`"-marked) premise following after possible conditions (shown on the right):

```
eq(signum(X),-1) :- X < 0.          signum(X) :- X < 0 & -1.
eq(signum(0),0).                    signum(0) :- & 0.
eq(signum(X),1)  :- X > 0.          signum(X) :- X > 0 & 1.
```

Hence, function calls need not be embedded into `eq` calls with auxiliary request variables, as in `eq(signum(-2.7),SignumA), eq(signum(3.1),SignumB), SignumA < SignumB`, but can be written directly, as in `signum(-2.7) < signum(3.1)`. We then interpret value-returning premises (after the ampersand) as generalized Horn-rule premises: apart from being terms like `-1` they may be calls like `*(-1,X)` or `member(X,[-1,-3,-5])` and nestings likes `+(*(-1,X),3)` or `member(X,rest([-1,-3,-5]))`. Nestings are evaluated strictly call-by-value, as, classically, in FP [Bac78].

The RELFUN notions of *relation* and *function* are amalgamated to an abstract *operator* concept: functions are generalized to non-ground, non-deterministic operators, hence relations can be viewed as characteristic functions. Our notion of relations as `true`-valued functions is like in SLOG [Fri85], except that RELFUN's valued facts return `true` implicitly. Another amalgamating notion is akin to LISP's "useful non-`nil` values": relation-like operators may on success return a value more informative than `true` (e.g., we can let `member` return the list starting from the element found). All kinds of RELFUN operators can be applied in generalized Horn-rule premises, which are usable uniformly to the left as well as to the right of the "`&`"-separator. Actually, such premises constitute a *valued conjunction*, also permitted as a top-level query (e.g., `member(X,L) & member(X,M)` non-deterministically returns rest lists of `M` whose first element also occurs in `L`). A special valued conjunction calling only relations to the left of "`&`" and having a single variable to its right (e.g., `country(X), between(X,atlantic,pacific) & X`) can be viewed as an indefinite description or η-expression (e.g., $\eta(x)[country(x) \wedge between(x, atlantic, pacific)]$), also provided in other relational/functional amalgamations (see [PS91]).

Certain RELFUN functions can be inverted by calling them non-ground (by-value) on the right-hand side (rhs) of a generalized PROLOG `is`-primitive, mimicking relations

(incl. the above `eq` predicate). RELFUN thus provides a version of *innermost* conditional narowing [Fri85]. Its operational semantics *flattens* functional nestings to relational conjunctions, thus inherits the search-space reduction of SLD-resolution [BGM88]. Hence, our WAM implementation of (first-order) RELFUN can approach the speed of PROLOG [Bol90].

Besides its attempt at integrating basic notions of PROLOG and LISP, many of RELFUN's extended concepts can also be transferred to relational and functional programming individually. In the following section (2) the extended relational component will be treated, including higher-order relations. The next section (3) will then augment this by the extended functional component and discuss its benefits. Finally, the section (4) before the conclusions will give three sample uses of the relational/functional style.

2 Relations Defined by Hornish Clauses

2.1 Open-World DATALOG

First we consider DATALOG i.e., PROLOG without structures (constructor symbols applied to arguments). This kernel language of deductive databases is also a subset of RELFUN. DATALOG clauses have identical syntax[1] and equivalent semantics in PROLOG and RELFUN. Queries to RELFUN differ only as follows: they **return** the truth-value `true` instead of **printing** the answer `yes`; they signal failure by yielding the truth-value `unknown` instead of printing `no`.

When we stay in the relational realm of RELFUN this makes not much of a difference since `true` can be mapped to `yes` and `unknown` can be mapped to `no`. However, when proceeding to RELFUN's functional realm, queries will be able to return the third truth-value `false`: this is to be mapped to those of PROLOG's `no` answers for which the closed-world assumption is justified. In general, however, RELFUN does not make the closed-world assumption, and in the absence of explicit negative information modestly yields `unknown` instead of 'omnisciently' answering `no`.

For example, given the DATALOG knowledge base

```
subfield(architecture,bridgebuilding).
applicable(pharmacy,medicine).
applicable(computerscience,bridgebuilding).
applicable(computerscience,computerscience).
applicable(Tool,Field) :- subfield(Field,Sub), applicable(Tool,Sub).
```

a successful query like `applicable(computerscience,architecture)` returns `true` in RELFUN and prints `yes` in PROLOG; however, a failing query like

[1]The syntax shown for full RELFUN will continue to be PROLOG-like. In the implementation it becomes equivalent LISP-like list expressions. Although the (older) LISP-like syntax will not be shown in this paper, it is actually used more often than the (newer) PROLOG-like syntax. RELFUN has been given two syntaxes to facilitate communication between users from the LISP and PROLOG communities.

`applicable(computerscience,agriculture)` yields `unknown` in RELFUN but prints `no` in PROLOG. As with most real-life knowledge, what we know about computer-science applications is inherently open-ended; RELFUN's `unknown` reply agrees to the required open-world semantics.

Later, in DATAFUN, certain relations such as `subfield` will be reformulated as functions (cf. subsection 3.1). This will also have consequences for 'Horn' rules such as the `applicable` rule which still define a relation but call a sub**function**, e.g., in an `is`-rhs: `applicable(Tool,Field) :- Sub is subfield(Field), applicable(Tool,Sub).` To accomodate such functional (and `is`-'equational') extensions in relational rules, we speak of *hornish rules* or, generally, *hornish clauses.*

Two further extensions of DATALOG, varying-arity DATALOG and higher-order DATALOG, will be treated implicitly in the corresponding full-PROLOG extensions (see subsections 2.4 and 2.5).

2.2 PROLOG-like Structures and Lists

Let us now proceed to PROLOG with *structures* and its RELFUN extensions. PROLOG has only constructor symbols and no defined function symbols; arguments to PROLOG relations must always be (passive) structures and can never be (active) calls. RELFUN, on the other hand, does support both of these categories, hence has a notational need to distinguish between them.

First consider the more basic distinction of relations on the one hand, and constructors and defined functions on the other hand: while mathematical accounts of first-order logic express the distinction by disjoint sets of relation (predicate) and function symbols, PROLOG just distinguishes predicate (top-level) and functor (sublevel) **uses** of these symbols, and permits the same symbol to occur as a predicate and as a functor. This permits metalogical reinterpretations of certain structures as goals (via `call`).

In the same interactive-programming spirit RELFUN does not distinguish active and passive functor **symbols** but just active and passive functor **uses**. For this we note that all functor uses take the form of applications, which we write with round parentheses for 'active' operator calls and with square brackets for 'passive' structured terms. In the relational part of RELFUN this means that only top-level relation calls are written with parentheses, PROLOG-like structures are written with brackets. (In the functional part both top-level and nested function calls will be parenthesized, too.)

Consider the successor constructor `s`, often used together with `0` for specifying invertible operations on natural numbers. Thus, while in PROLOG the structure corresponding to 2 is `s(s(0))`, in RELFUN it is `s[s[0]]`. For instance, the RELFUN `lesseq` relation definition

```
lesseq(0,N).
lesseq(s[M],s[N]) :- lesseq(M,N).
```

permits the call `lesseq(X,s[s[0]])` to generate the `X`-values `0`, `s[0]`, or `s[s[0]]`.

N-element RELFUN *lists*, as in LISP and PROLOG, can be regarded as a short-hand for nested binary structures (we use the distinguished constructor "`cns`" instead of the usual "."). For example, the (non-ground) list `[s[s[0]],[E,F],s[0]]` reduces to the nesting `cns[s[s[0]],cns[cns[E,cns[F,nil]],cns[s[0],nil]]]`. A vertical bar in lists causes their `cns`-reduction to end with the element after the "`|`" (usually a variable) rather than with the distinguished constant `nil`. Thus, `[X,Y|Z]` reduces to `cns[X,cns[Y,Z]]`. This "lists-to-structures" transformation is used both for WAM compilation and mathematical formalization. Note that the `sorted` relation definition in PROLOG/RELFUN

```
sorted([]).
sorted([X]).
sorted([X,Y|Z]) :- lesseq(X,Y), sorted([Y|Z]).
```

and its `cns`-reduced form in RELFUN

```
sorted(nil).
sorted(cns[X,nil]).
sorted(cns[X,cns[Y,Z]]) :- lesseq(X,Y), sorted(cns[Y,Z]).
```

consistently employ square brackets to indicate the 'passiveness' of lists and structures, while in PROLOG the `cns`-reduced form would employ round parentheses.

2.3 Varying-Arity Structures

Lists can also be given a direct N-element interpretation because RELFUN permits *varying-arity structures* i.e., structures containing a vertical bar. Like `cns` was used as a binary list constructor we use `tup` as an N-ary list constructor ($N \geq 0$). That is, `[...]` should be regarded as an abbreviation for `tup[...]`. This convention holds even if `[...]` contains a "`|`". So, the earlier lists really stand for `tup[s[s[0]],tup[E,F],s[0]]` and `tup[X,Y|Z]`. Such `tup` structures can again be viewed as nested `cns` structures as shown for lists above.

Varying arities are also permitted for all other RELFUN constructors. This can be used for reinterprating many 'untyped' list representations as constructor-'tagged' structures. For instance, unbounded staples and dumps of elements can be written as varying-arity structures `staple[...]` and `dump[...]`, whose constructors distinguish the two 'types' of element collections. The unification of RELFUN structures containing a "`|`" generates a **list** value for a variable after the "`|`", as if the "`|`" would appear in a list context. Similarly, lists constitute the only structures to be spliced into other structures after the "`|`". Lists are thus the 'neutral' data structure for transporting the "`|`"-remainders of varying-arity structures.

For example, `staple[book,folder,folder|Rest]` represents a staple with a book followed by two folders on the top, and some unspecified remainder `Rest`. When unified with `staple[book,Y,Y,paper,Z,paper]`, `Y` is bound to `folder` and `Rest` to the list

`[paper,Z,paper]`. This list can again be spliced into, say, a dump beginning with a book, `dump[book|Rest]`, resulting in `dump[book,paper,Z,paper]`.

