A Functional-Logic Language Kernel Exemplified

Harold Boley
DFKI Kaiserslautern

Seminar Talk, IIMAS, UNAM, Mexico City
19th Apr. 1999

Structure

Context Overall View of Declarative Programming

Design I The Two Paradigms and Their Amalgamation: Quicksort Example

Design II Four Kernel Notions for the Integration: Serialization Example

Semantics Generalized Herbrand Models: Component Example

Implementation Transformation and Compilation

Outlook

Context

Overall View of Declarative Programming

Declarative Programming: Idea

Largest possible decoupling

`What` User-oriented specification

`How` Machine-oriented control

Mechanization of **`How`**
$\longrightarrow$
Focussing on **`What`**

Disadvantages: Less efficient program translation & execution

Advantages: More efficient program development & maintenance/porting

Declarative Programming: Reality

Two often **separate** *major constructs*

`Relation:` Facts & rules for queries

`Function:` (λ-)equations for applications

Thus **separate** *language paradigms (already before OOP)*

`Relational:` PROLOG,..., Gödel

`Functional:` LISP,..., Haskell

Disadvantages for many programs decreased
Advantages still disturbed by paradigm gap

Relational-Functional Integration

Hence stronger attempts to integrate the declarative paradigms, especially motivated by

- Avoidance of duplication of
  - Documentation/standardization
  - Development/maintenance/porting
  - Teaching
- **Research of common**
  - **D e s i g n   p r i n c i p l e s**
  - **Semantics: 1st/higher order**
  - **Implementation: WAM/SECD**

Subject of Many Activities

Example: ESPRIT Working Group 7232 "Common Foundations of Functional and Logic Programming" (Gentzen)

Partner Universities and Representatives:

- **Rome:** *Böhm*
- St. Andrews: *Dyckhoff*
- Athens: *Koletsos*
- Paris: *De Rougemont*
- Tübingen: *Schroeder-Heister*
- Swansea: *Hindley*
- Munich: *Leiß*
- Kista: *Eriksson*
- Kaiserslautern: *Richter*

Central Integration Possibilities

Loose ($\uplus$) and *tight* ($\cup$), *practically* and *theoretically* oriented integrations can be studied. Some examples:

PROLOG $\uplus$ LISP: LOGLISP (*Robinson*), HORNE

Horn-Logic $\uplus$ Equality: EQLOG (*Goguen*), CURRY

Backtracking $\approx$ Laziness: SASLOG, BABEL

CLP $\cup$ Committed Choice: LIFE (*Aït-Kaci*), Oz

Gödel $\cup$ Haskell $\cup$ λProlog: ESCHER (*Lloyd*)

The approach here is basically a *tight*, *practically* oriented integration:

Relation construct $\cup$ Function construct: RELFUN

Kernel-Notion Integration

In order to distinguish real integration constructs from additional ‘features’, research on **minimally necessary integrative notions** is required:

- Adaptation of basically relational notions for functions

  R1: Logic variables

  R2: Non-determinism

- Adaptation of basically functional notions for relations

  F1: Operation nestings

  F2: Higher-order operations

Design I

The Two Paradigms and Their Amalgamation: Quicksort Example

Valued Clauses

RELFUN's definition construct for minimal integration is *value-returning clauses*; these comprise

`:- .` Horn clauses, which are implicitly `true`-valued (to define relations),

`:- & .` Directed, conditional equations rewritten as clauses with an additional value-returning premise (to define functions with "`if`-part"), and

`:& .` Directed, unconditional equations, which return the value of their only premise (to define functions without "`if`-part")

Relational Paradigm

As in PROLOG, but prefix notation also for “<” etc.

