1 Introduction

The objective of this project was to implement and experiment with a simple concurrent Prolog interpreter. Prolog has been developed and continuously improved for more than two decades. Today a wide variety of different implementations exist, including concurrent versions [3].

Besides the simplicity of the language, there are a lot of issues involved with its implementation [2]. Among these the most important is to make Prolog control more intelligent based on information about the problem. We are not going to deal with this topic here. Instead, we make the control more efficient by using a brute-force concurrent search technique. The technique is essentially based on that at unification decision points we can proceed with all unifiable clauses in parallel. For this purpose the concurrent features of Oz provide an excellent development environment.

In the next section we give a formal definition of the simplified Prolog syntax used in SCIPIO\(^1\). Then we discuss the theoretical background of Prolog control, specifically unification and resolution. Section 4 presents concurrent resolution. In section 5 some implementation issues are discussed briefly.

2 Prolog syntax

The basic building blocks of Prolog programs are constants, variables, and function applications. The usual distinction between variables and contents is that variable names start with a capital letter. For simplicity, here function applications are denoted in conventional prefix notation, e.g. \(+\)(1, 2). From these basic elements of the language – called atoms – we can create a set of expressions (recursive function applications with constants and variables), in which a most important subset is the set of predicates (boolean expressions). Note that Prolog places no interpretation on user-defined expressions, but uses predicates to express the truth value of a logical statement.

A Prolog program consists of a knowledge base, and one or more queries built from predicates. The knowledge base contains statements (basically predicates which are true by definition) and rules expressing mathematical deduction, commonly called clauses. A rule is made of assertion predicates and a conclusion predicate such that when the the assertion part is true, the conclusion part is satisfied. A query is a statement whose truth value is to be found from the knowledge base.

[^1]: SCIPIO: Simple Concurrent Interactive Prolog Interpreter in Oz :)
Here we assume that the program contains only one query, since multiple queries can be included as a new rule.

2.1 A formal definition

We use a simplified version of the formal definition of the syntax ([1] p.107):

| Program: | $\sigma ::= \gamma^+\psi$ |
| Clause:   | $\gamma ::= \pi : -\pi^+ . \pi$. |
| Predicate:| $\pi ::= \phi(\alpha^+) | \phi$ |
| Atom:     | $\alpha ::= \phi | \Phi | \phi(\alpha^+)$ |
| Query:    | $\psi ::= ? - \pi.$ |

Table 1: Formal definition of the syntax.

In the following example there are four statements and two rules in the knowledge base followed by a query:

father(tom, bob).
mother(mary, bob).
father(john, mary).
father(bob, bill).
grandfather(X, Y) :- father(X, Z); father(Z, Y).
grandfather(X, Y) :- father(X, Z); mother(Z, Y).
?- grandfather(X, Y).

The task of the interpreter is to find values for variables $X$ and $Y$ such that the statement of the query is satisfied. One solution is for example $X=\text{tom}$ and $Y=\text{bill}$, because the first rule can be applied to satisfy the query by substituting $Z=\text{bob}$ in it.

3 Fundamental principles of Prolog

3.1 Unification

Unification is the process of determining whether two expressions can be made identical by performing appropriate substitutions for their variables. In other words, two terms are unifiable, when they can be reduced to the same “canonical” form performing substitutions from a substitution list. Unification has to produce the minimal substitution list (most general unifier).

According to the first unification algorithm proposed by Robinson, expressions like $f(X, Y)$ and $f(Y, g(X))$ are not unifiable, essentially because we get to circular substitutions (a variable cannot unify with an expression containing it). Therefore, Robinson’s algorithm performed a verification, called occur check by traversing an expression tree looking for a given variable.

Most of the existing Prolog systems does not perform the occur check. (we decided to leave it out, too). This is explained by a noticable gain in execution time. However, the programmer has to be very careful not to unintentionally introduce circular expressions. Among the several solutions for this problem the most interesting is that of Prolog II, which basically allows the use of recursive expressions by extending unification to “rational trees”.

2
3.2 Resolution and Prolog control

Every Prolog program can be viewed as a theorem-proving system if we consider a query a theorem to be proven in the context of our knowledge base universe. The answer to a query is obtained by refutation, i.e., we have to show that the negation of the query is inconsistent with the clauses of the knowledge base. In other words, the knowledge base to which the query has been added is unsatisfiable or nonmodelizable. This technique is called resolution and originates from mathematical logic.