Unlike PROLOG we permit the vertical bar to follow directly after an opening square bracket, both in lists and in (other) structures. For any list `X`, the list `[|X]` is the same as `X`; additionally given a constructor `c`, the structure `c[|X]` exclusively uses the elements of the list `X` as its arguments. Thus with the `Rest` binding `[paper,Z,paper]`, `dump[|Rest]` is equivalent to `dump[paper,Z,paper]`.

It is now possible to define elementwise equality of staples and dumps using the facts

```
argumenteq(staple[|Args],dump[|Args]).
argumenteq(dump[|Args],staple[|Args]).
```

where the two `Args` occurrences of each fact will be bound to unifying **lists** of elements. Thus, while `staple[book,X,X]` and `dump[Y,paper,paper]` would not unify, the call `argumenteq(staple[book,X,X],dump[Y,paper,paper])` succeeds.

Another use of varying-arity structures is the term representation of clauses themselves. In PROLOG ":-" can be regarded as a binary functor whose arguments are the clause head and a nesting of binary "`,`"-conjunctions for the body; in RELFUN it is reinterpreted more concisely as an N-ary constructor ($N \geq 1$) whose first argument is the head and whose remaining arguments make up the body conjuncts. The rule of the DATALOG example in subsection 2.1 thus becomes the PROLOG structure

```
:-(applicable(Tool,Field),
   ','(subfield(Field,Subfield),applicable(Tool,Subfield)))
```

and the RELFUN structure

```
:-[applicable[Tool,Field],subfield[Field,Subfield],
                          applicable[Tool,Subfield]]
```

The use of lists to treat "`|`" in all contexts suggests a technique for reducing varying-arity structures to fixed-arity ones. Each varying-arity `c[x1,...,xN|X]` could be replaced by the unary `c[[x1,...,xN|X]]`, where the single argument is a list containing the original `c` arguments as elements. However, this naive method introduces unnecessary bracketing (which could be hidden to the user) and hinders *intrastructure WAM indexing* [Sin92] with respect to a structure's top-level arguments (which become 'neutralized' to a `tup` or `cns` constructor). Instead of listifying all `c` arguments, a 'semi-listifying' method might keep a fixed number, $K \leq N$, of initial arguments and only listify the remaining ones, resulting in the (K+1)-ary `c[x1,...,xK,[xK+1,...,xN|X]]`. However, even if global static analysis is used to find the smallest K such that a vertical bar or a closing square bracket is used after the Kth argument of `c` (for $K = 0$ leading back to the naive method), an interactive user could employ `c[a1,...,aI|R]` with $I < K$. In certain queries such a structure could be pretranslated to `c[a1,...,aI,R1,...,RJ,R*]`, with $I+J = K$, by 'unrolling' the variable `R` used after the "`|`" i.e., generating new variables `R1`, ..., `RJ` and `R*`, and on success binding `R` to `[R1,...,RJ|R*]`. In general, it is hard to avoid making possible query patterns statically known to the global analyzer.

2.4 Varying-Arity Relationships

Proceeding from constructor terms to atomic formulas, we come to the LISP-inspired PROLOG extension of *varying-arity relation applications* i.e., clause heads and bodies directly containing a "|". Thus, both structures and applications can be ended by a vertical bar followed by an ordinary variable; equivalently, they could be ended by a "sequence variable" as used in KIF [GF91]. Varying-arity applications give argument sequences the flavor of an implicit list data structure. For instance, the N-ary version (N $\geq$ 0) of the `sorted` relation

```
sorted().
sorted(X).
sorted(X,Y|Z) :- lesseq(X,Y), sorted(Y|Z).
```

permits calls like `sorted(0,W,s[s[0]],s[s[s[0]]])`, binding `W` to `0`, `s[0]`, or `s[s[0]]`.

As in LISP, the N-ary flexibility gained can be used, among other things, to flatten nestings of binary associative operators like `+` and `append`. Their output cannot go to the (usual) last argument position because of the asymmetry of "|"-list-splicing; the only uniformly usable output argument is the first one.

For example, while ordinary PROLOGs' ternary `append` relation is already quite flexible, LM-PROLOG [CK85] defines a natural N-ary extension (N > 0), which in RELFUN is rewritten as

```
append([]).
append(Total,[]|Back) :- append(Total|Back).
append([First|Total],[First|Front]|Back) :- append(Total,Front|Back).
```

It 'contains' LISP's unary `null` predicate, a `list`-typed PROLOG-like binary "=" relation, and a permuted, `list`-typed version of PROLOG's ternary `append` relation (`append(1,[],1)` won't succeed), but is actually a varying-arity relation, which can be used in surprisingly diverse ways. Two samples are `append([a,b,c],L1,...,Lm)`, splitting a given list into arbitrarily many lists, and `append([a,b,a,b,a,b],Leftcontext,[a,b,a],Rightcontext)`, unifying symmetric list segments.

Of course, a simple transformer can put the varying number of arguments of such relations into a single list. For `sorted` the additional brackets would lead back to the original definition; for `append`, with its distinguished first argument, however, they would become a syntactic burden. Also, the transformation can result in serious problems for even the standard WAM-indexing scheme because the first (and only) relation argument becomes of type `list` indiscriminately. This could be remedied by a version of the semi-listifying arity-fixing technique sketched for structures in subsection 2.3 (e.g., listifying only the N-1 input lists of `append`).

2.5 Higher-Order Constructors and Relations

While PROLOG restricts constructors and relations to constants, RELFUN also permits them to be variables or structures. This enables a restricted kind of *higher-order operators*, syntactically reducible to first-order operators, but more expressive and cleaner than PROLOG's use of extralogical builtins like `functor`, "`=..`", and metacall as higher-order substitutes. Higher-order **unification** of the kind studied with λProlog [NM90], however, is orthogonal to the extensions in RELFUN, which for simplicity and efficiency lives without λ-expressions (thus avoiding problems with λ-variables [Bac78]) and 'semantic' extensions of Robinson unification.

Constructor variables can be used to abstract from, or force equality of, the 'type' of structures, as encoded by their constructor. For example, the unification of `staple[book,X,X]` and `F[Y,paper,paper]` succeeds, binding `F` to the constructor `staple`. Also, the `argumenteq` definition of subsection 2.3 can be generalized to arbitrary constructors, using a single fact:

```
argumenteq(F[|Args],G[|Args]).
```

A converse definition, of `constructoreq`,

```
constructoreq(F[|Args1],F[|Args2]).
```

may be used to check equality of only the 'types' of two structures, as in the successful `constructoreq(staple[paper,book],staple[book,folder,X])`. For PROLOG's structures `constructoreq` could be simulated by two calls of the `functor` builtin.

Constructor structures embody parameterized constructors such as `stack[integer]`, which are themselves applicable to arguments as in `stack[integer][3,1,2]`. The above `constructoreq` fact can thus be refined to a `conspareq` definition, succeeding for equally parameterized constructor applications such as a `stack` and a `heap` of `integers`:

```
conspareq(F[Argtype][|Args1],G[Argtype][|Args2]).
```

The variables `F` and `G` stand here for constructors, e.g. `stack` and `heap`, of constructor structures, whose single `Argtype` parameters must be equal.

Relation variables in queries enable to find all relationships between given arguments. In the DATALOG knowledge base (cf. subsection 2.1) the query `R(X,bridgebuilding)` needs only fact retrieval for binding `R` to the relation `subfield` and `X` to the object `architecture` or, `R` to `applicable` and `X` to `computerscience`; the query `R(computerscience,architecture)` requires rule deduction for binding `R` to `applicable`. Later, using footed clauses (section 3), relations found in this way will become returnable values, as in `R(X,X) & self[R][X]`, returning `self[applicable][computerscience]`, where the `R`-value is part of a constructor structure. Note that the `R`'s employed here are 'relation-request' variables, free at the time of invoking the queries. More usual (mainly in LISP-based PROLOGs) is to permit variables

in relation position only if they are always bound at the time of the call, as in the example of the next paragraph.

Relation variables in clauses permit the use of higher-order facts (recognized as such by the context) like `virtue(supports)`, `virtue(protects)`, etc. to abstract rules like

```
honorable(X) :- supports(X,Y).
honorable(X) :- protects(X,Y).
```

etc. to the single rule ("Honorable is who has a virtuous relationship to someone")

```
honorable(X) :- virtue(R), R(X,Y).
```

Here we apply `virtue` as a unary second-order relation over binary relations, but more general higher-order relations can be useful.

Relation structures can be employed for defining operations on relations. For example, the relational product can be defined using the structure `relproduct[R,S]` as a relation, which permits relational square to be defined with just a fact that uses a `relproduct` structure as its second argument:

```
relproduct[R,S](X,Z) :- S(X,Y), R(Y,Z).
relsquare(R,relproduct[R,R]).
```

While the structure `relproduct[...]` can be (higher-order-)called directly, as in `relproduct[fathrel,mothrel](john,W)`, the constant `relsquare` is (first-order-)called to bind a variable, which is then used as a structure-valued relation variable, as in `relsquare(fathrel,T), T(john,W)`.

As discussed in [Bol90], higher-order relations of this form are not easily compiled into the WAM, which collects all clauses with the same **constant** relation name and arity into a procedure. However, relation variables and structures can be eliminated by simply introducing an `apply` relation constant as in [War82], which we shorten to `ap`: `hor(...)` is replaced by `ap(hor,...)` in all heads and bodies, moving the higher-order relation `hor` to the first argument position. The last example thus becomes

```
ap(relproduct[R,S],X,Z) :- ap(S,X,Y), ap(R,Y,Z).
ap(relsquare,R,relproduct[R,R]).
```

and can be queried by, e.g., `ap(relsquare,father,T), ap(T,john,W)`. Note that the `relsquare` clause and goal would not have needed the `ap` dummy because the `relsquare` relation is a constant. However, even if all calls to a relation in a program can be found to be first-order by static analysis, the user could still issue relation-variable queries like `P(R,relproduct[R,R])`. In the WAM these would only work in the form `ap(P,R,relproduct[R,R])`, and presuppose that the `relsquare` clauses are `ap`-transformed, like all other ones. Consider the effect of having all clauses collected into `ap/i` procedures, whose first arguments always are the former relation names (hence, i >

0). The discriminating effect of calling differently named procedures is lost, but is simulated by the usual first-argument indexing, loosing of course the refined discrimination of non-`ap` first-argument indexing. Fortunately, in our WAM we can index on all arguments (to the left of "`|`"), thus regaining full discriminative power for `ap`-reduced clauses.

