Declarative Programming with Persistent Information

Michael Hanus

Christian-Albrechts-Universität Kiel
General idea:

- no coding of algorithms
- description of logical relationships
- powerful abstractions
  ➔ domain specific languages
- higher programming level
- reliable and maintainable programs
  ➔ pointer structures ➔ algebraic data types
  ➔ complex procedures ➔ comprehensible parts
  (pattern matching, local definitions)
Approach to amalgamate ideas of declarative programming

- efficient execution principles of functional languages (determinism, laziness)
- flexibility of logic languages (constraints, built-in search)
- avoid non-declarative features of Prolog (arithmetic, I/O, cut)
- combine best of both worlds in a single model
  - higher-order functions
  - declarative I/O
  - concurrent constraints
Motivation: Persistency

Functional logic languages:
- functions, expressions, lazy evaluation
- logical variables, partial data structures
- search for solutions
- concurrent constraint solving

Advantages:
- optimal evaluation strategies [JACM’00]
- new design patterns [FLOPS’02]
  (GUIs [PADL’00], dynamic web pages [PADL’01])

Not yet sufficiently covered:
- access to persistent information (e.g., databases)
- manipulation of persistent information
**MOTIVATION: PERSISTENCY**

Functional logic languages:
- functions, expressions, lazy evaluation
- logical variables, partial data structures
- search for solutions
- concurrent constraint solving

Advantages:
- optimal evaluation strategies [JACM’00]
- new design patterns [FLOPS’02]
  (GUIs [PADL’00], dynamic web pages [PADL’01])

Not yet sufficiently covered:
- access to persistent information (e.g., databases)
- manipulation of persistent information
Functional logic languages:

- functions, expressions, lazy evaluation
- logical variables, partial data structures
- search for solutions
- concurrent constraint solving

Advantages:

- optimal evaluation strategies [JACM’00]
- new design patterns [FLOPS’02]
  (GUIs [PADL’00], dynamic web pages [PADL’01])

Not yet sufficiently covered:

- access to persistent information (e.g., databases)
- manipulation of persistent information

This talk: clean approach to handle dynamic (database) predicates
EXISTING APPROACHES

Logic programming:
- externally stored relations ≈ facts defining predicates
- deductive databases
- declarative knowledge management
- no separation between access and manipulation of facts

Prolog:
- asserta/assertz: add clauses
- retract: delete clauses

Problematic in the presence of backtracking:
    p :- assertz(p), fail.

Is p provable?
EXISTING APPROACHES

Logic programming:
- externally stored relations \( \approx \) facts defining predicates
- deductive databases
- declarative knowledge management
- no separation between access and manipulation of facts

Prolog:
- `asserta/assertz`: add clauses
- `retract`: delete clauses

Problematic in the presence of backtracking:
\[
p : - \ \text{assertz}(p), \ \text{fail}.
\]

Is \( p \) provable?

[Lindholm/O'Keefe’87] No!

\( \rightsquigarrow \) logical view of database updates
Advanced control rules (e.g., coroutining):

- better control behavior (termination, efficiency) [Naish’85]
- justified by flexible selection rule of SLD-resolution
- problematic w.r.t. database updates

\[
ap(X) :- \text{assertz}(p(X)).
q :- ap(X), p(Y), X=1.
\]

Is \( q \) provable?
Advanced control rules (e.g., coroutining):

- better control behavior (termination, efficiency) [Naish’85]
- justified by flexible selection rule of SLD-resolution
- problematic w.r.t. database updates

```
ap(X) :- assertz(p(X)).
q :- ap(X), p(Y), X=1.
```

Is q provable? Yes (Y unbound!)
Advanced control rules (e.g., coroutining):

- better control behavior (termination, efficiency) [Naish’85]
- justified by flexible selection rule of SLD-resolution
- problematic w.r.t. database updates

```prolog
:- block ap(-). % delay if argument unbound
ap(X) :- assertz(p(X)).
q :- ap(X), p(Y), X=1.
```

Is q provable? Yes (Y unbound!) No!
Advanced control rules (e.g., coroutining):

→ better control behavior (termination, efficiency) [Naish’85]
→ justified by flexible selection rule of SLD-resolution
→ problematic w.r.t. database updates

:- block ap(-). % delay if argument unbound
ap(X) :- assertz(p(X)).
q :- ap(X), p(Y), X=1.

Is q provable? Yes (Y unbound!) No!

