Integrated Logic Systems
Part 1: Deduction Systems

This part is concerned with the design and use of various deduction systems:

- Prolog
- Tableaux-Prover
- Answer Set Programming

Roadmap

- Prolog
  - Efficient implementation of terms, unification
  - Derivations
  - Search (Backtracking)

- Tableaux-Prover
  - Normal form transformation
  - Tableau Calculus
  - Application: Description Logics

- Answer Set Programming
  - Stable model semantics for logic programs
  - Implementation techniques
  - Applications

Schedule

Lectures		Tutorials
April, 13th, 2010	Prolog	April, 20th, 2010
April, 27th, 2010	Prolog	May, 4th, 2010
May, 11th, 2010	Prolog	May, 18th, 2010
June, 1st, 2010	Tableaux-Prover	June, 8th, 2010
June, 15th, 2010	Answer Set Programming	June, 22nd, 2010
June, 29th, 2010	Answer Set Programming	July, 6th, 2010
		July, 20th, 2010

Chapter Overview

- Introducing the Warren Abstract Machine (WAM)
- Language L_0: handling terms
- Compiling L_0-queries and L_0-programs
- Unification
- From L_0 to L_1: handling atoms
- Special data structures: constants and lists

Warren Abstract Machine

The WAM is the standard for all existing Prolog implementations.

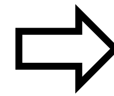
It provides an abstract method for compiling Prolog programs and queries into basic, “assembler-like” commands.

Knowing the WAM enables you to

- understand specific behaviors of a Prolog system
- design and implement basic functionalities of deduction systems
  - term unification
  - derivations (= sequences of deduction steps)
  - search (via backtracking)

The Basic Picture

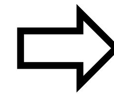
```
member(X,[X|_]).
member(X,[_|Y]):- member(X,Y).
subset([X|Y],Z):- member(X,Z),
                  ...
disjoint(X,Y):- ...
...
```



WAM commands

member/2:	...
m2:	...
	call member/2
subset/2:	...
	call member/2
	...
disjoint/2:	...
	...

```
?- subset([2|X],[1,Y,3]).
```



...

call subset/2

The Language L_0

Consider a term alphabet.

A program is a term.

A query is a term.

Intended meaning:

- "query" *t* fails wrt "program" *s* if *s* and *t* are not unifiable
- otherwise *t* succeeds with a binding of the variables in *t* obtained by unifying it with *s*

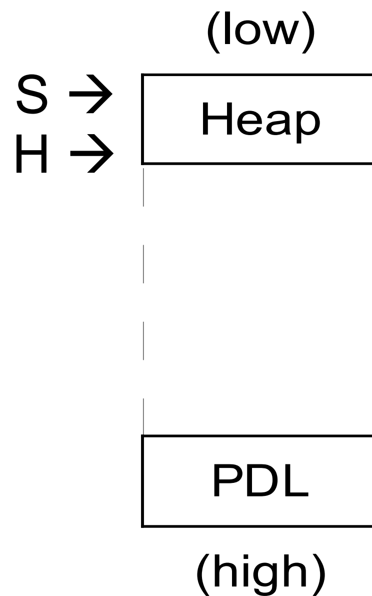
Example:

```
p(f(X),h(Y,f(a)),Y).

?- p(Z,h(Z,W),f(W)).
```

WAM_0 Memory Layout

Registers



S – special register,

pointing to “sub-term”

H – pointer to top of heap

Push-down-list

Term registers:
X_1, X_2, …

Heap Representation of Terms

Unbound variable:

i	REF	i

Bound variable:

i	REF	j	$j \neq i$

Non-variable term $t = f(t_1, ..., t_n)$:

i	STR	i+1
i+1	f/n	
i+2	$arg_1(t)$	
.	.	
.	.	
.	.	
.	.	
.	.	
.	.	
i+n+1	$arg_n(t)$	

Example

0	STR	1
1	h/2	
2	REF	2
3	REF	3
4	STR	5
5	f/1	
6	REF	3
7	STR	8
8	p/3	
9	REF	2
10	STR	1
11	STR	5

Heap representation of
p(Z, h(Z,W), f(W))

Compiling a Query

For processing a query term, there is a sufficient number of registers X_1, X_2, ...

- Register X_1 is allocated to the outermost term.
- The same register is allocated to all occurences of a given variable.