For constructor variables and structures an analogous first-order reduction is possible using a dummy constructor, which should again be `ap` in order to permit metacalls for reduced clauses. As for earlier reductions this will affect WAM indexing: (top-level) structure's constructors are all mapped to the same dummy constant, loosing the constructors' indexing power, which could be regained by also indexing on their first arguments.

3 Functions Defined by Footed Clauses

3.1 DATAFUN as a Functional Database Language

We now proceed to functions, first considering DATAFUN, the functional subset of RELFUN corresponding to PROLOG's DATALOG subset.

3.1.1 Footed Facts and Non-Ground Functions

Let us consider the database example in [WPP77], containing the following DATALOG facts about country areas (given in thousands of square miles):

```
area(china,3380).
area(india,1139).
area(ussr, 8708).
area(usa,  3609).
```

Although these binary relations would permit requests like `area(Cntry,8708)`, their normal use direction is of the kind `area(ussr,Area)`: the large value range of possible areas makes it unlikely that a user ask for a country with a precisely given thousands-of-square-miles area such as `8708` (the problem would become even more noticeable if the exact areas were stored, perhaps as real numbers, with rounding problems etc.) Therefore[2], in our opinion this 'historical' DATALOG example should be rewritten functionally, as already implied in [GM84]. For this we extract the second argument from the DATALOG facts and use it as the so-called *foot* after a "`:-&`"-infix:

```
area(china) :-& 3380.
area(india) :-& 1139.
area(ussr)  :-& 8708.
area(usa)   :-& 3609.
```

[2]We do not make use of the argument that $area : Cntry \longrightarrow Area$ is a mapping (or 'functional' in, e.g., the relational database sense) while its inverse is not (some small countries' areas coincide if rounded to 1000 sq. mi.), because RELFUN does allow non-deterministic functions, as will be shown shortly.

The resulting special DATAFUN clauses are called *footed facts*, here used for the pointwise definition of the RELFUN function **area** mapping from country names to natural numbers. The definition emphasizes the natural **area** use direction, as in **area(ussr)**, a function call **returning** the value 8708.

The main advantage of distinguishing an 'output' argument of a relation as the returned value of a corresponding function is the possibility of *nested calls* such as

```
+(area(china),area(india),area(usa))
```

where the parenthesized inner applications are (not passive structures but) active function calls that return their values to the ternary **+** use (cf. subsection 2.2); for reasons of conciseness, program analysis, and variable elimination this is preferable to flat relational conjunctions such as

```
area(china,A1), area(india,A2), area(usa,A3), +(Area,A1,A2,A3)
```

The main disadvantage lies in the issue of *inverted calls*, which are easier and sometimes more logically complete for 'usage-neutral' relations: a functional non-termination problem is illustrated in [Fri84]. However, RELFUN's inversion method for functions appears quite natural, and for its DATAFUN subset completeness problems do not arise. A generalized form of PROLOG's **is**-primitive is employed to unify the values of a free function call with the value to be used as the argument of the inverse function, where a call is *free* if all its (actual!) arguments are different free variables. More generally, DATAFUN (RELFUN) permits *non-ground* function calls which like DATALOG (PROLOG) goals may contain repeated logical variables (non-ground terms).

As a simple example with just one free variable consider **8708 is area(Cntry)**, the inverse function call corresponding to the above-discussed relational inversion **area(Cntry,8708)**. Independently from the context (e.g., in an **is**-rhs) the free call **area(Cntry)** non-deterministically returns the values **3380**, **1139**, **8708**, or **3609**, at the same time binding **Cntry** to **china**, **india**, **ussr**, or **usa**, respectively, in the textual order of the **area** footed facts in the knowledge base. Within the above **is**-call only the third of the returned values unifies with the left-hand side (lhs), so the inversion correctly binds **Cntry** to **ussr**.

Other operators such as the **exp**onentiation relation may be hardly or impossibly inverted, which again suggests to rewrite them as 'directed' functions, leading from non-ground facts like

```
exp(X,0,1).
exp(X,1,X).
```

to *non-ground footed facts* like

```
exp(X,0) :-& 1.
exp(X,1) :-& X.
```

Here, the first clause has a ground foot, `1`, while the second one has a *non-ground foot*, `X` (in DATAFUN this must be a variable). Non-ground foots can yield both ground and non-ground values, as in `exp(2,1)`, returning `2`, and `exp(Y,1)`, returning `Y`, respectively.

3.1.2 Footed Rules and the `density` Example

In [WPP77] there are also DATALOG Horn facts about `population` (in millions), which we think should be 'functionalized' to DATAFUN footed facts as demonstrated for `area`. On this basis the paper supplies the population `density` (per square mile) of a country, using the DATALOG rule (somewhat extralogical because of the `is`-call for `D`)

```
density(C,D) :- pop(C,P), area(C,A), D is (P*1000)/A.
```

This can be mimicked by the equivalent DATAFUN rule (with `is`-calls for `P` and `A`)[3]

```
density(C)  :-  P is pop(C), A is area(C)  &  /(*(P,1000),A).
```

which may be condensed to the DATAFUN rule (without `is`-calls or auxiliary variables)

```
density(C)  :-&  /(*(pop(C),1000),area(C)).
```

Rules containing an "`&`" separator are called *footed rules*. The rule premises to the left of "`&`" are called *body premises* and act exactly like the premises of a hornish rule. The premise to the right of "`&`" is called a *foot premise* and differs from the other premises only in that its value becomes the value of the entire rule. Together, these premises form a *valued conjunction*, which like an "`&`"-less conjunction can also be used directly as a query. Footed facts are special footed rules with an empty conjunction of body premises (the separator sequence "`:- &`" is normally joined to "`:-&`") and a foot premise which just denotes a value (without evaluation). So the shortened footed `density` rule above is not a footed fact since its foot evaluates an expression. The most natural use of the DATAFUN database would be functional calls like `density(usa)`, returning the `density` value for `usa`. However, these rule formulations could also be inverted or even be called freely to enumerate all country/density pairs as in the relational call `density(Cntry,Dnsty)` (delivering both countries and their densities as bindings) or the functional call `density(Cntry)` (delivering countries as bindings with their densities as values).

To conclude the `density` example of [WPP77], PROLOG's "database query" rule

```
ans(C1,D1,C2,D2) :- density(C1,D1), density(C2,D2),
                    D1 > D2, 20*D1 < 21*D2.
```

and request `ans(C1,D1,C2,D2)` for finding countries whose population density differs by less than 5%, in RELFUN could be mimicked directly but can also be rewritten as a single valued conjunction

[3]Following LISP, RELFUN currently does not distinguish arithmetic operators as infixes, but like all other operators applies them as prefixes.

```
D1 is density(C1), D2 is density(C2), >(D1,D2), <(*(20,D1),*(21,D2)) &
                                                    ans[C1,D1,C2,D2]
```

where the auxiliary global **ans** relation transmutes to a temporary **ans** constructor.

3.1.3 Non-Determinism, DATALOG Relationalizing, and WAM Compilation

While free **calls** for the inversion of the **area** and **density** functions produce non-deterministic results, the **area** and **density** **definitions** themselves are deterministic. In RELFUN *non-deterministic function definitions* are also allowed, which return more than one value even for ground calls.

For instance, the **subfield** relation of the DATALOG example in subsection 2.1 could be extended non-deterministically, expanded by a transitive-closure version **subclosure**, and transcribed into a function definition, as in the following DATAFUN example:

```
subfield(engineering) :-& mechanics.
subfield(engineering) :-& architecture.
subfield(architecture) :-& bridgebuilding.
subclosure(Field) :-& subfield(Field).
subclosure(Field) :-& subclosure(subfield(Field)).
applicable(pharmacy,medicine).
applicable(computerscience,bridgebuilding).
applicable(computerscience,computerscience).
applicable(Tool,Field) :- applicable(Tool,subclosure(Field)).
```

In this knowledge base the ground call **subfield(engineering)** non-deterministically returns the values **mechanics** or **architecture**; finding a **subfield** path from **engineering** to **bridgebuilding**, **applicable(computerscience,engineering)** returns **true**. Note that the operator **applicable** itself is left a relation but its former Horn rule using a flat relational conjunction became a hornish rule that nests the (non-deterministic!) **subclosure** function into the recursive call. The original relational form could again be mimicked using an **is**-call, leading to

```
applicable(Tool,Field) :- Sub is subclosure(Field), applicable(Tool,Sub).
```

This *flattening* of the **applicable** definition exemplifies the first step of RELFUN's *relationalize* transformation leading from DATAFUN clauses to DATALOG clauses. The second step introduces *extra arguments* for values returned in an **is**-rhs or in the foot, where new **first** (not: last) arguments are used to cope with varying-arity DATAFUN ("|"-calls); denotative foots directly become the extra argument of the conclusion while evaluative foots generate a new variable (from _1, _2, ...) used as the extra argument of both the foot and the conclusion. Thus, the relationalized form of the above DATAFUN example is

```
subfield(mechanics,engineering).
subfield(architecture,engineering).
subfield(bridgebuilding,architecture).
subclosure(_1,Field) :- subfield(_1,Field).
subclosure(_2,Field) :- subfield(_1,Field), subclosure(_2,_1).
applicable(pharmacy,medicine).
applicable(computerscience,bridgebuilding).
applicable(computerscience,computerscience).
applicable(Tool,Field) :- subclosure(_1,Field), applicable(Tool,_1).
```

Besides this kind of RELFUN-to-PROLOG translation we have implemented a more direct WAM compilation of non-deterministic, non-ground functions [Bol90]: the WAM temporary register `X1` (identical to the argument register `A1`) is also used for passing returned values, so that first-argument nestings need not be flattened because the caller directly finds the returned value of the first callee in argument register `X1`.