Be careful

→ with advanced control rules
→ with non-strict functional logic languages
   (demand-driven and concurrent evaluation)
DATABASE UPDATES AND ADVANCED CONTROL RULES

Advanced control rules (e.g., coroutining):

- better control behavior (termination, efficiency) [Naish’85]
- justified by flexible selection rule of SLD-resolution
- problematic w.r.t. database updates

```prolog
:- block ap(-). % delay if argument unbound 
ap(X) :- assertz(p(X)).
q :- ap(X), p(Y), X=1.
```

Is q provable? Yes (Y unbound!) No!

Be careful

- with advanced control rules
- with non-strict functional logic languages
  (demand-driven and concurrent evaluation)

Here: Solution for Curry (and similar functional logic languages)
C
URRY

[Dagstuhl’96, POPL’97]

http://www.informatik.uni-kiel.de/~curry

• declarative multi-paradigm language
  (higher-order concurrent functional logic language,
   features for high-level distributed programming)

• extension of Haskell (non-strict functional language)

• developed by an international initiative

• provide a standard for functional logic languages
  (research, teaching, application)

• several implementations available

• PAKCS (Portland Aachen Kiel Curry System):
  ➔ freely available implementation of Curry
  ➔ many libraries (GUI, HTML, XML, meta-programming,...)
  ➔ various tools (CurryDoc, CurryTest, Debuggers, Analyzers,...)
  ➔ used in various applications (e-learning, course management,...)
Values in declarative languages: **algebraic data types**

Haskell-like syntax: enumerate all data constructors

```
data Bool = True | False
data Maybe a = Nothing | Just a
data List a = [] | a : List a  -- [a]
data Tree a = Leaf a | Node [Tree a]
data Int = 0 | 1 | -1 | 2 | -2 | ...
```

**Value** \(\approx\) **data term, constructor term**: well-formed expression containing variables and data type constructors

(Just True) 1:(2:[]) [1,2] Node [Leaf 3, Node [Leaf 4, Leaf 5]]
**FUNCTIONAL LOGIC PROGRAMS**

**Functions**: operations on values defined by **equations** (or **rules**)

\[ f \; t_1 \; \ldots \; t_n \; | \; c = r \]

- **defined operation**
- **data terms**
- **condition** (optional)
- **expression**

\[
++ \; :: \; [a] \rightarrow [a] \rightarrow [a] \\
[] \quad ++ \; ys = ys \\
(x:xs) \quad ++ \; ys = x : xs ++ ys
\]

\[
last \; :: \; [a] \rightarrow a \\
last \; xs \; | \; ys ++ [x] =:= xs \; = \; x \quad \text{where} \; x,ys \; \text{free}
\]

\[
last \; [1,2] \; \sim \; 2
\]
**Expressions and Constraints**

\[
\begin{align*}
    e & ::= c \quad \text{(constants)} \\
    x & \quad \text{(variables } x) \\
    (e_0 \ e_1 \ldots e_n) & \quad \text{(application)} \\
    \backslash x \rightarrow e & \quad \text{(abstraction)} \\
    \text{if } b \text{ then } e_1 \text{ else } e_2 & \quad \text{(conditional)}
\end{align*}
\]
Expressions and Constraints

\[ e ::= c \quad \text{(constants)} \]
\[ x \quad \text{(variables } x) \]
\[ (e_0 \ e_1 \ldots e_n) \quad \text{(application)} \]
\[ \backslash x \rightarrow e \quad \text{(abstraction)} \]
\[ \text{if } b \text{ then } e_1 \text{ else } e_2 \quad \text{(conditional)} \]
\[ \text{success} \quad \text{(trivial constraint)} \]
\[ e_1 =:= e_2 \quad \text{(equational constraint)} \]
\[ e_1 \ & \ e_2 \quad \text{(concurrent conjunction)} \]
\[ \text{let } x_1, \ldots, x_n \text{ free in } e \quad \text{(existential quantification)} \]

Success: type of constraint expressions
**Expressions and Constraints**

\[
e::=\begin{align*}
&c & \text{(constants)} \\
x & \text{(variables } x) \\
(e_0 \ e_1 \ldots e_n) & \text{(application)} \\
\lambda x \rightarrow e & \text{(abstraction)} \\
\text{if } b \text{ then } e_1 \text{ else } e_2 & \text{(conditional)} \\
\text{success} & \text{(trivial constraint)} \\
e_1=:e_2 & \text{(equational constraint)} \\
e_1 \& e_2 & \text{(concurrent conjunction)} \\
\text{let } x_1,\ldots,x_n \text{ free in } e & \text{(existential quantification)}
\end{align*}
\]