Facts & Rules:

```
qsort([],[]).
qsort([X|Y], XY-sorted) :-
   partition(X,Y,Sm,Gr),
   qsort(Sm,Sm-sorted),
   qsort(Gr,Gr-sorted),
   apprel(Sm-sorted, [X|Gr-sorted], XY-sorted).

partition(X,[Y|Z],[Y|Sm],Gr) :-
   <(Y,X),  partition(X,Z,Sm,Gr).
partition(X,[Y|Z],Sm,[Y|Gr]) :-
   <(X,Y),  partition(X,Z,Sm,Gr).
partition(X,[X|Z],Sm,Gr) :- partition(X,Z,Sm,Gr).
partition(X,[],[],[]).

apprel([],L,L).
apprel([H|R],L,[H|RL]) :- apprel(R,L,RL).
```

Queries:

```
>>>>>> qsort([3,1,4,2,3],Sorted)
true
Sorted=[1,2,3,4]
>>>>>> qsort(Unsorted,[1,2,3,4])
error: free variable can't be arg to builtin
```

Functional Paradigm

$Pattern_{\mathcal{VAR}}$ `.=` $Application$ `&` $Expression_{\mathcal{VAR}}$

represents pattern-matching functional

`let(` ($Pattern_{\mathcal{VAR}}$,$Application$)`,` $Expression_{\mathcal{VAR}}$ `)`

Directed, conditional equations:

```
qsort([]) :& [].
qsort([X|Y]) :-
   seq[Sm,Gr] .= partition(X,Y) &
   appfun(qsort(Sm),tup(X|qsort(Gr))).

partition(X,[Y|Z]) :-
   <(Y,X),  seq[Sm,Gr] .= partition(X,Z) &
   seq[[Y|Sm],Gr].
partition(X,[Y|Z]) :-
   <(X,Y),  seq[Sm,Gr] .= partition(X,Z) &
   seq[Sm,[Y|Gr]].
partition(X,[X|Z]) :& partition(X,Z).
partition(X,[]) :& seq[[],[]].

appfun([],L) :& L.
appfun([H|R],L) :& tup(H|appfun(R,L)).
```

Application:

```
>>>>>> qsort([3,1,4,2,3])
[1,2,3,4]
```

Paradigm Amalgamation

As a preparatory stage for integration we now allow programs with functional-relational call alternations: the central constructs amalgamate

From functional paradigm:

`qsort` as function with a modification:

Instead of the ".="- subfunction call

```
seq[Sm,Gr] .= partition(X,Y)
```

now the subrelation call

```
partition(X,Y,Sm,Gr)
```

From relational paradigm:

`partition` as relation without modification

Directed, conditional equations (with subrelation call):

```
qsort([]) :& [].
qsort([X|Y]) :-
   partition(X,Y,Sm,Gr) &
   appfun(qsort(Sm),tup(X|qsort(Gr))).
```

Facts & Rules:

```
partition(X,[Y|Z],[Y|Sm],Gr) :-
   <(Y,X),  partition(X,Z,Sm,Gr).
partition(X,[Y|Z],Sm,[Y|Gr]) :-
   <(X,Y),  partition(X,Z,Sm,Gr).
partition(X,[X|Z],Sm,Gr) :-
            partition(X,Z,Sm,Gr).
partition(X,[],[],[]).
```

Auxiliary function:

```
appfun([],L) :& L.
appfun([H|R],L) :& tup(H|appfun(R,L)).
```

Relational-Functional Comparison

`qsort` logically **not meaningfully invertible** (permutations of a necessarily sorted list!), in implementation not at all (free variables as arguments of numerical comparisons lead to errors!).

`partition` has **two results** (lists of smaller and greater elements), which can be delivered simply via **two** logic variables `Sm` and `Gr` as argument**s**, or awkwardly via collection into **one** return value (here: structure `seq[Sm,Gr]`).

