Michael Sintek

FLIP:

Functional-plus-Logic Programming on an Integrated Platform

Universität Kaiserslautern
Fachbereich Informatik
AG Künstliche Intelligenz – Expertensysteme
Prof. Dr. M. M. Richter

April 1995

Betreuung:
Prof. Dr. Michael M. Richter
Dr. Harold Boley

Integration of Relational and Functional Languages

Motivation A

- to combine two declarative paradigms into a more expressive one
- to allow existing software libraries in relational and functional (here LL, a COMMON LISP derivative) languages to be used together without the need of re-implementation

Integration of Relational and Functional Languages

Motivation B

- to speed up relational programs by transforming deterministic relations into functions (factor 2–4)

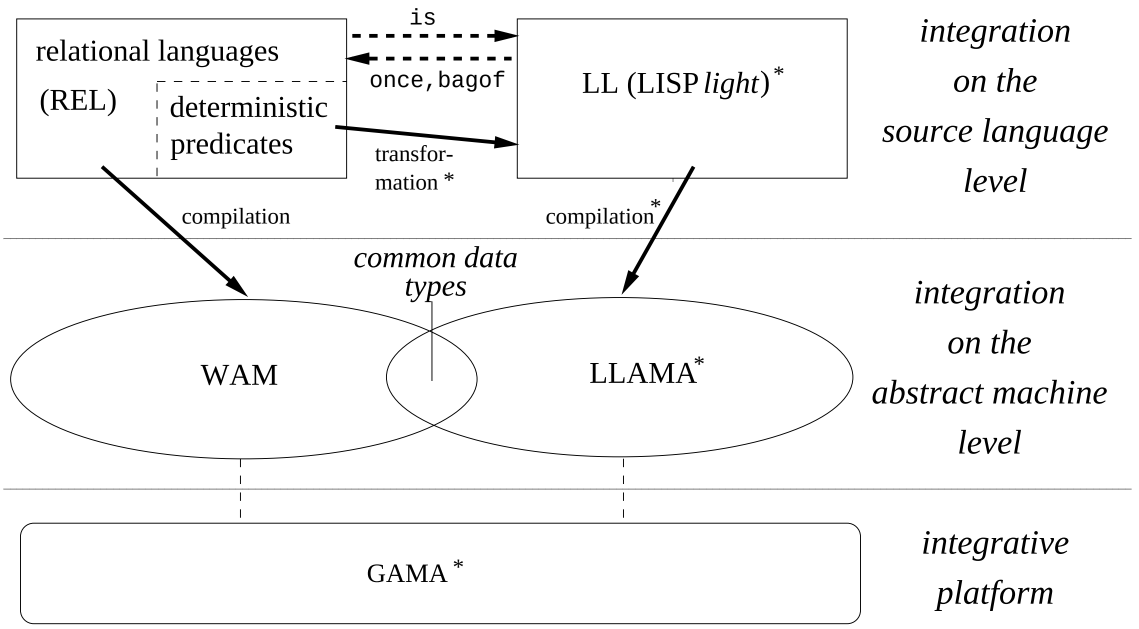
- to enhance the expressiveness of relational languages by new extra-logicals with the help of functions

Integration on Two Levels

1. *Abstract Machine Level:*
   the WAM and the LLAMA, an abstract machine especially designed for the efficient execution of LL, are coupled

2. *Source Language Level:*
   LL functions are accessible from relations and vice versa

Outline



A: LL

B: REL $\leftrightarrow$ LL

C: deterministic predicates $\rightarrow$ LL

D: LLAMA

E: LL $\rightarrow$ LLAMA

A: LL — LISP *light*

LL is a mainly functional subset of (COMMON) LISP extended by PROLOG structures.

LL Data Types

- symbols
- numbers (integers and reals)
- strings
- lists and dotted pairs
- structures