3.2.1 Relationship with first order predicate calculus

Prolog performs resolution on a special type of clauses named Horn clauses. A clause in predicate calculus is a universally quantified expression having the following form:

$$C = \forall(x_1, \ldots, x_k) A_1 \land \ldots \land A_n \Rightarrow B_1 \lor \ldots \lor B_m \equiv$$

$$\forall(x_1, \ldots, x_k) \neg A_1 \lor \ldots \lor \neg A_n \lor B_1 \lor \ldots \lor B_m$$

where $A_i$ and $B_i$ are predicates (literals), and $x_1, \ldots, x_k$ are their free variables. We say that the $B_i$ are positive, the $A_i$ are negative literals of the clause. A clause with at most one positive literal ($m \leq 1$) is called a Horn clause. There are four types of Horn clauses according to the values of $m$ and $n$:

- **Rules** ($m = 1$ and $n > 0$): $A_1 \land \ldots \land A_n \Rightarrow B$
- **Facts** ($m = 1$ and $n = 0$): $\Rightarrow B$
- **Denials** ($m = 0$ and $n > 0$): $A_1 \land \ldots \land A_n \Rightarrow$
- **Empty clause** ($m = 0$ and $n = 0$): $\Box$

Rules and facts are used to build a knowledge base. Denials represent inconsistent systems. An empty clause is always interpreted as false and is a result of a successful resolution.

Restricting the system to Horn clauses is particularly important, because we guarantee that there is no disjunctions in the conclusion predicate of a rule. Although not obvious, it has been shown that this restriction does not reduce the computational power of the logic.

3.2.2 Proof by refutation

A query of the form ? - $Q$ in which variables $x_1, \ldots, x_k$ appear means that there are values for $x_1, \ldots, x_k$ such that $Q$ is a logical consequence of the knowledge base. Let’s suppose that the knowledge base consists of clauses $C_0, \ldots, C_n$. Resolution starts with the negation of the query:

$$R_0 = \lnot Q$$

and tries to find a statement or head of a rule which unifies with the head of $R_0$ (where $R_i$ denotes the resolvent list and is interpreted as a disjunction of literals). To do this, first it selects a clause using a particular search rule. This is typically linear, i.e., the clauses are selected according to their order in the knowledge base. If there is no unifying clause, then resolution terminates, since $R_0$ is independent of, therefore consistent with the knowledge base. Thus, $Q$ is either inconsistent
with (if it is a pure statement without variables), or independent of the knowledge base (therefore cannot be a consequence of it).

If there is a clause \(C_i\) unifying with the head of \(R_i\), then resolution continues with generating a new resolvant list \(R_{i+1}\) by taking its head and if \(C_i\) is a rule by adding its assertion part to it. Notice that resolvant list \(R_i\) always contains negative literals which is due to our restriction to using Horn clauses. Also, we always unify a positive literal with a negative one, that’s why we can eliminate it from the resolvant list.

Resolution terminates when the resolvant list becomes empty (\(R_i = \emptyset\)). This means that we could successfully reduce \(\neg Q\) to the empty clause \(\square\) (interpreted as false) via successive applications of the contrapositive theorem\(^2\):

\[
\neg Q \Rightarrow \ldots \Rightarrow \square
\]

because when unifying a head of a rule with the head of \(R_i\), we get that:

\[
\neg L \land (A_1 \land \ldots \land A_n \Rightarrow L) \Leftrightarrow \neg (A_1 \land \ldots \land A_n) \equiv \neg A_1 \lor \ldots \lor \neg A_n
\]

which we add to \(R_i\) to get new resolvant list \(R_{i+1}\). Since \(\neg Q\) is essentially false in the context of our knowledge base, we get that \(Q\) is true.

Important to note that the substitution list obtained from one unification has to be kept and passed to the next one. Also, since the variables in a rule are local, we have to rename them every time before we unify a literal with the head of the rule. This way we make sure that we don’t interfere with variables having the same names already included in the substitution list.