3.2 Full RELFUN Exemplified by "Self"-Functions

When enriching DATAFUN with structures and lists we arrive at full RELFUN (we will immediately transfer the relational varying-arity extensions). Returning to successor structures for natural numbers, one should first note that it is illegal to nest active calls into passive structures like this: `s[+(M,N)]`. The usual equational definition of binary addition could still be transcribed by employing an `is`-call for `+`'s recursion:

```
+(0,N) :-& N.
+(s[M],N) :- A is +(M,N) & s[A].      (or   +(s[M],N) :-& +(M,s[N]).)
```

However, we prefer another method, relying on functions defined to simply return "their own call as a structure". Since the same functor can be a constructor and a defined function, we can define, e.g., `s` and `tup` as the following *self-passivating functions*:

```
s(M) :-& s[M].
tup(|Z) :-& tup[|Z].      (or   tup(|Z) :-& [|Z].   or   tup(|Z) :-& Z.)
```

Now, `s` and `tup` may also be called as active functions, evaluating their arguments in the usual call-by-value manner and returning passive structures that use the evaluated arguments as their arguments and the respective function names as their constructors.

For example, the call `tup(subfield(engineering),s(s(0)))` non-deterministically returns the lists `[mechanics,s[s[0]]]` or `[architecture,s[s[0]]]`; the LISP-`cons`-like `tup`-"`|`"-use `tup(s(0)|[0])` returns `[s[0],0]`; the COMMON LISP-`list*`-like `tup`-"`|`"-use `tup(a,b,c|[d e])` returns `[a,b,c,d,e]`. Moreover, the `s` definition enables a direct analogue to equational addition:

```
+(0,N) :-& N.
+(s[M],N) :-& s(+(M,N)).
```

It should also be noted here that RELFUN definitions obey the "constructor discipline" [O'D85], which with our notation amounts to saying simply that "clause heads must not have embedded parenthesized expressions". This would be violated by the eq-nested `signum` calls shown in the introduction.

The earlier relation-to-function transcriptions (e.g., for the `subfield` operator) decreased the arity by one because one relation argument was distinguished as the function value. Alternatively, relations can often be refined to functions of the same arity returning an additional useful value. One class of functions generated in this way is *filter functions* i.e., functions acting as the identity for certain arguments or argument combinations, and failing for other ones. For instance, the `sorted` relation on lists of subsection 2.2 can be refined to the following filter function, whose recursive call is nested after the "`|`" into a cons-like `tup` call:

```
sorted([]) :-& [].
sorted([X]) :-& [X].
sorted([X,Y|Z]) :- lesseq(X,Y) & tup(X|sorted([Y|Z])).
```

This `sorted` function returns sorted (possibly non-ground) lists like `[s[0],E,s[s[0]]]` unchanged (up to variable names), and fails for unsorted ones like `[s[s[0]],E,s[0]]`.

Below, a sample `sorted` call is given, which occurs in an (internally non-ground and non-deterministic) functional version of the well-known relational slow-sort program [Llo87]. This `sort` definition also exemplifies an essential use of non-ground function calls: since such calls both bind request variables and return a value, they can be used to split results into bindings, for the calls occurring somewhere above or after them, and a value, for the caller nested directly above them.

```
sort(X) :-& sorted(perm(X)).

perm([]) :-& [].
perm([X|Y]) :-& tup(U|perm(delete(U,[X|Y]))).

delete(X,[X|Y]) :-& Y.
delete(X,[Y|Z]) :-& tup(Y|delete(X,Z)).
```

Let us consider this bottom-up. The auxiliary function `delete` non-deterministically removes occurrences of its first argument from the list in its second argument. The `perm`utation function can then use a non-ground `delete` call for result splitting: it non-deterministically binds `U` to arbitrary list elements, for the cons-like `tup` call, and returns `U`-less lists, for the recursive `perm` call. Finally, the `sort` main function calls the above `sorted` filter on the non-deterministic `perm`utations of its argument. Note that this functional `sort` version specifies a computationally preferable (nesting) sequence by calling `perm` before `sorted`. In the relational `sort` specification commutativity of conjunction appears to permit calling `sorted` before `perm`, which, however, would not run in normal PROLOGs, as discussed in [Llo87]. A related benefit of the functional formulation is that the computationally less meaningful `sort` use for 'unsorting' a **given** sorted list is

syntactically marked by an `is`-call over a free `sort` call, whereas the relational version employs symmetrically-looking non-ground `sort` calls for both use modes, that would suggest "equality of rights".

A variant of filters is *self-testing functions*, which can also be viewed as self-passivating functions that yield `unknown` for an argument (sequence) considered "ill-formed". For example, the varying-arity `sorted` relation of subsection 2.4 can be refined to a self-testing function that fails for unsorted argument sequences:

```
sorted() :-& sorted[].
sorted(X) :-& sorted[X].
sorted(X,Y|Z) :- lesseq(X,Y), sorted[|W] is sorted(Y|Z) & sorted[X|W].
```

Now, the non-ground call `sorted(s(0),E,s(s[0]))` returns a renaming variant of `sorted[s[0],E,s[s[0]]]`, while `sorted(s(s[0]),E,s(0))` yields `unknown`.

Concluding the series of "self"-functions, let us proceed to *self-normalizing functions*, a variant of self-testing functions performing argument normalization. For instance, the previous list `sort` function can be used to define `bag` as a varying-arity function that returns a `bag` structure of the sorted arguments i.e., a normalized multiset:

```
bag(|X) :- W is sort(X) & bag[|W].
```

Now, the call `bag(s[s[0]],0,s(s[0]),s(0))` returns `bag[0,s[0],s[s[0]],s[s[0]]]`. Recalling the discussion in subsection 2.2, it should be clear that even for a defined function (e.g., `bag`) no evaluation (e.g., normalization) will happen if it is applied with square brackets: `tup(bag(s[0],0),bag[s[0],0])` returns `[bag[0,s[0]],bag[s[0],0]]`.

The flattening, extra-arguments, and relationalize transformations from DATAFUN to DATALOG in subsection 3.1.3 are easily generalized to corresponding RELFUN-to-PROLOG transformations. For example, the above varying-arity `bag` function becomes a relation which must bind normal forms to a request variable (the extra first argument) instead of just returning them: `bag(bag[|W]|X) :- sort(W,X)`. However, self-normalizing functions constitute a paradigmatic class of operators for which a relational reformulation seems not practically useful: a concise functional nesting like `set(bag(s[0],0),bag[0,s[0]])` would become the relational conjunction `bag(B,s[0],0), set(S,B,bag[0,s[0]])`, treating the active and passive `bag`s completely differently, even though they both evaluate to (equal) structures for the `set`. Again recalling subsection 3.1.3, the `X1`-reuse for value returning in the WAM also supports full RELFUN because `X1` can point to structured return values on the heap just as it points to structured variable values.

3.3 Higher-Order Constructors and Functions

Our derivation of functional programming extensions now arrives at variables and structures used as constructors or functions, and at their combination with non-ground and non-deterministic calls.

Constructor variables and structures, introduced in a relational context (subsection 2.5), are also useful in a functional setting. For instance, a function `genints` enumerates the integers in the alternating order 0, ±1, ±2, ..., returned as the infinitely nondeterministic values `0` or `s[0]` or `p[0]` or `s[s[0]]` or `p[p[0]]` or ... Its definition employs a constructor variable, `Sign`, for building up a homogeneous, 'absolute' nesting before binding the structure's "neutral signs" (equal in all levels!) to either the `s`uccessor or predecessor constructor. Instead of using the constants `s` and `p`, we could also apply as constructors the defined functions `1+` and `1-`[4] or structures like `inc[1]` and `inc[-1]`.

```
genints() :-& 0.
genints() :-& genints(Sign[0]).
genints(Sign[N]) :- Sign is s & Sign[N].
genints(Sign[N]) :- Sign is p & Sign[N].
genints(Sign[N]) :-& genints(Sign[Sign[N]]).
```

While the main **nullary** `genints/0` generates all integers, the auxiliary **unary** `genints/1` can also be called as `genints(Sign[...Sign[0]...])` to generate the integers whose absolute value is not less than the 'absolute' argument, as `genints(s[0])` to generate the positive integers, as `genints(p[0])` to generate the negative integers, and in other meaningful ways.

Function variables in queries can be utilized much like the corresponding relation variables (see subsection 2.5). For example, given the DATAFUN version of the `density` database (cf. subsection 3.1.2), the query `F(china)` asks for all unary properties of `china`, enumerating the attribute `F = area` with the returned value `3380`, the attribute `F = pop` with its value, etc.