LL Structures

PROLOG	RELFUN	LL
`f(a,b,c)`	`f[a,b,c]`	`[f a b c]`

- `(struct 'f 'a (struct 'g 'b 'c))` $\xrightarrow{\varepsilon}$ `[f a [g b c]]`

- `(arity` `[`*f* *arg*$_1$ ... *arg*$_n$`])` $\xrightarrow{\varepsilon}$ n

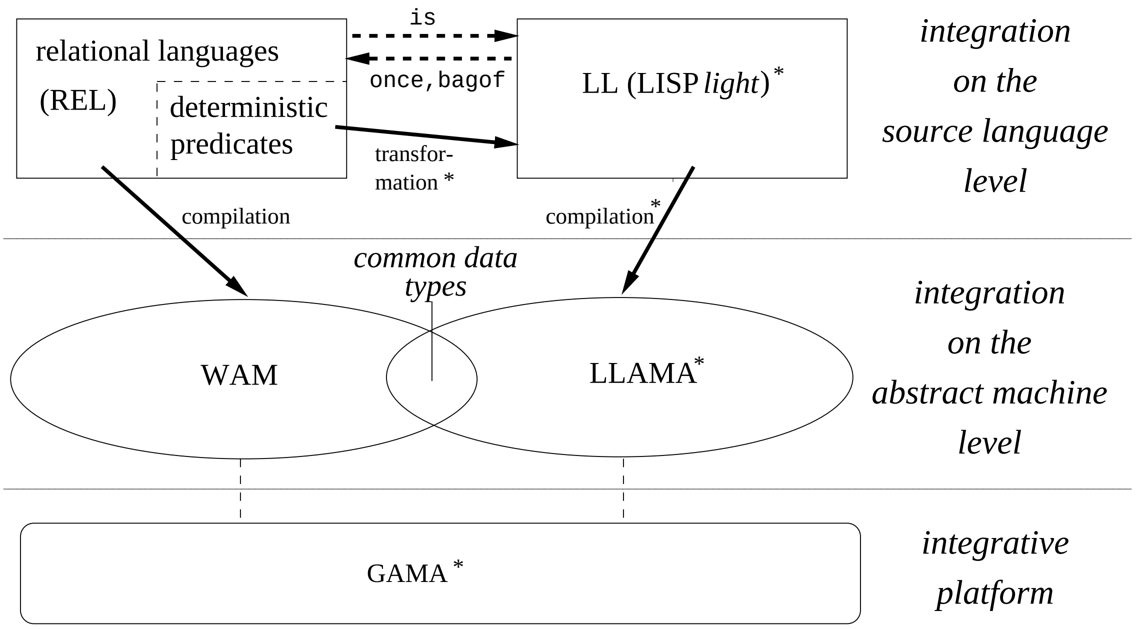
- `(functor` `[`*f* *arg*$_1$ ... *arg*$_n$`])` $\xrightarrow{\varepsilon}$ *f*

- `(elt` `[`*f* *arg*$_0$ ... *arg*$_n$`]` m`)` $\xrightarrow{\varepsilon}$ *arg*$_m$
  with $0 \leq m \leq n$

LL Builtins

LL Builtins	
quoting	quote, '
lists and dotted pairs	cons, list, car, cdr, null, consp
structures	struct, structp, arity, functor, elt
(single/re-) assignments	let, let*, setq, psetq
equality	equal, eq, eql
numerical builtins	+, -, *, /, 1+, 1-, =, /=, <, >, <=, >=
string builtins	string<, string>
I/O builtins	read, print
higher-order builtins	funcall, apply, lambda, function, eval
sequential evaluation	progn
conditional evaluation	if, cond, and, or
non-local exits and loops	catch, throw, loop, return, do
prelude	caar, cadr, cdar, cddr, mapcar, append, reverse, member, assoc, sort

Outline



√ A: LL

B: REL ↔ LL

C: deterministic predicates → LL

D: LLAMA

E: LL → LLAMA

B: REL $\leftrightarrow$ LL

1. *LL functions are called from relations:* generalized `is` builtin allowing functions defined in LL to be used on the right-hand side

2. *relations are called from LL functions:* the relation returns only the first solution (`once` mode), or the relation returns a list of all solutions (`bagof` mode)

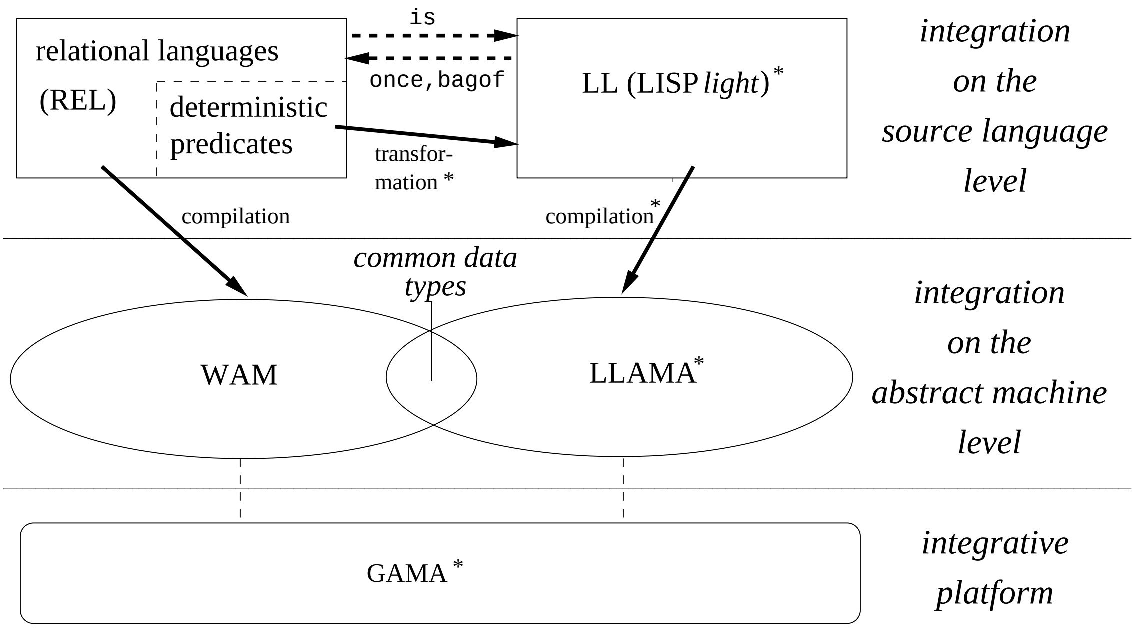
Calling LL Functions From Relations

```
(defun append (l1 l2)
  (if (null l1)
      l2
      (cons (car l1) (append (cdr l1) l2))))

(defun member (x l)
  (if l (or (equal x (car l))
            (member x (cdr l)))))


p(X,L1,L2,R) :-
  member(X,L1),
  R is append(L1,L2).
p(X,L1,L2,R) :-
  member(X,L2),
  R is append(L2,L1).
```