**Success**: type of constraint expressions

**Equational constraints over functional expressions**:

\[
ys ++ [x] =:= [1,2] \leadsto \{ys=[1],x=2\}
\]
EXAMPLE: PROBLEM SOLVING

Dutch National Flag (Dijkstra’76): arrange a sequence of objects colored by red, white or blue so that they appear in the order of the Dutch flag.

\[
\begin{align*}
\text{Data:} & \quad \text{Color} = \text{Red} \mid \text{White} \mid \text{Blue} \\
\text{Solve:} & \quad x++[\text{White}]++y++[\text{Red}]++z = \text{solve}(x++[\text{Red}]++y++[\text{White}]++z) \\
& \quad \text{where } x, y, z \text{ free} \\
\text{Solve:} & \quad x++[\text{Blue}]++y++[\text{Red}]++z = \text{solve}(x++[\text{Red}]++y++[\text{Blue}]++z) \\
& \quad \text{where } x, y, z \text{ free} \\
\text{Solve:} & \quad x++[\text{Blue}]++y++[\text{White}]++z = \text{solve}(x++[\text{White}]++y++[\text{Blue}]++z) \\
& \quad \text{where } x, y, z \text{ free} \\
\text{Solve:} & \quad \text{uni}++[\text{Red}]++\text{uni}++[\text{White}]++\text{uni}++[\text{Blue}] = \text{flag} \\
& \quad \text{where } \text{unicolor} = [] \text{ color:unicolor}
\end{align*}
\]
EXAMPLE: PROBLEM SOLVING

Dutch National Flag (Dijkstra’76): arrange a sequence of objects colored by red, white or blue so that they appear in the order of the Dutch flag.

```
data Color = Red | White | Blue
```

**EXAMPLE: PROBLEM SOLVING**

Dutch National Flag (Dijkstra’76): arrange a sequence of objects colored by red, white or blue so that they appear in the order of the Dutch flag

```haskell
data Color = Red | White | Blue

solve flag | flag := x++[White]++y++[Red]++z
            = solve (x++[Red]++y++[White]++z) where x,y,z free
```
**Example: Problem Solving**

**Dutch National Flag** (Dijkstra’76): arrange a sequence of objects colored by red, white or blue so that they appear in the order of the Dutch flag.

\[
\text{data Color} = \text{Red} \mid \text{White} \mid \text{Blue} \\
\text{solve flag | flag} = := x++[\text{White}]++y++[\text{Red}]++z \\
\text{solve } (x++[\text{Red}]++y++[\text{White}]++z) \quad \text{where } x,y,z \text{ free} \\
\text{solve flag | flag} = := x++[\text{Blue}]++y++[\text{Red}]++z \\
\text{solve } (x++[\text{Red}]++y++[\text{Blue}]++z) \quad \text{where } x,y,z \text{ free} \\
\text{solve flag | flag} = := x++[\text{Blue}]++y++[\text{White}]++z \\
\text{solve } (x++[\text{White}]++y++[\text{Blue}]++z) \quad \text{where } x,y,z \text{ free}
\]
**Example: Problem Solving**

**Dutch National Flag** (Dijkstra’76): arrange a sequence of objects colored by red, white or blue so that they appear in the order of the Dutch flag.

```haskell
data Color = Red | White | Blue

solve flag | flag :== x++[White]++y++[Red]++z
            = solve (x++[Red]++y++[White]++z)
            where x,y,z free

solve flag | flag :== x++[Blue]++y++[Red]++z
            = solve (x++[Red]++y++[Blue]++z)
            where x,y,z free

solve flag | flag :== x++[Blue]++y++[White]++z
            = solve (x++[White]++y++[Blue]++z)
            where x,y,z free

solve flag | flag :== uni Red ++ uni White ++ uni Blue = flag
            where uni color = []
                uni color = color : uni color
```
A specification of a counter GUI:

Col [ 
  Entry [WRef \texttt{val}, Text "0", Background "yellow"],
  Row [Button (updateValue incrText \texttt{val}) [Text "Increment"],
        Button (setValue \texttt{val} "0") [Text "Reset"],
        Button exitGUI [Text "Stop"]]

where \texttt{val} free

→ layout structure \leadsto hierarchical structure, algebraic data type
→ event handlers \leadsto functions contained in layout structure
→ logical structure \leadsto dependencies in layout structure: free variables
→ free variable \texttt{val}: reference to entry widget, used in event handlers
form "Question" [htxt "Enter a string: ", textfield ref "", hr, button "Reverse string" revhandler, button "Duplicate string" duphandler]

where

ref free

revhandler env = return $ form "Answer"
[h1 [htxt ("Reversed input: " ++ rev (env ref))]]
duphandler env = return $ form "Answer"
[h1 [htxt ("Duplicated input: " ++ env ref ++ env ref)]]
**Monadic Input/Output**

I/O actions: transformations on the external world

Interactive program: sequence(!) of actions applied to external world

Type of I/O actions: \( \text{I/O } a \approx \text{World } \to (a, \text{World}) \)
**MONADIC INPUT/OUTPUT**

I/O actions: transformations on the external world

Interactive program: sequence(!) of actions applied to external world

Type of I/O actions: \( I/O a \equiv World \to (a, World) \)

Some primitive I/O actions:

- \( \text{getChar} :: I/O \ Char \) -- read character from stdin
- \( \text{putChar} :: \text{Char} \to I/O () \) -- write argument to stdout
- \( \text{return} :: a \to I/O a \) -- do nothing and return argument
**MONADIC INPUT/OUTPUT**

**I/O actions**: transformations on the external world

**Interactive program**: sequence(!) of actions applied to external world

**Type of I/O actions**: \( \text{IO} \, a \cong \text{World} \rightarrow (a, \text{World}) \)

**Some primitive I/O actions**:

- \( \text{getChar} :: \text{IO} \, \text{Char} \) -- read character from stdin
- \( \text{putChar} :: \text{Char} \rightarrow \text{IO} () \) -- write argument to stdout
- \( \text{return} :: a \rightarrow \text{IO} \, a \) -- do nothing and return argument

**Compose actions**: \( (>>=) :: \text{IO} \, a \rightarrow (a \rightarrow \text{IO} \, b) \rightarrow \text{IO} \, b \)

- \( \text{getChar} >>= \text{putChar} \): copy character from input to output

**Specialized composition**: ignore result of first action:

- \( (>>) :: \text{IO} \, a \rightarrow \text{IO} \, b \rightarrow \text{IO} \, b \)
- \( x >>= y = x >>= \_ \rightarrow y \)
Example: output action for strings (String \(\approx\) [Char])

\[
\begin{align*}
\text{putStr} & : \text{String} \rightarrow \text{IO} () \\
\text{putStr} \; [] & = \text{return} () \\
\text{putStr} \; (c:cs) & = \text{putChar} \; c \; >> \; \text{putStr} \; cs
\end{align*}
\]
**MONADIC I/O: EXAMPLES**

Example: output action for strings \((\text{String} \approx [\text{Char}])\)

\[
\text{putStr :: String} \rightarrow \text{IO ()}
\]

\[
\begin{align*}
\text{putStr} & \: [] = \text{return ()} \\
\text{putStr} & \: (c:cs) = \text{putChar c} >> \text{putStr} \: cs
\end{align*}
\]

Syntactic sugar: Haskell’s do notation

\[
\text{do } p \leftarrow a_1 \quad \approx \quad a_1 \gg= \{ p \rightarrow a_2 \}
\]

Example: read a line

\[
\text{getLine} = \text{do } c \leftarrow \text{getChar}
\]

\[
\begin{align*}
\text{if } c & \text{==’
\}
\text{then return []} \\
\text{else do } cs & \leftarrow \text{getLine}
\text{return (c:cs)}
\end{align*}
\]
Predicates (logic programming) \( \approx \) functions with result type \( \text{Success} \)

\[
\begin{align*}
\text{isPrime} &:: \text{Int} \to \text{Success} \\
\text{isPrime 2} &= \text{success} \\
\text{isPrime 3} &= \text{success} \\
\text{isPrime 5} &= \text{success} \\
\text{isPrime 7} &= \text{success} \\
\text{isPrimePair} &:: \text{Int} \to \text{Int} \to \text{Success} \\
\text{isPrimePair x y} &= \text{isPrime x} \& \text{isPrime y} \& x+2 =:= y
\end{align*}
\]

Pure logic programs \( \sim \) direct translation into Curry programs
Dynamic predicate:

- semantics defined by ground facts
- facts not provided in program code
- only type signature provided (similar to external functions)
**Dynamic Predicates: General Concept**

**Dynamic predicate:**

- semantics defined by ground facts
- facts not provided in program code
- only type signature provided (similar to external functions)

```haskell
prime :: Int -> Dynamic -- instead of Success
prime dynamic -- instead of explicit rules
```
Dynamic Predicates: General Concept

**Dynamic predicate:**
- semantics defined by ground facts
- facts not provided in program code
- only type signature provided (similar to external functions)

```
prime :: Int -> Dynamic  -- instead of Success
prime dynamic          -- instead of explicit rules
```

**Dynamic:**
- abstract type (≈ Success)
- specific update and access functions
**DYNAMIC PREDICATES: GENERAL CONCEPT**

**Dynamic predicate:**
- semantics defined by ground facts
- facts not provided in program code
- only type signature provided (similar to external functions)

- prime :: Int -> Dynamic -- instead of Success
- prime dynamic -- instead of explicit rules

**Dynamic:**
- abstract type (≈ Success)
- specific update and access functions

- assert :: Dynamic -> IO () -- add new fact
- retract :: Dynamic -> IO Bool -- try to delete fact
- getKnowledge :: IO (Dynamic->Success) -- get current facts
assert :: Dynamic -> IO ()  -- add new fact
retract :: Dynamic -> IO Bool  -- try to delete fact

assert (prime 1) >> assert (prime 2) >> retract (prime 1)
\rightarrow asserts (prime 2) to database
assert :: Dynamic -> IO () -- add new fact
retract :: Dynamic -> IO Bool -- try to delete fact

assert (prime 1) >> assert (prime 2) >> retract (prime 1)
∽ asserts (prime 2) to database

getKnowledge :: IO (Dynamic->Success) -- get current facts

Retrieve set of currently stored facts:

do assert (prime 2)
  known <- getKnowledge
doSolve (known (prime x)) -- doSolve c | c = return ()
∽ {x=2}
ENCAPSULATING NON-DETERMINISM

Note: I/O actions must be deterministic ("cannot copy the world")

encapsulate non-deterministic search in I/O actions
ENCAPSULATING NON-DETERMINISM

Note: I/O actions must be deterministic ("cannot copy the world")

\( \rightsquigarrow \) encapsulate non-deterministic search in I/O actions

\[
\text{getAllSolutions} :: (a \to \text{Success}) \to \text{IO} \ [a]
\]

returns list of all solutions for constraint abstraction

\[
\text{getAllSolutions} (\backslash x \to \text{known (prime x)}) \ \rightsquigarrow \text{all known primes}
\]
**ENCAPSULATING NON-DETERMINISM**

Note: I/O actions must be deterministic ("cannot copy the world")

\[ \sim \text{encapsulate non-deterministic search in I/O actions} \]

```haskell
getAllSolutions :: (a -> Success) -> IO [a]
```

returns list of all solutions for constraint abstraction

```haskell
getAllSolutions (\x -> known (prime x)) \sim \text{all known primes}
```

Print list of all known primes:

```haskell
printKnownPrimes = do
    known <- getKnowledge
    primes <- getAllSolutions (\x -> known (prime x))
    print primes
```
**Logic Programming with Dynamic Predicates**

General technique:
- pass result of `getKnowledge` into deductive part
- wrap all calls to dynamic predicate

Print all prime pairs:

```haskell
printPrimePairs = do
  known <- getKnowledge
  ppairs <- getAllSolutions (\p -> primePair known p)
  print ppairs

primePair known (x,y) =
  known (prime x) & known (prime y) & x+2 =:= y
```
An even more logic programming style:

- pass result of `getKnowledge` into deductive part
- define composition of knowledge and dynamic predicate

Define sequence of primes:

```
primeSequence known l = primes l
where
  isPrime = known . prime

primes [p] = isPrime p
primes (p1:p2:ps) = isPrime p1 &
  isPrime p2 &
  (p1<p2)=:=True &
  primes (p2:ps)
```
COMBINING UPDATES AND ACCESSES