Relational

- $\ominus$ `qsort` as relation suggests non-intended computation direction
- $\oplus$ `partition` as relation simplifies the delivery of two results

Functional

- $\oplus$ `qsort` as function shows intended computation direction
- $\ominus$ `partition` as function necessitates awkward collection of two results into one return value

Amalgamated

- $\oplus$ `qsort` as function shows intended computation direction
- $\oplus$ `partition` as relation simplifies the delivery of two results

Design II

Four Kernel Notions for the Integration:

Serialization Example

R1: Logic Variables

Function calls can, like relation calls, use (free) logic variables as actual **arguments** and, additionally, return them as **values**. Likewise, non-ground terms, which contain logic variables, are permitted. The processing is based on unification

```
pairlists([],[]) :& [].
pairlists([X|L],[Y|M]) :&
  tup([X,Y]|pairlists(L,M)).
```

```
>>>>>> pairlists([d,a,l,l,a,s],R)
[[d,Y1],[a,Y2],[l,Y3],[l,Y4],[a,Y5],[s,Y6]]
R=[Y1,Y2,Y3,Y4,Y5,Y6]
```

R2: Non-Determinism

Function calls are allowed to **repeatedly return values**, just as relation calls can **repeatedly bind query variables**. Thus, uniform ("don't know") non-determinism for solution search (implemented as in PROLOG via backtracking)

Pure, ground-deterministic functions can also be called non-ground non-deterministically (e.g. *restriction-free*, with distinct logic variables). The enumeration of *computed answers* delivers pairs consisting of argument bindings and a return value (their ground instantiations for restriction-free calls constitute the *success set*):

```
pairlists([],[]) :& [].
pairlists([X|L],[Y|M]) :& tup([X,Y]|pairlists(L,M)).
```

```
>>>>>> pairlists(Q,R)

[]
Q=[]
R=[]

[[X1,Y1]]
Q=[X1]
R=[Y1]

[[X1,Y1],[X2,Y2]]
Q=[X1,X2]
R=[Y1,Y2]

[[X1,Y1],[X2,Y2],[X3,Y3]]
Q=[X1,X2,X3]
R=[Y1,Y2,Y3]
. . .
```

F1: Operation Nestings

Functions and relations return general values and truth values, resp. **Call-by-value** nestings in any combination, embedded logic variables, and embedded non-determinism:

Function-Function: `appfun(qsort(...),...)`

Relation-Relation: `or(and(true,false),true)`

Function-Relation: `tup(apprel(L,M,[a,b,c]))`

Relation-Function: `numbered(...,+(...,...))`

```
numbered([],N).
numbered([[X,N]|R],N) :- numbered(R,+(N,1)).
```

```
>>>>>> numbered([[a,Y2],[d,Y1],[l,Y3],[s,Y6]],1)
true
Y2=1,   Y1=2,   Y3=3,   Y6=4
```

F2: Higher-Order Operations

Functional and relational arguments as well as values. **Restriction** to *named* functions and relations (no λ-expressions), as they are dominant in practice and more easily integrated (avoids λ/logic-variable distinction and higher-order unification): `apply`-*reducible* to 1st order

```
qsort[Cr]([X|Y]) :-
   partition[Cr](X,Y,Sm,Gr) &
   appfun(qsort[Cr](Sm),tup(X|qsort[Cr](Gr))).

partition[Cr](X,[Y|Z],[Y|Sm],Gr) :-
   Cr(Y,X),  partition[Cr](X,Z,Sm,Gr).

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