Queries

```
rfe-p> p(3, [1,2,3], [4,5,6], R)
true
R = [1, 2, 3, 4, 5, 6]
rfe-p> more
unknown
rfe-p> p(3, [1,2,3], [3,4,5], R)
true
R = [1, 2, 3, 3, 4, 5]
rfe-p> more
true
R = [3, 4, 5, 1, 2, 3]
rfe-p> more
unknown
rfe-p> p(3, [1,2], [4,5])
unknown
```

Outline



√ A: LL

√ B: REL $\leftrightarrow$ LL

C: deterministic predicates $\rightarrow$ LL

D: LLAMA

E: LL $\rightarrow$ LLAMA

C: Deterministic Predicates $\rightarrow$ LL

- definition of deterministic predicates
  - deterministic predicate
  - deeply deterministic predicate
- query set restriction: modes
- determinism analysis and transformation
  - detection of (deeply) deterministic predicates
  - transformation with several intermediate representations

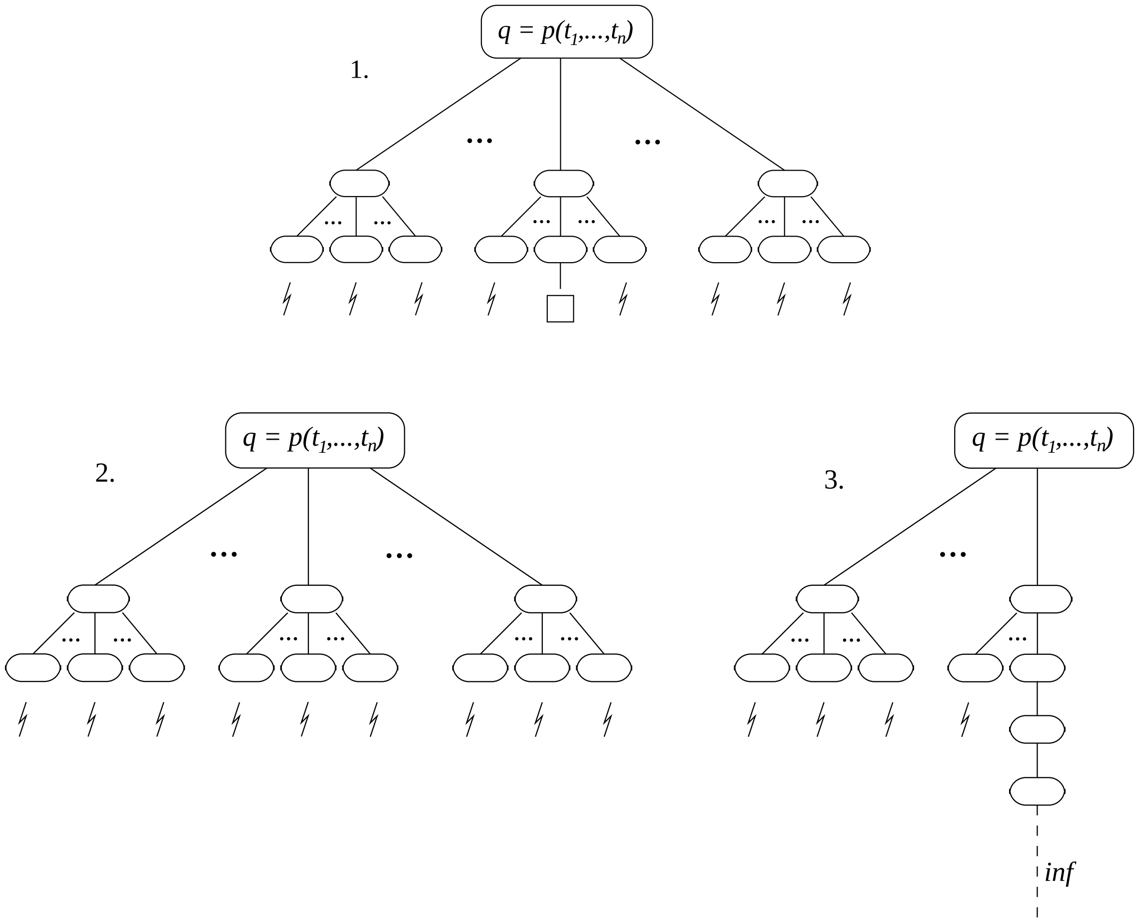
Deterministic Predicates

For a program P and a query set Q, a predicate p is deterministic if for all possible queries in Q all activated subgoals q with p as predicate either

1. have exactly one solution,
2. fail, or
3. enter a non-terminating branch (before computing a first solution)

when called separately.