Function variables in clauses give us the abstraction power of functional arguments in the fashion of functional programming. Thus, `revise` is a ternary function applying any unary function `F` to the Nth element of a list (for N greater than the list length or N less than 1 it returns the list unchanged):

```
revise(F,N,[]) :-& [].
revise(F,1,[H|T]) :-& tup(F(H)|T).
revise(F,N,[H|T]) :-& tup(H|revise(F,1-(N),T)).
```

Similarly, the `sort` function could be parameterized by a `Compare` relation to be handed to the `sorted` filter, which would abstract from the specific `lesseq` relation (in particular, from the representation of naturals as `s` structures). Of course, this "functional style" of universally quantified operator variables, occurring on both sides of definitions, is also useful in purely relational examples. Conversely, the "relational style" of existentially quantified operator variables, occurring only on the rhs of definitions, would also be useful

[4] RELFUN accesses a selected subset of COMMON LISP functions as builtins. Unusually named examples are the numeric successor and predecessor functions `1+` and `1-`, whose application to an argument, say `6`, in mathematical or PROLOG syntax becomes `1+(6)` and `1-(6)`, returning `7` and `5`, respectively. RELFUN's `ecal` primitive, a combination of LISP's `eval` and PROLOG's `metacall`, permits the activation of structures, as in `ecal(1-[1-[0]])`, returning `-2`.

in purely functional examples. Thus, the earlier function-variable query about `china` could be further abstracted for use in the rhs of a rule returning attribute/value pairs of an object. The below `attval` function employs both the rhs-only variable `Attribute` and an lhs/rhs variable `Valfilter` (bound, e.g., to `numfilter`) for filtering the values returned by `Attribute`:

```
attval(Obj,Valfilter) :-& tup(Attribute,Valfilter(Attribute(Obj))).
numfilter(X) :- numberp(X) & X.
```

Note that the free variable `Attribute` in the first `tup` position becomes bound by its application in the second `tup` position before the `tup` actually returns the pair.

Function structures can be employed like "function-forming operators" in FP [Bac78]. Bringing the relational-product example in subsection 2.5 back to functional programming, functional composition can be defined by using the structure `compose[F,G]` as a function, which permits `twice` to be defined as a `compose`-structure-valued footed fact:

```
compose[F,G](X) :-& F(G(X)).
twice(F) :-& compose[F,F].
```

Again, while the structure `compose[...]` can be (higher-order-)called directly, as in `compose[fathfun,mothfun](john)`, the constant `twice` is (first-order-)called in function position to return an applicable function structure, as in `twice(fathfun)(john)`.

Let us now turn to the combination of higher-order operators with non-ground and non-deterministic calls.

For example, F^{-1}, the inversion of a unary function `F`, can be defined as a function structure `inv[F]` which calls `F` freely within an `is`-call only accepting `F` values that match the argument `V` of F^{-1} (for an N-ary `F` we just add a "`|`"):

```
inv[F](V) :- V is F(X) & X.      (generally   inv[F](V) :- V is F(|X) & X.)
```

Thus, `inv[area](1139)` calls `1139 is area(X)`, hence returns `india` or other countries for which `area` returns `1139` (the general `inv` would also work, returning `[india]`).

Another example, a version of the μ-operator, additionally employs a result-splitting-like technique (used in the `sort` definition) to fork the entire result of a call into both the binding of a request variable and the returned value. First, we define a non-deterministic generator `naturals`, enumerating the naturals from an initialization given in the first argument, where the next natural is always both bound to the second argument and returned.

```
naturals(N,N) :-& N.
naturals(N,V) :-& naturals(1+(N),V).
```

For instance, `naturals(3,V)` binds `V` to `3` or `4` or ...; at the same time it returns each of these values. This then permits to concisely define a non-deterministic `mu` higher-order

function taking unary functions over the naturals as its argument and returning **their** smallest argument for which they return 0, then their second-smallest argument, etc., diverging if there is none (left[5]):

```
mu(F) :- 0 is F(naturals(0,V)) & V.
```

The ap reduction of higher-order relations in subsection 2.5 directly transfers to function variables and structures. For instance, the above rule could be reduced to

```
ap(mu,F) :- 0 is ap(F,ap(naturals,0,V)) & V.
```

changing F from a function variable into an argument variable. The effects on the WAM implementation are the same as discussed for higher-order relations.

4 The Logic/Functional Style in Use

Recent RELFUN projects have explored the use of relational/functional programming for non-toy problems: the language has been evaluated and tuned by programs for realistic tasks such as hypergraph processing [Bol92] and NC-program generation [BHH+91]. In order to facilitate comparison with other languages, this section gives versions of three well-known non-trivial programs in RELFUN's logic/functional style (some features of RELFUN, most notably higher-order operations, will not be needed in these examples).

4.1 serialise: Inplace Updates of Non-Ground Structures

After the **density** database, the second practical PROLOG example given in [WPP77] is the relational **serialise** program. Its task is to transform a list of items into a corresponding list of their alphabetic serial numbers; e.g., `[p,r,o,l,o,g]` should become `[4,5,3,2,3,1]`.

The subrelations of **serialise** demonstrate the use of the "logical" variable: first **pairlists** binds a request variable to a non-ground list of free variables (the prospective answer list), e.g. `[Y1,Y2,Y3,Y4,Y5,Y6]`, and another request variable to a corresponding list of non-ground structures, e.g. `[pair[p,Y1],pair[r,Y2],pair[o,Y3],pair[l,Y4],pair[o,Y5],pair[g,Y6]]`, thus generating two variable-coupled "incomplete data structures"; then **arrange** (quick)sorts the list of **pairs** into a binary tree, calling a **partition** relation that uses the items in the first **pair** arguments for (string) comparison; now

[5] RELFUN's **once** primitive (or "!" symbol) could be employed above (or after) the **is**-call to prevent the possibly diverging search for further solutions when the first solution is found, thus simulating the usual – unbounded – minimalization of unary functions [for (1+N)-ary functions "|" can be used to minimalize over the **first** argument, making **mu[F]** a function structure, as shown in square brackets]:

```
mu(F) :- once(0 is F(naturals(0,V))) & V.   (or mu(F) :- 0 is F(naturals(0,V)) ! & V.)
[mu[F](|Nargs) :- once(0 is F(naturals(0,V)|Nargs)) & V.]
```

numbered can count the items, left to right, at the fringe of the tree and note the resulting serial numbers by "inplace updates" in the second **pair** arguments, which by logical-variable equality instantiate **pairlists**' prospective answer list to the final, ground result.

Although this binary **serialise** relation can be called like `serialise([p,r,o,l,o,g], [4,5,3,2,3,1])`, to check the relationship, and like `serialise([p,r,o,l,o,g], S)`, to generate the list of serial numbers, it cannot be called like `serialise(I, [4,5,3,2,3,1])`, to generate all item lists mapping to given serial numbers (the comparison relation expects constant items): the main **serialise** algorithm just employs a relational syntax to express a non-invertible function from item lists to serial-number lists, while it does make an essential, "two-results" use of the subrelation **partition** and of intermediate non-ground terms.

Therefore it appears natural to reformulate **serialise** in RELFUN's non-ground functional style, keeping the **partition** relation and intermediate non-ground returned values[6]. Relationship-checking calls will then look like `[4,5,3,2,3,1] is serialise([p,r,o,l,o,g])`, an **is**-call containing a ground function call for serial-number generation as its rhs. The above explanation for the relational version can be transferred to this functional one by noting that the "principal result" is now always returned as a value instead of being bound to a request variable: the **pairlists** non-ground function only binds its prospective-answer result for use as **serialise**'s foot premise **R**, but directly returns the list of **pair**s to the **arrange** function, which again returns its non-ground tree to the first argument of the self-nested **numbered** function.

```
serialise(L) :- numbered(arrange(pairlists(L,R)),1) & R.

pairlists([X|L],[Y|R]) :-& tup(pair[X,Y]|pairlists(L,R)).
pairlists([],[]) :-& [].

arrange([X|L]) :-
    partition(L,X,L1,L2),
    T1 is arrange(L1),
    T2 is arrange(L2) &
    tree[T1,X,T2].
arrange([]) :-& void.

partition([X|L],X,L1,L2) :- partition(L,X,L1,L2).
partition([X|L],Y,[X|L1],L2) :-
    before(X,Y), partition(L,Y,L1,L2).
partition([X|L],Y,L1,[X|L2]) :-
    before(Y,X), partition(L,Y,L1,L2).
partition([],Y,[],[]).
```

[6]While the need for non-ground terms is self-evident in relational programming (ground relational programming isn't very useful), they require some justification in functional programming. The **serialise** example shows how non-ground terms can be useful internally in a computation even if its external input/output is ground terms. This is analogous to an internal use of **complex** numbers in computing **real** results.

```
before(pair[X1,Y1],pair[X2,Y2]) :- string<(X1,X2).

numbered(tree[T1,pair[X,N1],T2],N0) :-&
    numbered(T2,1+(N1 is numbered(T1,N0))).
numbered(void,N) :-& N.
```

Note that the body of the first `arrange` clause can be simplified to `partition(L,X,L1,L2) & tree(arrange(L1),X,arrange(L2))` if `tree` is defined as a self-passivating function or, similarly, if 3-`tup`s are used instead of labeled binary `tree`s (cf. subsection 3.2). Also notice that `numbered` 'updates' the `pair` structures at the roots of the `tree` structures by `is`-binding the unavoidable logical variable `N1` to the recursion result obtained from traversing the left subtree `T1`, a value which is incremented by `1+` for use in traversing the right subtree `T2`. This works since RELFUN's `is`-builtin both binds and returns the value of its rhs.

4.2 wang: On-the-Fly Construction of Proof Trees

Since its pure LISP description in [MAE+62], Wang's proof algorithm for the propositional calculus has often been reformulated to demonstrate the use of declarative languages. The algorithm applies reduction rules to a sequent representation of propositional formulas until an atomic formula occurs in both the antecedent and consequent of all derived sequents, reporting `true`, or no more rule is applicable to a sequent, reporting `false`. [PS91] gives a version with an extra relation argument for constructing a proof tree "on-the-fly", whose size can be computed by an invertible function.

Here we give a RELFUN version that returns the trees of successful proofs, where subtrees are built and their roots labeled "on-the-fly" by a constructor and two self-passivating functions: the constructor indicates that an atomic formula occurs on `both` sequent sides and the self-passivating functions exhibit the reduction of a formula on the `right` (consequent) or on the `left` (antecedent) side.