```
>>>>>> qsort[<]([3,1,4,2,3])     % Cr bound to <
[1,2,3,4]
```

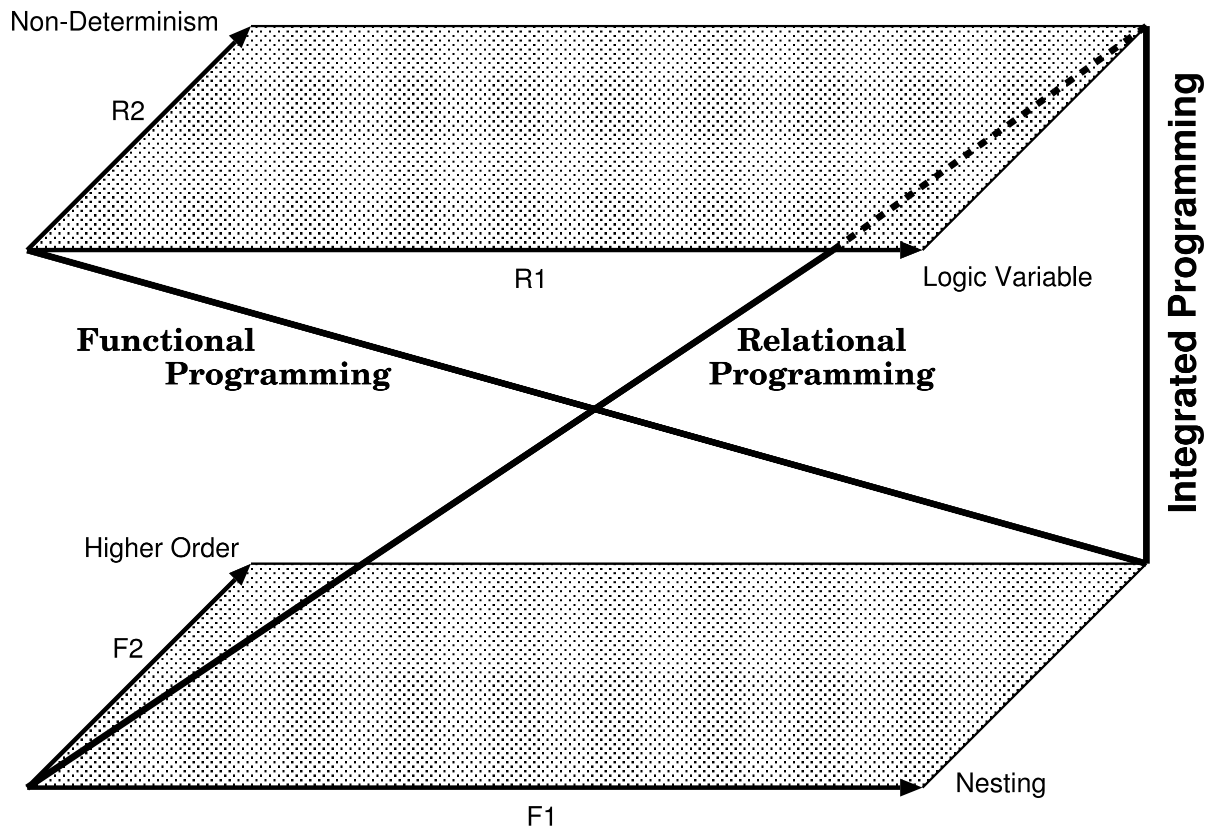
```
>>>>>> qsort[before]([[d,Y1],[a,Y2],[l,Y3],[l,Y4],[a,Y5],[s,Y6]])
[[a,Y2],[d,Y1],[l,Y3],[s,Y6]]
Y4=Y3                                      % Cr bound to before
Y5=Y2
```

```
% generic qsort via Cr parameter

qsort[Cr]([]) :& [].
qsort[Cr]([X|Y]) :-
   partition[Cr](X,Y,Sm,Gr) &
   appfun(qsort[Cr](Sm),tup(X|qsort[Cr](Gr))).

partition[Cr](X,[Y|Z],[Y|Sm],Gr) :-
   Cr(Y,X),  partition[Cr](X,Z,Sm,Gr).
partition[Cr](X,[Y|Z],Sm,[Y|Gr]) :-
   Cr(X,Y),  partition[Cr](X,Z,Sm,Gr).
partition[Cr](X,[X|Z],Sm,Gr) :-
             partition[Cr](X,Z,Sm,Gr).
partition[Cr](X,[],[],[]).