Deterministic Predicates



Deeply Deterministic Predicates

All predicates called in the execution of deterministic predicates must themselves be deterministic:

internal backtracking in the execution of a deterministic predicate is not allowed.

$\Rightarrow$

All deterministic predicates can be replaced by a simple and directly obtainable procedure or function specified in a deterministic imperative or functional language.

Query Set Restriction: Modes

For a predicate p, the mode declaration is of the form

$$p(\mu_1, \ldots, \mu_n)$$

with $\mu_i \in \{\mathcal{G}, \mathcal{X}\}$, $i \in \{1, \ldots, n\}$

- $\mathcal{G}$: the argument is *ground* when the predicate is called
- $\mathcal{X}$: the argument is *unrestricted* (i.e. either free, bound, or even ground)

Detection of Deeply Deterministic Predicates

P program with predicates $\mathcal{P}$ and mode declarations M:

1. $\mathcal{C} \leftarrow \{p \in \mathcal{P} \mid \exists p(\mu_1, \ldots, \mu_n) \in M \ \exists i \in \{1, \ldots, n\} : \mu_i = \mathcal{G}\}$

2. while $\exists p \in \mathcal{C} : (p(\ldots) :- \ldots, q, \ldots .) \in P \ \wedge \pi(q) \notin \mathcal{C}$ do $\mathcal{C} \leftarrow \mathcal{C} \backslash \{p\}$

3. try to transform all predicates in $\mathcal{C}$ into LL; remove all predicates from $\mathcal{C}$ which could not be thus transformed

4. *same as* 2.

Transformation 1:
Input (REL) → RELFUN Functions

mode fac($\mathcal{G}$, $\mathcal{X}$).

```
a) fac(0,1) :- !.
b) fac(X,Y) :-
     X1 is -(X,1),
     fac(X1, FX1),
     Y is *(X, FX1).
```

$$\rightarrow$$

```
+) fac(X1, X2) :- X2 is fac/2-1(X1).
a) fac/2-1(0) :- !& 1.
b) fac/2-1(X) :-
     X1 is -(X, 1),
     FX1 is fac/2-1(X1),
     Y is *(X, FX1)
     & Y.
```

Transformation 2:

RELFUN Functions $\rightarrow (\mathcal{E}, \upsilon)$

a	$\mathcal{E}$	Arg#1	0
	υ	1	
b	$\mathcal{E}$	X1 FX1 Y	-(Arg#1, 1) fac/2-1(X1) *(Arg#1, FX1)
	υ	Y	

$(\mathcal{E}, \upsilon)$ is created by (local) abstract interpretation:

- normalizing heads
- normalizing lists and structures
- static unification
- unification of common subexpressions

Transformation 3:

$$(\mathcal{E}, \upsilon) \rightarrow \Lambda$$

```
a) fac/2-1(Arg#1) :- equal(0, Arg#1) & 1.
b) fac/2-1(Arg#1) :-&
     *(Arg#1, fac/2-1(-(Arg#1, 1))).
```

Λ clauses are RELFUN clauses with

- expressions have been inserted as arguments for other expressions when possible
- unification $\rightarrow$ assignments (via `is`) and equality tests (via `equal`)
- premises reordered: all arguments ground, test predicates before selectors
- structures and lists $\rightarrow$ LL constructors, selectors, and test predicates

Transformation 4:

$\Lambda \rightarrow$ LL

Λ clauses are collected into a single LL function definition, where the clauses are connected by a cascade of `if` and `let` statements:

```
(defun fac/2-1 (arg#1)
  (if (equal 0 arg#1)
      1
      (* arg#1 (fac/2-1 (- arg#1 1)))))
```

Preconditions:

- all (REL input) clauses but the last must contain a *quasi-final* cut

- the last clause must be a *catch-all* clause

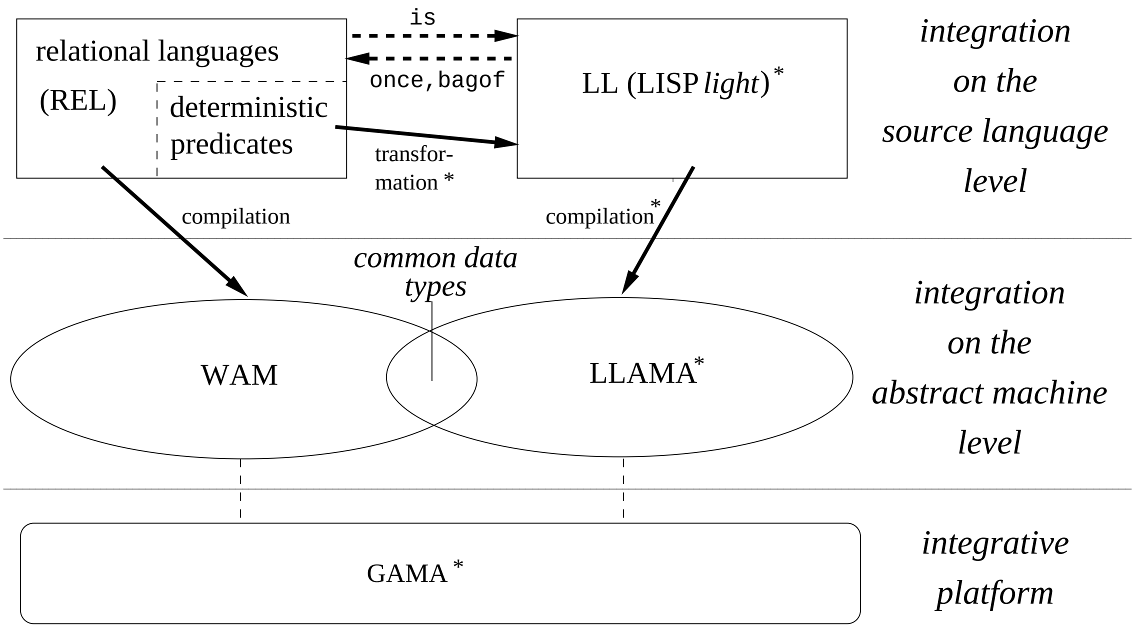
Additional Example

```
f(s[X,Y], u[s[X,Y],s[A,B]]) :-
      A is -(B,A),
      s[A,B] is s[X,Y],
      fac(X,A),
      fac(Y,B) !.
f(X, [X,X]).
```

$\rightarrow$

```
(defun f/2-1 (arg#1)
  (if (and (structp arg#1)
           (equal 's (functor arg#1))
           (equal 2 (arity arg#1)))
      (let ((x (elt arg#1 0)))
           (if (equal x (fac/2-1 x))
               (let ((y (elt arg#1 1)))
                    (if (and (equal x (- y x))
                             (equal y (fac/2-1 y)))
                        (struct 'u arg#1 arg#1)
                        (cons arg#1 (cons arg#1 nil))))
               (cons arg#1 (cons arg#1 nil))))
      (cons arg#1 (cons arg#1 nil))))
```

Outline



√ A: LL

√ B: REL $\leftrightarrow$ LL

√ C: deterministic predicates $\rightarrow$ LL

D: LLAMA

E: LL $\rightarrow$ LLAMA

D: LLAMA

LLAMA: simple and universal abstract stack machine as the target machine for the compilation of LL

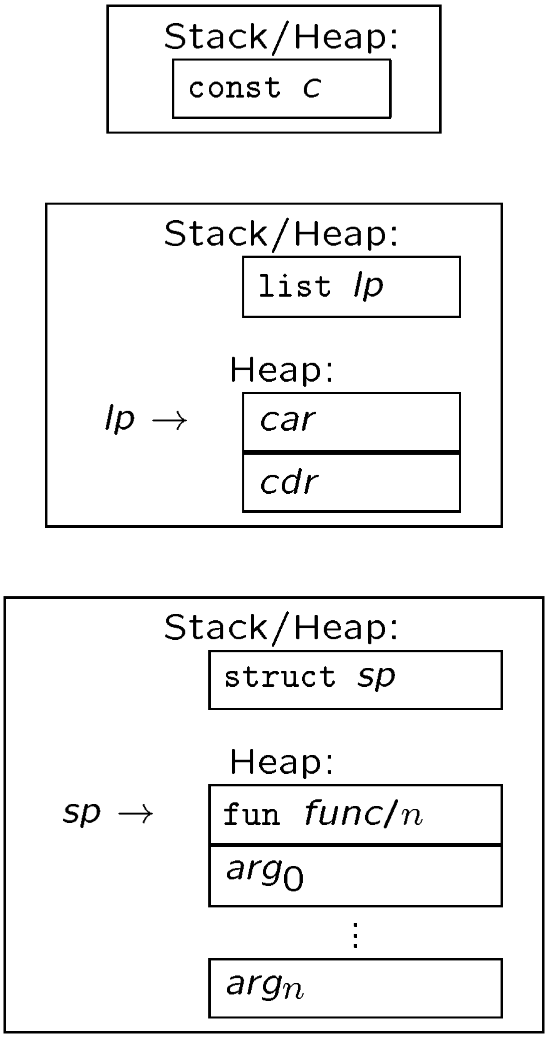
- registers and memory organization
- representation of data structures
- calling conventions
- instructions

Registers and Memory Organization

LLAMA registers	
P	program counter
SP	stack pointer
H	heap pointer
CT	catch stack pointer

LLAMA memory organization	
Stack	data and return address stack
Heap	heap area for complex data structures
Catch Stack	catch stack for non-local exits
Code	code area
Memory	general purpose memory

Representation of Data Structures



Calling Conventions

Stack:

⋮

previous
arg_1

⋮

arg_n
ret-addr
$local_1$

⋮

SP→ $local_m$

$\longrightarrow$

Stack:

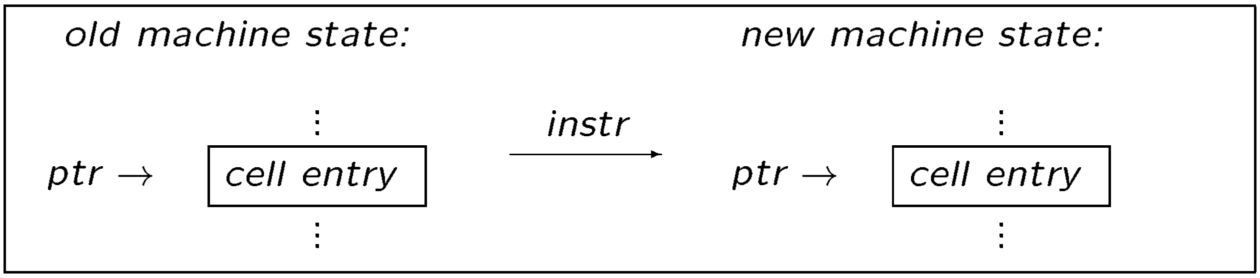
⋮

previous
SP→ *ret-value*

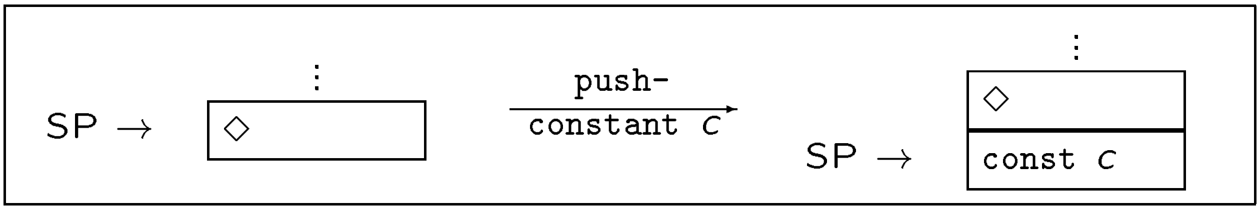
P = *ret-addr*

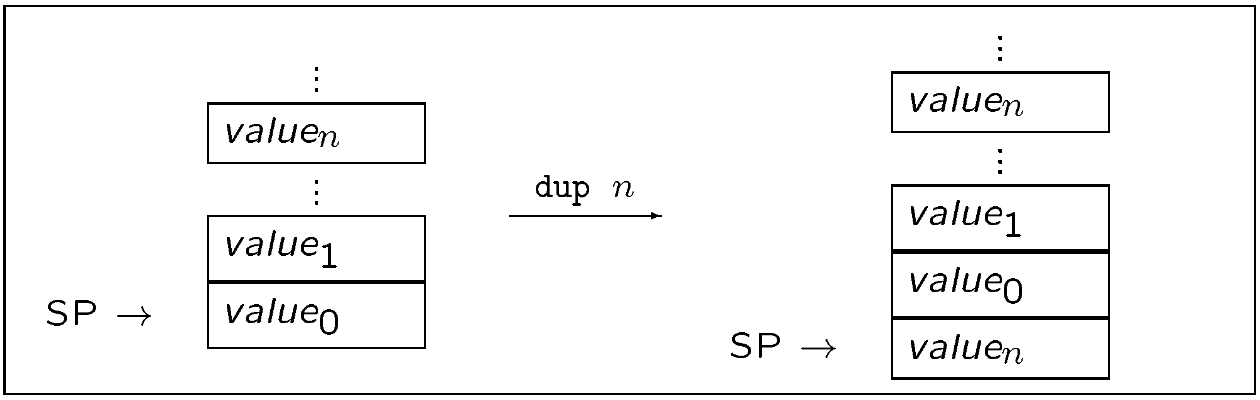
Instructions

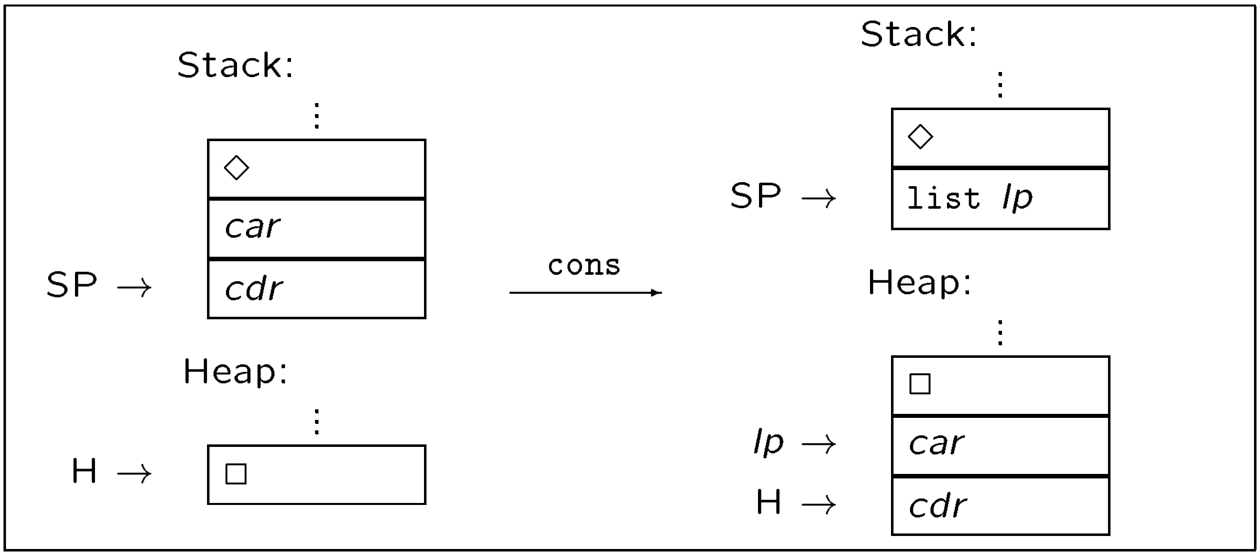
LLAMA instruction set	
constants	`push-constant`
lists	`cons`, `car`, `cdr`, `null`, `consp`
structures	`struct`, `structp`, `functor`, `arity`, `elt`
equality	`eq`, `eql`, `equal`
stack manipulation	`pop`, `remove`, `dup`, `set-nth`, `set-nth-`
control instr.	`ll-call`, `return`, `goto`, `on-nil-goto`, `catch`, `remove-tag`, `throw`, `catch-nil`, `throw-nil`
higher-order instr.	`funcall`, `apply`, `eval`
global variables	`set-global`, `set-global-`, `push-global`
