OSU, May 16, 2005

Declarative Programming
with
Persistent Information

Michael Hanus

Christian-Albrechts-Universität Kiel

DECLARATIVE PROGRAMMING

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)

FUNCTIONAL LOGIC LANGUAGES

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])

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

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 ≈ 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?

[Lindholm/O’Keefe’87] No!

$\rightsquigarrow$ logical view of database updates

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

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

Is `q` provable?

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

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

Is `q` provable? Yes (`Y` unbound!)

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

```
:- 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!

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

```
:- 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

```
:- 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)

CURRY

[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 CURRY

Values in declarative languages: **algebraic data types**

Haskell-like syntax: enumerate all data constructors

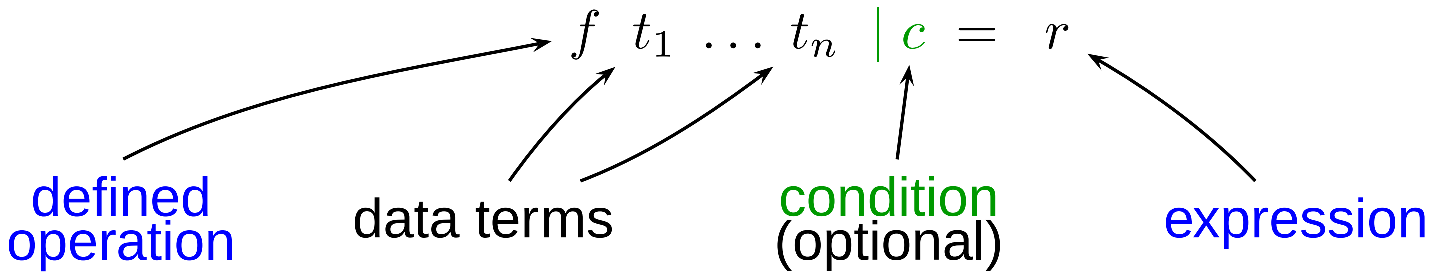
```
data Bool     =  True     |  False
data Maybe a  =  Nothing  |  Just a
data List a   =  []       |  a : List a       -- [a]
data Tree a   =  Leaf a   |  Node [Tree a]

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)



$$f\ \ t_1\ \dots\ t_n\ \mid c\ =\ r$$

```
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : xs ++ ys

last :: [a] -> a
last xs | ys ++ [x] =:= xs
        = x                  where x,ys free
```

`last [1,2]` $\leadsto$ `2`

EXPRESSIONS AND CONSTRAINTS

e	$::=$	c	(constants)
		x	(variables x)
		$(e_0\ \ e_1 \ldots e_n)$	(application)
		$\backslash x\ \texttt{->}\ e$	(abstraction)
		$\texttt{if}\ \ b\ \ \texttt{then}\ \ e_1\ \ \texttt{else}\ \ e_2$	(conditional)

EXPRESSIONS AND CONSTRAINTS

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

`Success`: type of constraint expressions

EXPRESSIONS AND CONSTRAINTS

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

`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

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

```
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

```
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
```

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

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
```

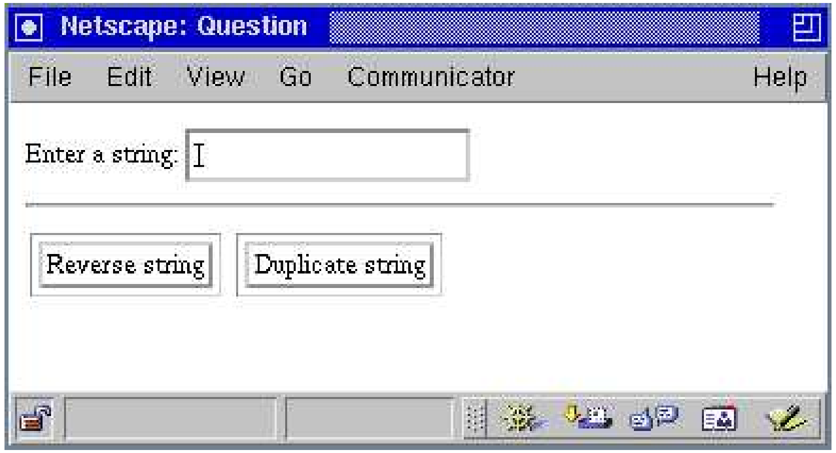
EXAMPLE: GUI PROGRAMMING [PADL'00]

A specification of a counter GUI:

```
Col [
   Entry [WRef val, Text "0", Background "yellow"],
   Row [Button (updateValue incrText val) [Text "Increment"],
        Button (setValue val "0")         [Text "Reset"],
        Button exitGUI                    [Text "Stop"]]]
     where val free
```

- ➜ layout structure ⇝ hierarchical structure, algebraic data type
- ➜ event handlers ⇝ functions contained in layout structure
- ➜ logical structure ⇝ dependencies in layout structure: free variables
- ➜ free variable `val`: reference to entry widget, used in event handlers

EXAMPLE: HTML PROGRAMMING [PADL'01]



```
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: `IO a` $\approx$ $World$ `->` `(a,`$World$`)`

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: $\boxed{\texttt{IO a} \approx \textit{World}\ \texttt{->}\ \texttt{(a,}\textit{World}\texttt{)}}$

Some primitive I/O actions:.