For example, `wang([],[impl[and[p,and[q,r]],and[and[p,q],r]]])` returns the and-associativity proof tree

```
right[
  impl[and[p,and[q,r]],and[and[p,q],r]],
  right[
    and[and[p,q],r],
    right[
      and[p,q],
      left[
        and[p,and[q,r]],
        left[and[q,r],both[p,wang[[r,q,p],[p]]]] ],
      left[
        and[p,and[q,r]],
        left[and[q,r],both[q,wang[[r,q,p],[q]]]] ] ]
    left[and[p,and[q,r]],left[and[q,r],both[r,wang[[r,q,p],[r]]]]] ] ]
```

The main **wang** function's first clause initializes with [] two auxiliary (atomic formula) arguments of a **work**horse function that either returns a proof tree, or yields **unknown**. In the former case, **wang** commits to the tree value by employing a 'sole' cut ("! ."), joined to "!", instead of "." as the footed-clause terminator). In the latter case the second **wang** clause returns **false**, thus implementing a procedure-specific closed-world assumption for the **wang** operator. The **work** function realizes the usual reduction rules deterministically, employing 'ankle' cuts ("! &" or "!&" instead of just a "&" separator) for committing to each rule before its foot is reached. In most **work** clauses no body premises are needed between the conclusion and the foot, hence their ankle cut coincides with a 'neck' cut (":- !" is joined to "!-", ":- !&" to "!-&")[7].

```
wang(L,R) :-& work(L,R,[],[])!  (or  wang(L,R) :-& if W is work(L,R,[],[])
wang(L,R) :-& false.                                then W else false.)
```

```
work([],[],A,B) :- member(X,A), member(X,B) !&
    both[X,wang[A,B]].
work([X|L],R,A,B) :- atomic(X) !&
    work(L,R,[X|A],B).
work(L,[X|R],A,B) :- atomic(X) !&
    work(L,R,A,[X|B]).
work(L,[not[P]|R],A,B) !-&
    right(not[P],work([P|L],R,A,B)).
work([not[P]|L],R,A,B) !-&
    left(not[P],work(L,[P|R],A,B)).
work(L,[and[P,Q]|R],A,B) !-&
    right(and[P,Q],work(L,[P|R],A,B),work(L,[Q|R],A,B)).
work([and[P,Q]|L],R,A,B) !-&
    left(and[P,Q],work([P,Q|L],R,A,B)).
work(L,[or[P,Q]|R],A,B) !-&
    right(or[P,Q],work(L,[P,Q|R],A,B)).
work([or[P,Q]|L],R,A,B) !-&
    left(or[P,Q],work([P|L],R,A,B),work([Q|L],R,A,B)).
```

[7] RELFUN only permits a single cut per clause, so premises to the left of "!" can be interpreted as the arguments of an implicit **once** operator followed by a neck cut. Also, as a "single-cut language", it is akin to a committed-choice language (CCL), obtainable by (1) restricting the left-"!" premises to 'guards' and by (2) parallelizing clause invocation. Like for PROLOG, a cut(-avoidance) discussion will be necessary for relational/functional languages. For example, **wang**'s sole cut can be encapsulated into an **if**...**then**...**else**... (as shown in parentheses), a valued version of PROLOG's ...->...;..., but this entails an **is**-variable to avoid recomputation of the entire **work**. Although some relational/functional cuts may be justified by the determinism of many functions, the question of better ways of determinism specification remains. For instance, one could **declare** the **work** procedure as deterministic in one place instead of using a "!" in each of its clauses (in the final clause, just for uniformity and CCL kinship). Note, however, that RELFUN employs "!" as **part of the clause syntax**, like CCLs use "|", not as an "extra-logical goal". In **work** this syntax acts like clause-oriented determinism annotations (for a non-neck cut also specifying a clause's "commit point"), from which a declaration for the entire procedure could be extracted.

```
work(L,[impl[P,Q]|R],A,B) !-&
    right(impl[P,Q],work([P|L],[Q|R],A,B)).
work([impl[P,Q]|L],R,A,B) !-&
    left(impl[P,Q],work([Q|L],R,A,B),work(L,[P|R],A,B)).
work(L,[equiv[P,Q]|R],A,B) !-&
    right(equiv[P,Q],work([P|L],[Q|R],A,B),work([Q|L],[P|R],A,B)).
work([equiv[P,Q]|L],R,A,B) !-&
    left(equiv[P,Q],work([P,Q|L],R,A,B),work(L,[P,Q|R],A,B)).

left(|R)  :-& left[|R].
right(|R) :-& right[|R].

member(X,[X|R]).                (or   member(X,[X|R]) :-& [X|R].)
member(X,[Y|R]) :- member(X,R).    (or   member(X,[Y|R]) :-& member(X,R).)

atomic(F[|R]) !- unknown.
atomic(X).
```

The alternative **member** definition (using both parenthesized clauses) is the LISP-like version mentioned in the introduction. (Using the first parenthesized clause and the second unparenthesized clause gives a definition returning **[X|R]** instead of **true** only for an **X** occurring as the **first** element of the **original** list.) Our functional **wang** algorithm could again be degenerated to a non-tree-building relational algorithm by just omitting **all** "**&**"-separators; the resulting hornish clauses could then be simplified, mainly by bringing the **work** recursions to the top-level.

4.3 eval: Interpreting a LISP Subset in RELFUN

Most LISP-in-LISP metainterpreters descended from the metacircular **eval/apply** specification of LISP 1.5 [MAE+62]. The operational semantics of pure LISP was later transcribed to a concise pure PROLOG **relation eval** [PP82]. The below deterministic RELFUN **function eval**, corecursive with **apply**, defines, without concern for efficiency, a non-trivial LISP subset including closures, macros, and an object-level **eval**[8].

LISP lists (and function calls) are represented as RELFUN lists (with distinguished first elements). As usual, lists with first element **lambda** are interpreted as temporary functions. Permanent functions (and macros) become relational **defun** (and **defmacro**) facts from which calls extract **lambda** functions.

For instance, **defun(ff,[x],[cond,[[atom,x],x],[t,[ff,[car,x]]]])**, asserted as

[8]We do not try here to capture the LISP subset in RELFUN which is required for our implementation of RELFUN in LISP; it would need some profane features for reading/printing etc., but could avoid the advanced features mentioned. This would provide a 'codefinition' of RELFUN and LISP, like the one proposed for PROLOG and LISP in Kenneth M. Kahn's "Pure Prolog in Pure Lisp" response (Logic Programming Newsletter 5, Winter 83/84) to the "Pure Lisp in Pure Prolog" [PP82] paper. A direct definition of RELFUN in RELFUN has been prepared by reducing it to a meaning-preserving kernel (via flattening or relationalizing), for which a PROLOG-like (vanilla) metainterpreter can be given.

a fact, can be called as in `eval([ff,[list,[cdr,[quote,[a,[[b,c],d]]]],2,3]],[])`, returning the atom b.

```
eval([],A) !-& [].
eval(t,A) !-& t.
eval(E,A) :- numberp(E) !& E.
eval(E,A) :- atom(E) ! [_,V] is assoc(E,A) & V.

eval([quote,Exp],A) !-& Exp.
eval([function,Fn],A) !-& [closure,Fn,A].

eval([cond],A) !-& [].
eval([cond,[P,Q]|R],A) :- [] is eval(P,A) !& eval([cond|R],A).
eval([cond,[P,Q]|R],A) !-& eval(Q,A).

eval([Fn|Exps],A) :-
    atom(Fn),
    defmacro(Fn,Args,Body) !&
    eval(apply([lambda,Args,Body],Exps,A),A).

eval([Fn|Exps],A) :-& apply(Fn,evlis(Exps,A),A).

apply(Fn,Vals,A) :-
    atom(Fn),
    defun(Fn,Args,Body) !&
    apply([lambda,Args,Body],Vals,A).

apply(car,[[Hd|Tl]],A) !-& Hd.
apply(cdr,[[Hd|Tl]],A) !-& Tl.
apply(cons,[Hd,Tl],A) !-& [Hd|Tl].
apply(atom,[Val],A) !-& lispatom(Val).
apply(eq,[Val1,Val2],A) !-& lispeq(Val1,Val2).

apply(add1,[Val],A) !-& 1+(Val).
apply(sub1,[Val],A) !-& 1-(Val).

apply(list,Vals,A) !-& Vals.
apply(eval,[Val],A) !-& eval(Val,A).

apply([lambda,[],Body],[],A) !-& eval(Body,A).
apply([lambda,[Arg|Rargs],Body],[Val|Rvals],A) !-&
    apply([lambda,Rargs,Body],Rvals,[[Arg,Val]|A]).

apply([closure,Fn,Env],Vals,A) !-& apply(Fn,Vals,Env).

apply(Fn,Vals,A) :-& apply(eval(Fn,A),Vals,A).
```

```
evlis([],A) :-& [].
evlis([E|Re],A) :-& tup(eval(E,A)|evlis(Re,A)).

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

lispatom([Hd|Tl]) !-& [].
lispatom(X) :-& t.

lispeq(X,X) :-& t is lispatom(X)!
lispeq(X,Y) :-& [].
```

This LISP interpreter performs more well-formedness checks than most LISP-based ones: the correct number and structure of arguments is verified by unification (e.g., **quote** should have exactly one argument), yielding **unknown** for ill-formed expressions. The **eval** function corecurses with the usual **evlis** auxiliary to evaluate actual arguments; for uniformity, even arguments of special forms and macros are submitted to **evlis** (in the final **eval** clause) iff their function is itself the result of an **eval**uation (in the final **apply** clause). The specification has no need for the usual **pairlis** auxiliary because a **lambda** application leads to an **apply** recursion through the **lambda**-argument and actual-value lists; that the **Arg/Val** pairs thus extend the environment, **A**, in reversed order does not matter for legal LISP operators, having no duplicate **lambda** variables (as usual, our interpreter does not prevent formal-argument repetitions; however, by reversing the pair order in the **A**-list it effects LISP's normal left-to-right evaluation even on **lambda** binding).

5 Conclusions

The RELFUN research attempts to combine and extend programming concepts and techniques that have accumulated in the relational (principally, PROLOG) and functional (prototypically, LISP) communities.

A comprehensive subset of PROLOG is kept as a sublanguage with little syntactic modification (structures written with square brackets instead of parentheses, cut used as a separator instead of a goal). This basis is then systematically extended by advanced relational notions, a rich set of functional notions, and a combination of both.