A term is seen as a set of “flattened” equations $X_i = f(X_{i_1}, ..., X_{i_n})$

Processing Flattened Equations

The equations need to be ordered

$X_1 = f_1(\vec{X}_1), \ldots, X_k = f_k(\vec{X}_k)$

such that no X_i occurs in $\vec{X}_1, \ldots, \vec{X}_{i-1}$

An equation $X_i = f(X_{i_1}, \ldots, X_{i_n})$ is processed as:

1. `put_structure f/n,` X_i
2. `set_variable` X_{i_1} or `set_value` X_{i_1}

⋮

n+1. `set_variable` X_{i_n} or `set_value` X_{i_n}

`set_variable` X_i used if X_i has not been processed before

`set_value` X_i used if X_i has been processed before

WAM_0 Machine Instructions (I)

`put_structure f/n,`X_i ≡ $HEAP[H] \leftarrow \langle STR, H+1 \rangle;$

$HEAP[H+1] \leftarrow$ `f/n`;

$X_i \leftarrow HEAP[H];$

$H \leftarrow H+2;$

`set_variable` X_i ≡ $HEAP[H] \leftarrow \langle REF, H \rangle;$

$X_i \leftarrow HEAP[H];$

$H \leftarrow H+1;$

`set_value` X_i ≡ $HEAP[H] \leftarrow X_i;$

$H \leftarrow H+1;$

Compiling a Program

The flattened equations need to be ordered

$X_1 = f_1(\vec{X}_1), \ldots, X_k = f_k(\vec{X}_k)$

such that no X_i occurs in $\vec{X}_{i+1}, \ldots, \vec{X}_k$

An equation $X_i = f(X_{i_1}, \ldots, X_{i_n})$ is processed as:

1. `get_structure f/n,` X_i
2. `unify_variable` X_{i_1} or `unify_value` X_{i_1}

⋮

n+1. `unify_variable` X_{i_n} or `unify_value` X_{i_n}

`unify_variable` X_i used if X_i has not been processed before

`unify_value` X_i used if X_i has been processed before

Dereferencing
(Through a Chain of Variable References)

```
function deref(a: address): address;
    begin
        ⟨tag,value⟩ ← STORE[a];
        if (tag=REF) ∧ (value ≠ a)
            then return deref (value)
            else return a
    end deref
```

WAM_0 Machine Instructions (II)

```
get_structure f/n,X_i ≡  addr ← deref (X_i);
                         case STORE[addr] of
                           ⟨REF,_⟩ :   HEAP[H] ← ⟨STR,H+1⟩;
                                       HEAP[H+1] ← f/n;
                                       bind(addr,H);
                                       H ← H+2;
                                       mode ← write;
                           ⟨STR,a⟩ :   if HEAP[a] = f/n
                                       then
                                           begin
                                               S ← a+1;
                                               mode ← read
                                           end
                                       else fail ← true;
                           other:   fail ← true;
                         endcase
```

Binding in WAM_0

```
procedure bind(a1,a2: address);
    begin
        case ← STORE[a1] of
            ⟨REF,_⟩   : STORE[a1] ← STORE[a2];
            other     : STORE[a2] ← STORE[a1];
        endcase;
    end bind
```

WAM_0 Machine Instructions (III)

```
unify_variable Xi ≡ case mode of
                         read:  Xi ← HEAP[S];
                         write: HEAP[H] ← ⟨REF,H⟩;
                                Xi ← HEAP[H];
                                H ← H+1;
                    endcase;
                    S ← S+1
unify_value Xi    ≡ case mode of
                         read:  unify (Xi ,S);
                         write: HEAP[H] ← Xi ;
                                H ← H+1;
                    endcase;
                    S ← S+1
```

Unification (Part I)

```
procedure unify(a₁,a₂: address);
    begin
        push(a₁,PDL); push(a₂,PDL);
        fail ← false;
        while ¬(empty(PDL) ∨ fail) do
            begin
                d₁ ← deref(pop(PDL)); d₂ ← deref(pop(PDL));
                if d₁ ≠ d₂ then
                    begin
                        ⟨t₁,v₁⟩ ← STORE[d₁];
                        ⟨t₂,v₂⟩ ← STORE[d₂];
                        if (t₁=REF) ∨ (t₂=REF)
                            then bind(d₁,d₂)
```

Unification (Part II)

```
else
    begin
        f₁ / n₁ ← STORE[v₁]; f₂ / n₂ ← STORE[v₂];
        if (f₁=f₂) ∧ (n₁=n₂)
            then
                for i ← 1 to n₁ do
                    begin
                        push(v₁ + i, PDL);
                        push(v₂ + i, PDL)
                    end
            else fail ← true
    end
end
  end
end unify
```

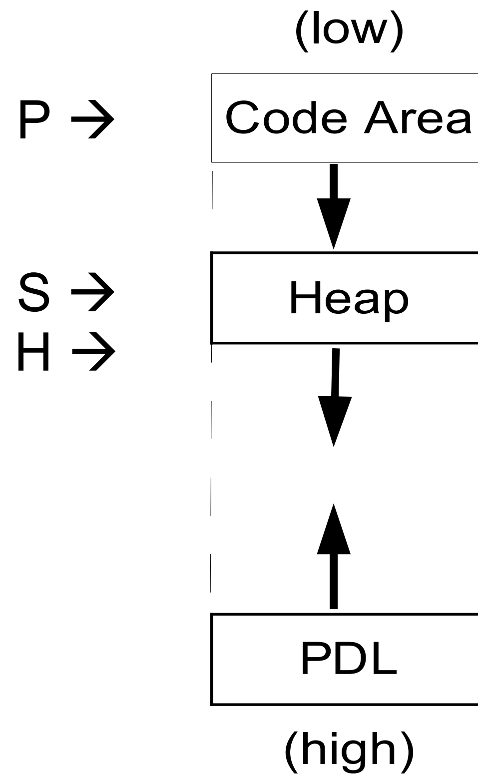
From L_0 to L_1

A program is a set of atoms, with at most one for each predicate name.