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

The Relational-Functional Kernel

- Provision of the four notions in an essential language kernel with integrated relational/functional *operator* construct:
  - RELFUN functions (general values) can use logic variables (R1) and can be non-deterministic (R2)
  - RELFUN relations (truth values) can use nestings (F1) and can be of higher order (F2)
  - Classical functions and relations become extrema in a 4-dimensional construct space

- New fundamental basis extended as a whole by finite domains, sort hierarchies, determinism specifications, etc.


Non-Determinism
R2
R1
Logic Variable
Functional Programming
Relational Programming
Integrated Programming
Higher Order
F2
Nesting
F1

Classical Relations and Functions

Let $\mathcal{Z}$ be the set of integers, $\mathcal{B}$ the set $\{true, false\}$ of truth values

The **binary relation**
$\{(0,0),(1,1),(1,-1),(2,2),(2,-2),...\} \subseteq \mathcal{Z}\mathcal{X}\mathcal{Z}$

due to its lack of right-uniqueness cannot be written as a unary function
$\mathcal{Z} \rightarrow \mathcal{Z}$

but both as a **binary characteristic function**
$pmr : \mathcal{Z}\mathcal{X}\mathcal{Z} \rightarrow \mathcal{B}$

and as a **unary function into the power set** of $\mathcal{Z}$:
$pmf : \mathcal{Z} \rightarrow \wp(\mathcal{Z})$
$pmf(x) = \{\pm x\}$ for $x \geq 0$

$pmf(0) \leadsto \{0\}$

$pmf(1) \leadsto \{1,-1\}$

$pmf(2) \leadsto \{2,-2\}$

...

RELFUN Relations and Functions

In RELFUN classical relations can be defined in both notations. Correspondences:

Characteristic function ———- relation ——— `pmr`
Power-set function – non-deterministic function – `pmf`

```
pmr(X,Y) :- >=(X,0), Y .= pm()(X).

pmf(X) :- >=(X,0) & pm()(X).


>>>>>> pmf(0)
0

>>>>>> pmf(1)
1
-1

>>>>>> pmf(2)
2
-2

...
```

The 'non-deterministic' sign "$\pm$", **presupposed** in mathematics, can be **defined** in RELFUN as parameterless, non-deterministic function `pm` of higher order, which returns the identity `id` or the sign-exchanging function `minus`:

```
pm() :& id.
pm() :& minus.
```

Serialization Program (R1,F1,F2)

Task: Transform a list of symbols into the list of their lexicographic serial numbers

Example: `[d,a,l,l,a,s]` $\leadsto$ `[2,1,3,3,1,4]`

Solution:

```
> numbered(qsort[before](pairlists([d,a,l,l,a,s],R)),1)
  & R
[2,1,3,3,1,4]
R=[2,1,3,3,1,4]
```

$\longrightarrow$ Abstraction: `[d,a,l,l,a,s] = L` $\longrightarrow$

```
serialise(L) :-
  numbered(qsort[before](pairlists(L,R)),1)
  & R.
```

Derivation of the `serialise` Solution

```
>>>>>> pairlists([d,a,l,l,a,s],R)
[[d,Y1],[a,Y2],[l,Y3],[l,Y4],[a,Y5],[s,Y6]]
R=[Y1,Y2,Y3,Y4,Y5,Y6]
```

```
>>>>>> qsort[before]([[d,Y1],[a,Y2],[l,Y3],[l,Y4],[a,Y5],[s,Y6]])
[[a,Y2],[d,Y1],[l,Y3],[s,Y6]]
Y4=Y3
Y5=Y2
```

```
>>>>>> numbered([[a,Y2],[d,Y1],[l,Y3],[s,Y6]],1)
true
Y2=1,   Y1=2,   Y3=3,   Y6=4
```

```
>>>>>> numbered(
            qsort[before](
                    pairlists([d,a,l,l,a,s],R)
                          ),
            1 )
       &  R
[2,1,3,3,1,4]
R=[2,1,3,3,1,4]
```