builtins	`ll-builtin`
I/O builtins	`read`, `print`



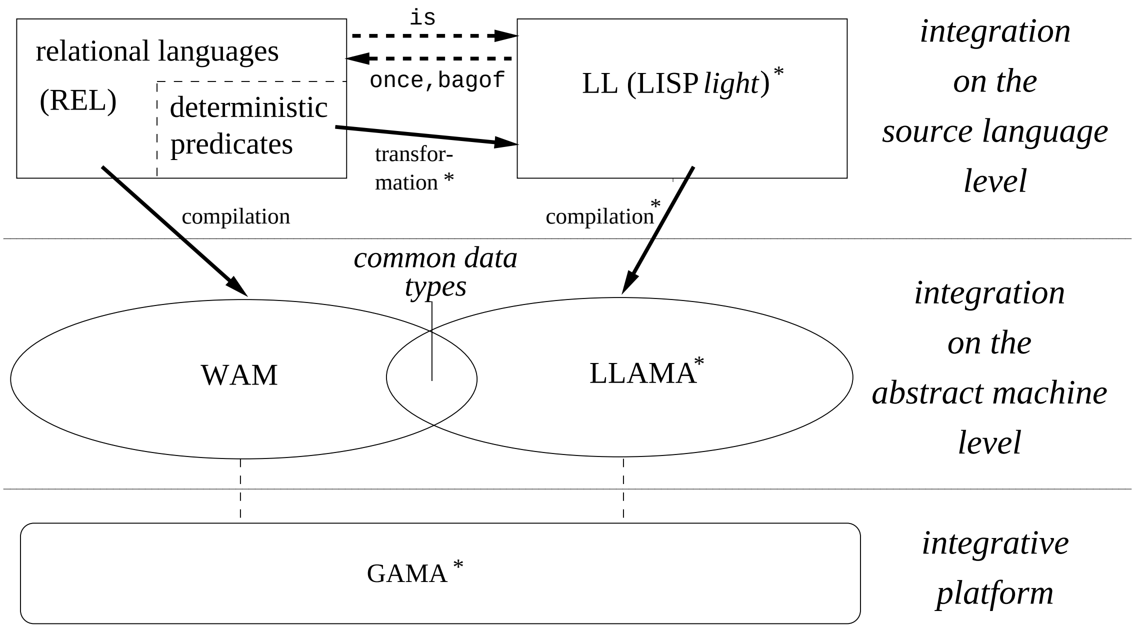
Sample Instructions







Outline



√ A: LL

√ B: REL $\leftrightarrow$ LL

√ C: deterministic predicates $\rightarrow$ LL

√ D: LLAMA

E: LL $\rightarrow$ LLAMA

E: LL $\rightarrow$ LLAMA

- compiler environment (for determining the positions of local variables at compile time)
- transformation rules
- peep-hole optimizer (for simplifying the code)

Compiler Environment

```
(defun f5 (x y) (+ x 5 y))
```

```
.proc f5/2
  dup 2              % variable x
  push-constant 5
  dup 3              % variable y
  ll-builtin + 3
  return 2
.end
```