The functional sublanguage of RELFUN is much influenced by the implementation language LISP. But as in newer functional languages, like ML and MIRANDA, a function is defined by "pattern→action" clauses instead of a **conditional** expression. Generalizing pattern matching to unification, RELFUN permits non-ground functions, as allowed in other logic/functional integrations [DL86]. This also leads to non-deterministic functions, enumerating finitly or infinitly many values via backtracking.

The relational/functional integration entails a continuing cross-fertilization of the two language styles. For instance, relational (logical) variables are reused for enabling the non-ground function arguments and values; also, the relational (extra-logical) `once`/"`!`" is reused for making function calls/definitions deterministic. In RELFUN these constructs are employed in the same fashion for relations and functions. Conversely, varying-arity and certain higher-order operators are transferred from the functional to the relational world. Again, the cross-fertilization leads to a uniform use of such operators in both sublanguages.

In fact, some operators can play the role of both functions and relations. For example, the concise pair of clauses

```
disj[Op|Ops](|Args) :-& Op(|Args).
disj[Op|Ops](|Args) :-& disj[|Ops](|Args).
```

defines `disj` as a varying-arity, higher-order, non-ground, non-deterministic **function** structure that recursively applies its operator parameters `op1`, ... (one or more relations or functions) to zero or more (possibly non-ground) arguments `arg1,...`, enumerating the (possibly non-ground) values of `op1(arg1,...)`, ... A `disj` call fails if none of these operator calls successfully returns a value, hence we have at the same time defined a disjunction **relation** of 'success'/'failure' logic. (A cut ending the first clause would prevent functional value enumeration as well as relational truth multiplicity after the first success.)

Summarizing, RELFUN provides a tunable system of relational/functional language extensions, which can be used in isolation and in free combination. In particular, this holds for the orthogonal functional, varying-arity, higher-order, and cut extensions of the pure-PROLOG-like kernel. Several other extensions of pure PROLOG, e.g. types (incl. finite domains) and modules, being quite independent from the ones in RELFUN, could probably be added without difficulty, e.g. leading to typed values as well as arguments.

Besides the 'dynamic' interplay between our language extensions, there are 'static' reduction possibilities for several of them. Most notably, the functional sublanguage can be relationalized and the higher-order part can be reduced to the first-order part. While with these reductions RELFUN's semantics is indirectly founded on the usual Herbrand models for Horn clauses, the author is working on a more direct characterization of RELFUN's first-order hornish and footed clauses using functionally extended Herbrand models (instead of distinguishing an `equality` relation). The 'horizontal' transformations of the full language into a kernel are also important in preparation for RELFUN's 'vertical' WAM compilation. While the complete language is implemented as an interpreter, a first-order subset is realized as a compiler/emulator. The RELFUN sources are available in (a portable subset of) COMMON LISP along with the program samples of this paper.

In our hybrid expert-system shell, COLAB, RELFUN's backward rules are augmented by forward rules, taxonomies, and constraints [BHHM91]. Problems of realistic size are now being solved by RELFUN [Bol92] and RELFUN/COLAB [BHH+91] programs.

References

[Bac78] John Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *CACM*, 21(8):613–641, August 1978.

[BGM88] P.G. Bosco, E. Giovannetti, and C. Moiso. Narrowing vs. SLD-resolution. *Theoretical Computer Science*, 59:3–23, 1988.

[BHH+91] Harold Boley, Philipp Hanschke, Martin Harm, Knut Hinkelmann, Thomas Labisch, Manfred Meyer, Joerg Mueller, Thomas Oltzen, Michael Sintek, Werner Stein, and Frank Steinle. μCAD2NC: A declarative lathe-workplanning model transforming CAD-like geometries into abstract NC programs. Technical Report Document D-91-15, University of Kaiserslautern, DFKI, November 1991.

[BHHM91] H. Boley, P. Hanschke, K. Hinkelmann, and M. Meyer. COLAB: A Hybrid Knowledge Compilation Laboratory. 3rd International Workshop on Data, Expert Knowledge and Decisions: Using Knowledge to Transform Data into Information for Decision Support, September 1991.

[Bol86] Harold Boley. RELFUN: A relational/functional integration with valued clauses. *SIGPLAN Notices*, 21(12):87–98, December 1986.

[Bol90] 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.

[Bol92] Harold Boley. Declarative operations on nets. In Fritz Lehmann, editor, *Semantic Networks in Artificial Intelligence.* Special Issue of Computers & Mathematics with Applications, Pergamon Press, 1992. Preprinted as: DFKI Research Report RR-90-12, Oct. 1990.

[CK85] Mats Carlsson and Kenneth M. Kahn. LM-Prolog user manual. Technical Report UPMAIL 24, Uppsala University, Department of Computer Science, Revised April 1985.

[DL86] D. DeGroot and G. Lindstrom, editors. *Logic Programming: Functions, Relations, and Equations.* Prentice-Hall, 1986.

[Fri84] Laurent Fribourg. Oriented equational clauses as a programming language. *J. Logic Programming*, 1(2):165–177, 1984.

[Fri85] Laurent Fribourg. SLOG: A logic programming language interpreter based on clausal superposition and rewriting. In *1985 Symposium on Logic Programming*, pages 172–184. IEEE Computer Society Press, 1985.

[GF91] Michael R. Genesereth and Richard Fikes in collaboration with Danny Bobrow, Piero Bonissone, Ron Brachman, Ramanathan Guha, Reed Letsinger, Valdimir Lifschitz, Bob MacGregor, John McCarthy, Peter Norvig, Ramesh Patil, and Len Schubert. Knowledge Interchange Format Version 2.2 Reference Manual. Technical Report Logic-90-4, Stanford University, Computer Science Department, Logic Group, March 1991.

[GM84] Joseph A. Goguen and José Meseguer. Equality, types, modules, and (why not?) generics for logic programming. *J. Logic Programming*, 1(2):179–210, 1984.

[Llo87] John W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, Heidelberg, New York, 1987.

[MAE+62] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. *LISP 1.5 programmer's manual.* MIT Press, Cambridge, Mass., 1962.

[NM90] Gopalan Nadathur and Dale Miller. Higher-order Horn clauses. *JACM*, 37(4):777–814, October 1990.

[O'D85] M. J. O'Donnell. *Equational Logic as a Programming Language.* MIT Press, Cambridge, Mass., 1985.

[PP82] Luís Moniz Pereira and António Porto. Pure Lisp in pure Prolog. Logic Programming Newsletter 3, Summer 1982. Universidade Nova de Lisboa, Departamento de Informática.

[PS91] Lawrence C. Paulson and Andrew W. Smith. Logic programming, functional programming, and inductive definitions. In P. Schroeder-Heister, editor, *Extensions of Logic Programming*, pages 283–309, Berlin, Heidelberg, New York, 1991. Springer-Verlag. LNCS 475.

[Sin92] Michael Sintek. Generalized indexing methods for higher-order PROLOG extensions: A case study with the RELFUN WAM. Technical report, University of Kaiserslautern, DFKI, Forthcoming 1992.

[War82] David H. D. Warren. Higher-order extensions to PROLOG: Are they needed? *Machine Intelligence*, 10:441–454, 1982.

[WPP77] David H. D. Warren, Luis M. Pereira, and Fernando Pereira. Prolog - the language and its implementation compared with Lisp. *SIGPLAN Notices*, 12(8):109–115, August 1977. Special Issue.

Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

DFKI
-Bibliothek-
PF 2080
D-6750 Kaiserslautern
FRG

DFKI Publikationen

Die folgenden DFKI Veröffentlichungen sowie die aktuelle Liste von allen bisher erschienenen Publikationen können von der oben angegebenen Adresse bezogen werden.
Die Berichte werden, wenn nicht anders gekennzeichnet, kostenlos abgegeben.

DFKI Publications

The following DFKI publications or the list of all publisched papers so far can be ordered from the above address.
The reports are distributed free of charge except if otherwise indicated.

DFKI Research Reports

RR-90-15
Harald Trost: The Application of Two-level Morphology to Non-concatenative German Morphology
13 pages

RR-90-16
Franz Baader, Werner Nutt: Adding Homomorphisms to Commutative/Monoidal Theories, or: How Algebra Can Help in Equational Unification
25 pages

RR-90-17
Stephan Busemann:
Generalisierte Phasenstrukturgrammatiken und ihre Verwendung zur maschinellen Sprachverarbeitung
114 Seiten

RR-91-01
Franz Baader, Hans-Jürgen Bürckert, Bernhard Nebel, Werner Nutt, Gert Smolka: On the Expressivity of Feature Logics with Negation, Functional Uncertainty, and Sort Equations
20 pages

RR-91-02
Francesco Donini, Bernhard Hollunder, Maurizio Lenzerini, Alberto Marchetti Spaccamela, Daniele Nardi, Werner Nutt: The Complexity of Existential Quantification in Concept Languages
22 pages

RR-91-03
B.Hollunder, Franz Baader: Qualifying Number Restrictions in Concept Languages
34 pages

RR-91-04
Harald Trost: X2MORF: A Morphological Component Based on Augmented Two-Level Morphology
19 pages

RR-91-05
Wolfgang Wahlster, Elisabeth André, Winfried Graf, Thomas Rist: Designing Illustrated Texts: How Language Production is Influenced by Graphics Generation.
17 pages

RR-91-06
Elisabeth André, Thomas Rist: Synthesizing Illustrated Documents: A Plan-Based Approach
11 pages

RR-91-07
Günter Neumann, Wolfgang Finkler: A Head-Driven Approach to Incremental and Parallel Generation of Syntactic Structures
13 pages

RR-91-08
Wolfgang Wahlster, Elisabeth André, Som Bandyopadhyay, Winfried Graf, Thomas Rist: WIP: The Coordinated Generation of Multimodal Presentations from a Common Representation
23 pages