A query is an atom.

WAM$_1$ Memory Layout

Registers



Term registers:

$X_1, X_2, \ldots, X_n, \ldots$

Control Instructions

`call p/n` ≡ $P \leftarrow @(p/n);$

where @(*p*/*n*) stands for the address in the code area of the instructions for the clause defining predicate *p*/*n*.

If *p*/*n* has not been defined, failure occurs.

`proceed` ≡ no-op; (for the moment)

Argument Registers

The first n registers X_1, ..., X_n are systematically allocated to the arguments of an n-ary predicate.

To emphazise this, we write A_i if X_i is used as argument register.

Example: A_1, A_2, A_3, X_4, X_5, X_6, X_7 when processing a predicate with 3 arguments

Now also a variable can occur on the right hand side of a flattened equation.

Instructions for Variables as Roots

- `put_variable` X_n, A_i
  handles a root variable which occurs for the first time in a query
- `put_value` X_n, A_i
  handles a root variable which previously occured in a query
- `get_variable` X_n, A_i
  handles a root variable which occurs for the first time in a program atom
- `get_value` X_n, A_i
  handles a root variable which previously occurred in a program atom

WAM$_1$ Machine Instructions

`put_variable` X_n, A_i	≡	HEAP[H] ← ⟨REF,H⟩;
		X_n ← HEAP[H];
		A_i ← HEAP[H];
		H ← H+1;
`put_value` X_n, A_i	≡	A_i ← X_n;
`get_variable` X_n, A_i	≡	X_n ← A_i;
`get_value` X_n, A_i	≡	*unify*(X_n, A_i);

Special Datastructures: Heap Representation with Constants

7	STR	8
8	g/1	
9	CON	a
10	STR	11
11	f/2	
12	CON	b
13	STR	8

Heap representation of

f(b, g(a))

Instructions for Constants (I): Constants as Arguments

`put_constant` c, A_i ≡ $A_i \leftarrow \langle \text{CON}, c \rangle$;

`get_constant` c, A_i ≡ *addr* ← *deref*(A_i);

case STORE[*addr*] **of**

⟨REF,_⟩ : STORE[*addr*] ← ⟨CON,*c*⟩;

trail(addr);

⟨CON,*c*'⟩ : *fail* ← ($c \neq c'$);

other : *fail* ← **true**;

endcase;

trail(addr) ≡ no-op (for the moment)

Instructions for Constants (II): Constants Inside a Term

`set_constant c` ≡ HEAP[H] ← ⟨CON,*c*⟩;
H ← H+1;

`unify_constant c` ≡ **case** *mode* **of**
  read : *addr* ← *deref*(*S*);
    case STORE[*addr*] **of**
      ⟨REF,_⟩ : STORE[*addr*] ← ⟨CON,*c*⟩;
        trail(addr);
      ⟨CON,*c*'⟩ : *fail* ← ($c \neq c'$);
      other : *fail* ← **true**;
    endcase;
  write : HEAP[H] ← ⟨CON,*c*⟩;
    H ← H+1;
endcase;
S ← S+1

Heap Representation with Lists

1	REF	1
2	CON	[]
3	REF	3
4	REF	3
5	LIS	1
6	STR	7
7	f/1	
8	REF	1

Heap representation of

p(Z,[Z,W],f(W))

A1: ⟨REF 3⟩

A2: ⟨LIS 4⟩

A3: ⟨STR 7⟩

Instructions for Lists

`put_list` X_i ≡ $X_i \leftarrow \langle \text{LIS,H} \rangle;$

`get_list` X_i ≡ *addr* ← *deref*(X_i);

case STORE[*addr*] **of**

⟨REF,_⟩ : HEAP[H] ← ⟨LIS,H+1⟩;
bind(*addr*,H);
H ← H+1;
mode ← write;

⟨LIS,a⟩ : S ← a;
mode ← read;

other : *fail* ← **true**;

endcase;

Example: Query p(Z,[Z,W],f(W))

```
put_list X5
set_variable X6
set_constant []
put_variable X4,A1
put_list A2
set_value X4
set_value X5
put_structure f/1,A3
set_value X6
call p/3
```

Example: Fact p(f(X),[Y,f(a)],Y)

get_structure f/1,A_1

unify_variable X_4

get_list A_2

unify_variable X_5

unify_variable X_6

get_value X_5,A_3

get_list X_6

unify_variable X_7

unify_constant []

get_structure f/1,X_7

unify_constant a

proceed

Objectives

- Introducing the Warren Abstract Machine (WAM)
- Language L_0: handling terms
- Compiling L_0-queries and L_0-programs
- Unification
- From L_0 to L_1: handling atoms
- Special data structures: constants and lists