Novelties wrt Warren's `serialise`

- Workhorse function `qsort` separated out **reusably** (with `Cr = before`)
- Purely relational program changed into **more concise integrated** one
- Main function `serialise` specifies **intended computation direction**, since not invertible (builtin `string<` requires ground arguments)
- Subfunction `pairlists` binds free 2nd argument to list of logic variables, which become bound via occurrences in **returned** pair list
- Subrelation `numbered` with embedded "+" used because of "declarative binding side-effects". Parameterized subrelation `partition` used to deliver multiple solutions

Our `serialise` maps ground arguments deterministically to ground values, but for this essentially uses non-ground lists internally: **Utilization of logic variables in functional programming**
$\longrightarrow$ **Higher (declarative) language level and (imperative) efficiency than equivalent pure function without logic variables**

Measurements of `serialise` in the Compiler

```
serialise([q,w,e,r,t,y,u,i,o,p,a,s,d,f,g,h,j,k,l,z,x,c,v,b,n,m,
           m,n,b,v,c,x,z,l,k,j,h,g,f,d,s,a,p,o,i,u,y,t,r,e,w,q,
           q,a,z,w,s,x,e,d,c,r,f,v,t,g,b,y,h,n,u,j,m,i,k,o,l,p,
           p,l,m,o,k,n,i,j,b,u,h,v,y,g,c,t,f,x,r,d,z,e,s,a,q,w])
```

```
0.  non-ground relational (GWAM/RAWAM):  0.82 s / 0.018 s

1.  non-ground integrated (GWAM/RAWAM):  0.73 s / 0.016 s

2.  ground functional (GWAM/RAWAM):      1.03 s / 0.03 s

3.  ground functional (LL):              0.65 s
```

Semantics

Generalized Herbrand Models: Component Example

Model-Theoretic Semantics

- Herbrand model theory possible via reduction of our restricted higher-order operations to 1st-order (valued Horn clauses)

- Generalized models of integrated programs, besides undirected elements for relations (ground atoms), also contain directed elements for functions (ground molecules)

- An *atom* applies a relation to argument terms. A *molecule* pairs a function, applied to argument terms, with a value term

- The *ground atoms* in the model are exactly the *true ground queries* to the program. The *ground molecules* in the model are exactly the *ground pairs of application and value* of the program

Relational program: Herbrand(-base) model

Functional program: Herbrand-**cross model**

Integrated program: Herbrand-**crossbase model**

Material Components (R2,F1)

Relational program:

```
partr(W,W).
partr(W,P) :- mpartr(W,I),
              partr(I,P).
partr(W,P) :- apartr(W,I),
              partr(I,P).
mpartr(feconcrete,concrete).
mpartr(feconcrete,steel).
mpartr(steel,iron).
apartr(steel,carbon).
```

Functional program:

```
partf(W) :& W.
partf(W) :& partf(mpartf(W)).

partf(W) :& partf(apartf(W)).

mpartf(feconcrete) :& concrete.
mpartf(feconcrete) :& steel.
mpartf(steel) :& iron.
apartf(steel) :& carbon.
```

Least Herbrand-base model:

{
partr(feconcrete, feconcrete),
...,
partr(carbon, carbon),
partr(feconcrete, concrete),
partr(feconcrete, steel),
partr(feconcrete, iron),
partr(feconcrete, carbon),
partr(steel, iron),
partr(steel, carbon),
mpartr(feconcrete, concrete),
mpartr(feconcrete, steel),
mpartr(steel, iron),
apartr(steel, carbon)
}

Least Herbrand-cross model:

{
partf(feconcrete) :& *feconcrete*,
...,
partf(carbon) :& *carbon*,
partf(feconcrete) :& *concrete*,
partf(feconcrete) :& *steel*,
partf(feconcrete) :& *iron*,
partf(feconcrete) :& *carbon*,
partf(steel) :& *iron*,
partf(steel) :& *carbon*,
mpartf(feconcrete) :& *concrete*,
mpartf(feconcrete) :& *steel*,
mpartf(steel) :& *iron*,
apartf(steel) :& *carbon*
}