RR-91-09
Hans-Jürgen Bürckert, Jürgen Müller, Achim Schupeta: RATMAN and its Relation to Other Multi-Agent Testbeds
31 pages

RR-91-10
Franz Baader, Philipp Hanschke: A Scheme for Integrating Concrete Domains into Concept Languages
31 pages

RR-91-11
Bernhard Nebel: Belief Revision and Default Reasoning: Syntax-Based Approaches
37 pages

RR-91-12
J.Mark Gawron, John Nerbonne, Stanley Peters:
The Absorption Principle and E-Type Anaphora
33 pages

RR-91-13
Gert Smolka: Residuation and Guarded Rules for Constraint Logic Programming
17 pages

RR-91-14
Peter Breuer, Jürgen Müller: A Two Level Representation for Spatial Relations, Part I
27 pages

RR-91-15
Bernhard Nebel, Gert Smolka:
Attributive Description Formalisms ... and the Rest of the World
20 pages

RR-91-16
Stephan Busemann: Using Pattern-Action Rules for the Generation of GPSG Structures from Separate Semantic Representations
18 pages

RR-91-17
Andreas Dengel, Nelson M. Mattos:
The Use of Abstraction Concepts for Representing and Structuring Documents
17 pages

RR-91-18
John Nerbonne, Klaus Netter, Abdel Kader Diagne, Ludwig Dickmann, Judith Klein:
A Diagnostic Tool for German Syntax
20 pages

RR-91-19
Munindar P. Singh: On the Commitments and Precommitments of Limited Agents
15 pages

RR-91-20
Christoph Klauck, Ansgar Bernardi, Ralf Legleitner
FEAT-Rep: Representing Features in CAD/CAM
48 pages

RR-91-21
Klaus Netter: Clause Union and Verb Raising Phenomena in German
38 pages

RR-91-22
Andreas Dengel: Self-Adapting Structuring and Representation of Space
27 pages

RR-91-23
Michael Richter, Ansgar Bernardi, Christoph Klauck, Ralf Legleitner: Akquisition und Repräsentation von technischem Wissen für Planungsaufgaben im Bereich der Fertigungstechnik
24 Seiten

RR-91-24
Jochen Heinsohn: A Hybrid Approach for Modeling Uncertainty in Terminological Logics
22 pages

RR-91-25
Karin Harbusch, Wolfgang Finkler, Anne Schauder:
Incremental Syntax Generation with Tree Adjoining Grammars
16 pages

RR-91-26
M. Bauer, S. Biundo, D. Dengler, M. Hecking, J. Koehler, G. Merziger:
Integrated Plan Generation and Recognition
- A Logic-Based Approach -
17 pages

RR-91-27
A. Bernardi, H. Boley, Ph. Hanschke, K. Hinkelmann, Ch. Klauck, O. Kühn, R. Legleitner, M. Meyer, M. M. Richter, F. Schmalhofer, G. Schmidt, W. Sommer:
ARC-TEC: Acquisition, Representation and Compilation of Technical Knowledge
18 pages

RR-91-28
Rolf Backofen, Harald Trost, Hans Uszkoreit:
Linking Typed Feature Formalisms and Terminological Knowledge Representation Languages in Natural Language Front-Ends
11 pages

RR-91-29
Hans Uszkoreit: Strategies for Adding Control Information to Declarative Grammars
17 pages

RR-91-30
Dan Flickinger, John Nerbonne:
Inheritance and Complementation: A Case Study of *Easy* Adjectives and Related Nouns
39 pages

RR-91-31
H.-U. Krieger, J. Nerbonne:
Feature-Based Inheritance Networks for Computational Lexicons
11 pages

RR-91-32
Rolf Backofen, Lutz Euler, Günther Görz:
Towards the Integration of Functions, Relations and Types in an AI Programming Language
14 pages

RR-91-33
Franz Baader, Klaus Schulz:
Unification in the Union of Disjoint Equational Theories: Combining Decision Procedures
33 pages

RR-91-34
Bernhard Nebel, Christer Bäckström:
On the Computational Complexity of Temporal Projection and some related Problems
35 pages

RR-91-35
Winfried Graf, Wolfgang Maaß: Constraint-basierte Verarbeitung graphischen Wissens
14 Seiten

RR-92-03
Harold Boley:
Extended Logic-plus-Functional Programming
28 pages

RR-92-04
John Nerbonne: Feature-Based Lexicons: An Example and a Comparison to DATR
15 pages

RR-92-05
Ansgar Bernardi, Christoph Klauck, Ralf Legleitner, Michael Schulte, Rainer Stark:
Feature based Integration of CAD and CAPP
19 pages

DFKI Technical Memos

TM-91-01
Jana Köhler: Approaches to the Reuse of Plan Schemata in Planning Formalisms
52 pages

TM-91-02
Knut Hinkelmann: Bidirectional Reasoning of Horn Clause Programs: Transformation and Compilation
20 pages

TM-91-03
Otto Kühn, Marc Linster, Gabriele Schmidt:
Clamping, COKAM, KADS, and OMOS: The Construction and Operationalization of a KADS Conceptual Model
20 pages

TM-91-04
Harold Boley (Ed.):
A sampler of Relational/Functional Definitions
12 pages

TM-91-05
Jay C. Weber, Andreas Dengel , Rainer Bleisinger:
Theoretical Consideration of Goal Recognition Aspects for Understanding Information in Business Letters
10 pages

TM-91-06
Johannes Stein: Aspects of Cooperating Agents
22 pages

TM-91-08
Munindar P. Singh: Social and Psychological Commitments in Multiagent Systems
11 pages

TM-91-09
Munindar P. Singh: On the Semantics of Protocols Among Distributed Intelligent Agents
18 pages

TM-91-10
Béla Buschauer, Peter Poller, Anne Schauder, Karin Harbusch: Tree Adjoining Grammars mit Unifikation
149 pages

TM-91-11
Peter Wazinski: Generating Spatial Descriptions for Cross-modal References
21 pages

TM-91-12
Klaus Becker, Christoph Klauck, Johannes Schwagereit: FEAT-PATR: Eine Erweiterung des D-PATR zur Feature-Erkennung in CAD/CAM
33 Seiten

TM-91-13
Knut Hinkelmann:
Forward Logic Evaluation: Developing a Compiler from a Partially Evaluated Meta Interpreter
16 pages

TM-91-14
Rainer Bleisinger, Rainer Hoch, Andreas Dengel:
ODA-based modeling for document analysis
14 pages

TM-91-15
Stefan Bussmann: Prototypical Concept Formation
An Alternative Approach to Knowledge Representation
28 pages

TM-92-01
Lijuan Zhang:
Entwurf und Implementierung eines Compilers zur Transformation von Werkstückrepräsentationen
34 Seiten

DFKI Documents

D-91-01
Werner Stein , Michael Sintek: Relfun/X - An Experimental Prolog Implementation of Relfun
48 pages

D-91-02
Jörg P. Müller: Design and Implementation of a Finite Domain Constraint Logic Programming System based on PROLOG with Coroutining
127 pages

D-91-03
Harold Boley, Klaus Elsbernd, Hans-Günther Hein, Thomas Krause: RFM Manual: Compiling RELFUN into the Relational/Functional Machine
43 pages

D-91-04
DFKI Wissenschaftlich-Technischer Jahresbericht 1990
93 Seiten

D-91-06
Gerd Kamp: Entwurf, vergleichende Beschreibung und Integration eines Arbeitsplanerstellungssystems für Drehteile
130 Seiten

D-91-07
Ansgar Bernardi, Christoph Klauck, Ralf Legleitner
TEC-REP: Repräsentation von Geometrie- und Technologieinformationen
70 Seiten

D-91-08
Thomas Krause: Globale Datenflußanalyse und horizontale Compilation der relational-funktionalen Sprache RELFUN
137 Seiten

D-91-09
David Powers, Lary Reeker (Eds.):
Proceedings MLNLO'91 - Machine Learning of Natural Language and Ontology
211 pages
Note: This document is available only for a nominal charge of 25 DM (or 15 US-$).

D-91-10
Donald R. Steiner, Jürgen Müller (Eds.):
MAAMAW'91: Pre-Proceedings of the 3rd European Workshop on „Modeling Autonomous Agents and Multi-Agent Worlds"
246 pages
Note: This document is available only for a nominal charge of 25 DM (or 15 US-$).

D-91-11
*Thilo C. Horstmann:*Distributed Truth Maintenance
61 pages

D-91-12
Bernd Bachmann:
$H^{iera}C_{on}$ - a Knowledge Representation System with Typed Hierarchies and Constraints
75 pages

D-91-13
International Workshop on Terminological Logics
Organizers: Bernhard Nebel, Christof Peltason, Kai von Luck
131 pages

D-91-14
Erich Achilles, Bernhard Hollunder, Armin Laux, Jörg-Peter Mohren: KRIS : Knowledge Representation and Inference System
- Benutzerhandbuch -
28 Seiten

D-91-15
Harold Boley, Philipp Hanschke, Martin Harm, Knut Hinkelmann, Thomas Labisch, Manfred Meyer, Jörg Müller, Thomas Oltzen, Michael Sintek, Werner Stein, Frank Steinle:
µCAD2NC: A Declarative Lathe-Worplanning Model Transforming CAD-like Geometries into Abstract NC Programs
100 pages

D-91-16
Jörg Thoben, Franz Schmalhofer, Thomas Reinartz:
Wiederholungs-, Varianten- und Neuplanung bei der Fertigung rotationssymmetrischer Drehteile
134 Seiten

D-91-17
Andreas Becker:
Analyse der Planungsverfahren der KI im Hinblick auf ihre Eignung für die Abeitsplanung
86 Seiten

D-91-18
Thomas Reinartz: Definition von Problemklassen im Maschinenbau als eine Begriffsbildungsaufgabe
107 Seiten

D-91-19
Peter Wazinski: Objektlokalisation in graphischen Darstellungen
110 Seiten

Extended Logic-plus-Functional Programming

Harold Boley

RR-92-03
Research Report