Instruction		*Stack*						*Environment*
`.proc f5/2`		2: x	1: y	0: $retaddr$				$\{\langle x, 2\rangle, \langle y, 1\rangle\}$
`dup 2`	$\xrightarrow{x}$	3: x	2: y	1: $retaddr$	0: x			$\{\langle x, 3\rangle, \langle y, 2\rangle\}$
`push-constant 5`	$\xrightarrow{5}$	4: x	3: y	2: $retaddr$	1: x	0: 5		$\{\langle x, 4\rangle, \langle y, 3\rangle\}$
`dup 3`	$\xrightarrow{y}$	5: x	4: y	3: $retaddr$	2: x	1: 5	0: y	$\{\langle x, 5\rangle, \langle y, 4\rangle\}$
`ll-builtin + 3`	$\longrightarrow$	3: x	2: y	1: $retaddr$	0: $x + 5 + y$			$\{\langle x, 3\rangle, \langle y, 2\rangle\}$
`return 2`	$\longrightarrow$	0: $x + 5 + y$						$\{\}$

Transformation Rule Examples

$$\frac{\textit{constant}}{\texttt{push-constant}\ \textit{constant}}$$

$$\frac{[\textit{functor}\ \textit{arg}_1\ \ldots \textit{arg}_n]}{<\ (\texttt{struct}\ \texttt{'}\textit{functor}\ \texttt{'}\textit{arg}_1\ \ldots \texttt{'}\textit{arg}_n)\ >}$$

$$\frac{(\texttt{struct}\ \textit{functor}\ \textit{arg}_1\ \ldots \textit{arg}_n)}{\begin{array}{l} <\ \textit{functor}\ > \\ <\ \textit{arg}_1\ > \\ \ldots \\ <\ \textit{arg}_n\ > \\ \texttt{struct}\ \textit{n} \end{array}}$$

$$\frac{(\texttt{let}\ ((v_1\ \ e_1)\ \ldots (v_n\ \ e_n))\ \ .\ \ \textit{body})}{\begin{array}{l} <\ e_1\ > \\ \ldots \\ <\ e_n\ > \\ <\ (\texttt{progn}\ \ .\ \ \textit{body})\ >_{v_1 \ldots v_n} \\ \texttt{remove}\ \textit{n} \end{array}}$$

Compilation Example

```
(defun fac (x)
  (if (equal 0 x)
      1
      (* x (fac (- x 1)))))

.proc fac/1
  push-constant 0    % 0
  dup 2              % x
  equal              % (equal 0 x) ?
  on-nil-goto "L1"   % no, go to "L1"
  push-constant 1
  goto "L2"
"L1"
  dup 1              % x
  dup 2              % x
  push-constant 1    % 1
  ll-builtin - 2     % (- x 1)
  ll-call fac/1      % (fac (- x 1))
  ll-builtin * 2     % (* x (fac (- x 1)))
"L2"
  return 1
.end
```

Conclusions

- FLIP: implementation technique for relational (PROLOG) and for relational-functional languages (RELFUN)

- algorithms from existing software libraries can be combined even if specified in different paradigms without the need of re-implementation

- improving the efficiency of relational and relational-functional programs (factor 2–4)

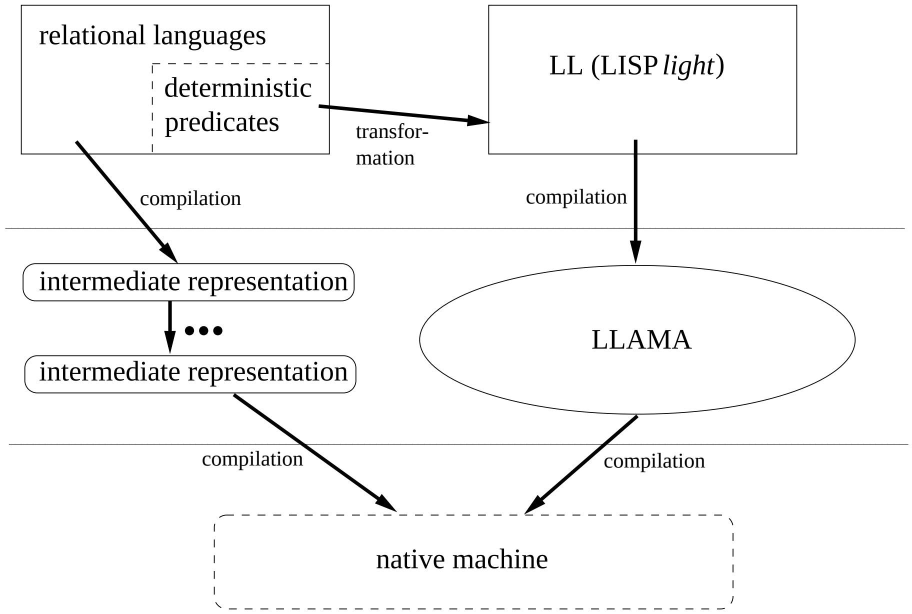
Future Work A

- adding optimizations exploiting the extra-functional builtins of LL in the transformation of deterministic predicates into LL functions (e.g. tail recursion $\rightarrow$ loops)

- using type definitions and signatures in addition to mode declarations $\rightarrow$ quasi-final cuts and catch-all clauses can be avoided

Future Work *B*

- using LL and LLAMA in the native code compilation of relational languages



- identifying additional programming paradigms integratable in an analogous manner (declarative or operational)