Clear separation between update and access
independent of computation order:

\[
\text{do assert (prime 2)} \\
\text{known1 \leftarrow getKnowledge -- should be [2]} \\
\text{assert (prime 3)} \\
\text{assert (prime 5)} \\
\text{known2 \leftarrow getKnowledge -- should be [2,3,5]} \\
\text{sols2 \leftarrow getAllSolutions (x \rightarrow known2 (prime x))} \\
\text{sols1 \leftarrow getAllSolutions (x \rightarrow known1 (prime x))} \\
\text{return (sols1,sols2) \rightarrow ([2],[2,3,5])}
\]

Computation (\text{getAllSolutions}) later than access (\text{getKnowledge})

\rightarrow \text{getKnowledge conceptually copies current database}
\rightarrow \text{efficiently implemented by time stamps}
Real applications require persistent data

→ survive program executions (or crashes)
→ store in (XML) files or databases
→ complex access/update routines
Real applications require persistent data

- survive program executions (or crashes)
- store in (XML) files or databases
- complex access/update routines

Our approach: declare dynamic predicate as persistent (nothing else!)

```prime :: Int -> Dynamic
prime persistent "file:prime_infos" -- instead of dynamic```

Consequences:

1. all facts are persistently stored
2. changes immediately written into log file (recovered after restart/crash)
3. `getKnowledge` gets current persistently stored knowledge (e.g., changes by other processes)
Problem with persistent data: changes by concurrent processes

- synchronization necessary
- database community: transactions
Problem with persistent data: changes by concurrent processes

- synchronization necessary
- database community: transactions

**Transaction**: updates completely performed or ignored (error/failure)

(only complete transactions visible to other processes)

```
transaction :: IO a -> IO (Maybe a)
abortTransaction :: IO a  -- failure of transaction
```
Problem with persistent data: changes by concurrent processes

- synchronization necessary
- database community: transactions

**Transaction**: updates completely performed or ignored (error/failure)
(only complete transactions visible to other processes)

```haskell
transaction :: IO a -> IO (Maybe a)
abortTransaction :: IO a  -- failure of transaction

try42 = do
    assert (prime 42)
    abortTransaction
    assert (prime 43)

transaction try42  \(\sim\) Nothing (no change to prime)
```
Dynamic predicates implemented in PAKCS (Curry→Prolog):

- dynamic predicate ≈ data structure (actual arguments, file name)
- facts stored in main memory
- assert/retract ≈ Prolog’s assert/retract
- facts with time stamps [birth, death]
Dynamic predicates implemented in PAKCS (Curry→Prolog):

- dynamic predicate ≈ data structure (actual arguments, file name)
- facts stored in main memory
- assert/retract ≈ Prolog’s assert/retract
- facts with time stamps [birth,death]

Current time (CT): incremented for each assert/retract

- assert  ~→ time stamp [CT,∞]
- retract  ~→ set death time to CT
- getKnowledge  ~→ keep CT and check time stamp of unifiable facts
Dynamic predicates implemented in PAKCS (Curry→Prolog):

- dynamic predicate ≈ data structure (actual arguments, file name)
- facts stored in main memory
- assert/retract ≈ Prolog’s assert/retract
- facts with time stamps [birth, death]

Current time (CT): incremented for each assert/retract
assert  ~  time stamp [CT, ∞]
retract ~  set death time to CT
getKnowledge ~  keep CT and check time stamp of unifiable facts

Persistent predicates:

- all facts stored in main memory and Prolog file
- each update written into log file
- program initialization: merge log file into Prolog file
  (exclusive by one process with OS locks)
- reduce load time: store facts in intermediate format (Sicstus-Prolog “.po”)
Transactions and concurrent access:

- operating system locks
- version numbers for database (concurrent updates)
- mark log files with transactions (ignore incomplete transactions)
Transactions and concurrent access:

- operating system locks
- version numbers for database (concurrent updates)
- mark log files with transactions (ignore incomplete transactions)

Preliminary results:

Experiment: bibliographic database with 10,000 entries

- machine: 2.0 GHz Linux-PC (AMD Athlon XP 2600)
- load time (for 12.5 MB Prolog source code): 120 msec
- query time: few milliseconds

Current implementation used in a larger application (SOL - web-based test and examination system)
Dynamic predicates:

- defined by facts
- updates and access initialization as I/O actions
- actual access controlled by time stamps (independence of evaluation time!)
- easy to use: only three basic I/O actions
- supports
  - logic programming style
  - persistence
  - concurrency and transactions

Future work: relational database instead of files
(first implementation with MySQL just finished)

Available with latest PAKCS release:
http://www.informatik.uni-kiel.de/~pakcs/