```
getChar :: IO Char             -- read character from stdin
putChar :: Char -> IO ()       -- write argument to stdout
return  :: a -> IO 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: `IO a` ≈ *World* `->` `(a,`*World*`)`

Some primitive I/O actions:.

```
getChar :: IO Char           -- read character from stdin
putChar :: Char -> IO ()     -- write argument to stdout
return  :: a -> IO a         -- do nothing and return argument
```

Compose actions: `(>>=) ::  IO a -> (a -> IO b) -> IO b`

`getChar >>= putChar`: copy character from input to output

Specialized composition: ignore result of first action:

```
   (>>)    :: IO a -> IO b -> IO b
  x >> y  =  x >>= \_->y
```

MONADIC I/O: EXAMPLES

Example: output action for strings (`String` ≈ `[Char]`)

```
putStr :: String -> IO ()
putStr []     = return ()
putStr (c:cs) = putChar c >> putStr cs
```

MONADIC I/O: EXAMPLES

Example: output action for strings (`String` ≈ `[Char]`)

```
putStr :: String -> IO ()
putStr []     = return ()
putStr (c:cs) = putChar c >> putStr cs
```

Syntactic sugar: Haskell's `do` notation

$$\begin{array}{lll} \texttt{do}\ p\ \texttt{<-}\ a_1 & \approx & a_1\ \texttt{>>=}\ \backslash p\ \texttt{->}\ a_2 \\ \quad\ \ a_2 & & \end{array}$$

Example: read a line

```
getLine = do c <- getChar
             if c=='\n' then return []
                        else do cs <- getLine
                                return (c:cs)
```

PREDICATES

Predicates (logic programming) ≈ functions with result type `Success`

```
isPrime :: Int -> Success
isPrime 2 = success
isPrime 3 = success
isPrime 5 = success
isPrime 7 = success

isPrimePair :: Int -> Int -> Success
isPrimePair x y = isPrime x & isPrime y & x+2 =:= y
```

Pure logic programs ⇝ direct translation into Curry programs

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)

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 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
```

BASIC EXAMPLES

```
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

BASIC EXAMPLES

```
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")

$\leadsto$ encapsulate non-deterministic search in I/O actions

ENCAPSULATING NON-DETERMINISM

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

⇝ encapsulate non-deterministic search in I/O actions

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

returns list of all solutions for constraint abstraction

`getAllSolutions (\x -> known (prime x))` ⇝ all `known` primes

ENCAPSULATING NON-DETERMINISM

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

⇝ encapsulate non-deterministic search in I/O actions

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

returns list of all solutions for constraint abstraction

`getAllSolutions (\x -> known (prime x))` ⇝ all known primes

Print list of all known primes:

```
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:

```
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
```

LOGIC PROGRAMMING WITH DYNAMIC PREDICATES

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:

```
do assert (prime 2)
   known1 <- getKnowledge   -- should be [2]
   assert (prime 3)
   assert (prime 5)
   known2 <- getKnowledge   -- should be [2,3,5]
   sols2 <- getAllSolutions (\x -> known2 (prime x))
   sols1 <- getAllSolutions (\x -> known1 (prime x))
   return (sols1,sols2)     ⇝ ([2],[2,3,5])
```

Computation (`getAllSolutions`) later than access (`getKnowledge`)

- ➜ `getKnowledge` conceptually copies current database
- ➜ efficiently implemented by time stamps

PERSISTENCE

Real applications require persistent data

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

PERSISTENCE

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)

TRANSACTIONS

Problem with persistent data: changes by concurrent processes

- ➜ synchronization necessary
- ➜ database community: transactions

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
```

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
```

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

`transaction try42` ⇝ `Nothing` (no change to `prime`)

IMPLEMENTATION

Dynamic predicates implemented in PAKCS (Curry$\mapsto$Prolog):

- ➜ dynamic predicate $\approx$ data structure (actual arguments, file name)
- ➜ facts stored in main memory
- ➜ `assert`/`retract` $\approx$ Prolog's assert/retract
- ➜ facts with time stamps [birth,death]

IMPLEMENTATION

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` $\leadsto$ time stamp [CT,∞]

`retract` $\leadsto$ set death time to CT

`getKnowledge` $\leadsto$ keep CT and check time stamp of unifiable facts

IMPLEMENTATION

Dynamic predicates implemented in PAKCS (Curry$\mapsto$Prolog):

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

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

`assert` $\leadsto$ time stamp [CT,∞]

`retract` $\leadsto$ set death time to CT

`getKnowledge` $\leadsto$ 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”)

IMPLEMENTATION (CONT'D)

Transactions and concurrent access:

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

IMPLEMENTATION (CONT'D)

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)

CONCLUSIONS

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/`