

# Parallelism in Prolog

One reason that Prolog is of interest to computer scientists is that its search mechanism lends itself to *parallel evaluation*.

In fact, it supports two different kinds of parallelism:

- AND Parallelism
- OR Parallelism

# And Parallelism

When we have a goal that contains subgoals connected by the “,” (And) operator, we may be able to utilize “and parallelism.”

Rather than solve subgoals in sequence, we may be able to solve them in parallel *if* bindings can be properly propagated.

Thus in

**$a(x), b(x,y), c(x,z), d(y,z).$**

we may be able to first solve  **$a(x)$** , binding  **$x$** , then solve  **$b(x,y)$**  and  **$c(x,z)$**  *in parallel*, binding  **$y$**  and  **$z$** , then finally solve  **$d(y,z)$** .

An example of this sort of and parallelism is

```
member(X,list1),  
  member1(X,list2), isPrime(X).
```

Here we can let **member(X,list1)** select an **x** value, then test **member1(X,list2)** and **isPrime(X)** in parallel. If one or the other fails, we just select another **x** from **list1** and retest **member1(X,list2)** and **isPrime(X)** in parallel.

# OR Parallelism

When we match a goal we almost always have a choice of several rules or facts that may be applicable.

Rather than try them in sequence, we can try several matches of different facts or rules in parallel. This is "or parallelism."

Thus given

**a(X) :- b(X).**

**a(Y) :- c(Y).**

when we try to solve

**a(10).**

we can simultaneously check both

**b(10)** and **c(10).**

Recall our definition of

```
member(X,L) :-  
    append(P,[X|S],L).
```

where `append` is defined as

```
append([],L,L).
```

```
append([X|L1],L2,[X|L3]) :-  
    append(L1,L2,L3).
```

Assume we have the query

```
| ? member(2,[1,2,3]).
```

This immediately simplifies to

```
append(P,[2|S],[1,2,3]).
```

Now there are two `append` definitions we can try in parallel:

(1) match `append(P,[2|S],[1,2,3])` with `append([],L,L)`. This requires that `[2|S] = [1,2,3]`, which must fail.

(2) match `append(P,[2|S],[1,2,3])` with `append([X|L1],L2,[X,L3])`.

This requires that  $P = [x | L1],$   
 $[2 | s] = L2, [1, 2, 3] = [x, L3].$   
Simplifying, we require that  $x = 1,$   
 $P = [1 | L1], L3 = [2, 3].$

Moreover we must solve  
`append(L1, L2, L3)` which simplifies  
to `append(L1, [2 | s], [2, 3]).`

We can match this call to `append` in  
two different ways, so or parallelism  
can be used again.

When we try matching  
`append(L1, [2 | s], [2, 3])` against  
`append([], L, L)` we get  
 $[2 | s] = [2, 3],$  which is satisfiable if  $s$   
is bound to  $[3].$  We therefore signal  
back that the query is true.

# Speculative Parallelism

Prolog also lends itself nicely to *speculative* parallelism. In this form of parallelism, we “guess” or speculate that some computation *may* be needed in the future and start it early. This speculative computation can often be done in parallel with the main (non-speculative) computation.

Recall our example of

```
member(X,list1),  
    member1(X,list2), isPrime(X).
```

After `member(X,list1)` has generated a preliminary solution for `x`, it is tested (perhaps in parallel) by `member1(X,list2)` and `isPrime(X)`.

But this value of `x` may be rejected by one or both of these tests. If it is,

we'll ask `member(x, list1)` to find a new binding for `x`. If we wish, this next binding can be generated *speculatively*, while the current value of `x` is being tested. In this way if the current value of `x` is rejected, we'll have a new value ready to try (or know that no other binding of `x` is possible).

If the current value of `x` is accepted, the extra speculative work we did is ignored. It wasn't needed, but was useful insurance in case further `x` bindings were needed.



# Reading Assignment

- Java for C++ Programmers  
(linked from class web page)
- Scott: Chapter 10