Component-example models (DB-like): **finite**
Model of `pairlists` (list processing): **infinite**

Functional program:

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

Least Herbrand-cross model:

For all x_i, y_i from the Herbrand universe:

$\{$
$pairlists([],[])$:& $[]$,
$pairlists([x_1],[y_1])$:& $[[x_1,y_1]]$,
$pairlists([x_1,x_2],[y_1,y_2])$:& $[[x_1,y_1],[x_2,y_2]]$,
$pairlists([x_1,x_2,x_3],[y_1,y_2,y_3])$:&
$[[x_1,y_1],[x_2,y_2],[x_3,y_3]]$,
...
$\}$

Equivalent to the success set of the interpreter (see R2), which basically realizes "*innermost, conditional narrowing*". *Soundness* and *completeness* shown generally

Implementation

Transformation and Compilation

Source-Text Transformers

Principle: Processing at the highest possible (source-text) level

1. Transformations between the **paradigms**
   - Between integrated, relational, and functional languages
   - In particular, from RELFUN to PROLOG and, for deterministic ground programs, to LISP
2. Transformations oriented towards the **machine** (ARC-TEC project)
   - `apply` introduction for higher-order operations
   - Flattening of nestings
   - Lists to cons pairs, etc.
3. Transformations oriented towards the **user** (VEGA project)
   - Compression/generalization by means of finite domains
   - Generation of rules from sample facts

Compilers for Abstract Machines

1. Subsequent to source-text transformations 2.

2. Declarative **clause annotations**, e.g. for temporary and permanent variables

3. Generation of instructions for an extension of Warren's Abstract Machine (WAM):

   - A **functional value** is put into **temporary register X1** prior to jumping back from clause (`put` instruction)

   - The caller can fetch it from **X1 as an argument** (`get` instruction) as if loaded by a relational `put`

4. Usual **deterministic ground programs**: Speedup by translation into functional language LL und its WAM-coupled abstract machine

Outlook

Learnability and Applications

- Learnability of the language
  - Easy with 1-year PROLOG knowledge (teaching projects with high-school pupils)
  - Possible with imperative specialization (knowledge-base course for telematics students)

- **Practical applications of the language**
  1. **Package for declarative processing of generalized graphs**
  2. **NC-program generator which classifies CAD-like geometries through production-relevant 'features', and maps these to skeletal production plans, which are specialized to NC programs by qualitative simulation**
  3. **Reusable knowledge bases for materials selection about chemical elements, engineering properties of recyclable thermoplastics/composites, and micro nutrients**

Contributions to the State of the Art

1. **Kernel-notion integration by valued clauses: R1/R2/F1/F2 system**

2. **Integrated extensions** by finite domains, sort hierarchies, and determinism specifications

3. **Transfer of logic variables to the functional paradigm** for computations with ground-I/O (encapsulated partial data structures)

4. Foundation of the semantics of **functions on the same model-theoretic level** as relations

5. Development of relational-functional **transformers, compilers, and abstract machines**

6. **Relational-functional implementation** in ANSI COMMON LISP and ANSI C, **ported from UNIX** to MAC and WINDOWS platforms

7. **Application studies on declarative programming**, mainly in engineering

8. **Reusability** of concepts, techniques, and source programs shown with COLAB (ARC-TEC) and DRL (VEGA)

Further Work

1. Unification over description lists/ψ-terms (starting interpretatively)

2. More powerful higher-order operations (cf. λProlog)

3. Denotational semantics of extended language, e.g. incl. higher-order operations

4. Utilization of existing module system and sort hierarchies for search restriction

5. More efficient compilation of deterministic programs (cf. Lisp Light)

6. AND/OR parallelism as language concept and, possibly, implementation technique

7. JAVA interface for replacing the current TCL/TK and CGI-script interfaces

8. Web-distributed development of a useful, sharable knowledge base (cf. NUTRIMINE)