Resolution is stuck when we cannot do unification of the head of the current resolvant list with any of the clauses in the knowledge base. In this case, backtracking is needed to get back to the last point when there were several clauses unifiable with the current literal. This is called the last decision point. Since we proceed in linear order, we select the next unifiable clause from the knowledge base.

Let’s see how resolution works with our example from the previous section (see Figure 1). We start with the denial consisting of the negation of the query, and find that none of the statements but

\(\neg A \Rightarrow B \Leftrightarrow \neg B \Rightarrow \neg A\)

Figure 1: Linear input resolution.
the head of the first rule is unifiable to it with substitutions \((X<-X_0; \ Y<-Y_0; \ Z<-Z_0)\) (we basically just renamed the variables).

Thus, we can discard the first literal and append the assertion part to the resolvent list. We proceed similarly and unify the first and the fourth statement to get the empty clause. Notice that by flattening the substitution list, we get that \((X<-tom; \ Y<-bill; \ Z<-bob)\). This particular kind of resolution is called *linear input resolution* or *SLD resolution*, because we consider the clauses according to their order in the knowledge base, and always select the head of the current resolvent list [2]. Notice that there is another solution \((X<-john; \ Y<-mary; \ Z<-bob)\), which we can get if we backtrack to the first step (the last decision point), and choose the second rule. If the search is sequential, this backtracking is necessary to get all solutions. However, if we have the opportunity of concurrent processing, we can traverse the search tree in parallel.

4 Concurrent resolution

Resolution can be viewed as the traversal of an *And/Or* proof tree (Figure 2). The *And* nodes correspond to appending the assertion part of a rule to the resolvent list, while the *Or* nodes represent unification points. The search rule (also called *or-control*) determines which available unifiable clause to choose when generating the next resolvent list. The *computation rule* or *and-control* deals with which element of the resolvent list to choose for this operation. In the sequential case, both are usually linear corresponding to the depth-first traversal of the proof tree. This fixed strategy of control is not always efficient. Several programming constructs (e.g., the *cut* mechanism or the *free* predicate of Prolog II), and implementation tricks (e.g., *clause indexing*) help to make the search more efficient.

![Figure 2: Traversal of an And/Or proof tree.](image)

When we look for only one solution, sequential search is quite acceptable. However, there is a problem with the fixed strategy of linear input resolution: it might not terminate. Let’s consider the following simple example:

integer(succ(N)) :- integer(N).
integer(zero).
?- integer(ten).

Prolog goes into an infinite loop because of always considering the first clause. However, there exists a derivation leading to the empty clause. In this case the problem can be solved by putting
the statement for zero in front of the rule. In general this will not work; a good example is the arithmetic program distributed with SCIPIO. A common solution for this problem is to limit the depth of the proof tree, and backtrack as soon as we reach this limit.

If we have concurrent processing available, it is possible to traverse the tree in parallel. One way to do it is to parallelize the search at unification choice points (at even levels of the tree). Another choice is to process And nodes concurrently (at the odd levels). However, it is quite difficult to combine the two, because the handling of the substitution lists becomes very complicated. We have implemented the first type simply because the unification list only has to be duplicated if dealing with multiple unifications at the same time.

Concurrent resolution is attractive if we need to find multiple (or all) solutions for a problem. One drawback is that the size of the proof tree grows exponentially. Thus, a huge number of threads have to be created and run concurrently. Supercomputers might handle some situations, but on a single processor system finding even one solution might take a lot of time (which could be found otherwise quickly by an intelligent sequential search). Also, there has to be a depth limit for the search and/or a terminating mechanism (Ctrl-c).

## 5 Implementation issues

Variables and constants are represented by \( v(\text{Name}) \) and \( c(\text{Name}) \), respectively, where \text{Name} stands for an Oz string (a list of characters). Variable names have to start with capital letters. Function applications are of the form \( p(\text{Name} \ \text{ArgList}) \) where \text{Name} denotes the name of the function and \text{ArgList} = \{ \text{Arg1} \ \text{Arg2} \ldots \ \text{Argn} \} \) where the \text{Argi} can be variables, constants, or function applications. For example \( p(\text{"father"} \ [v(\text{"X"}) \ c(\text{"bob"})]) \) corresponds to \( \text{father}(X,\text{bob}) \).

Programs are built from a clause list (the knowledge base) and a query statement \( \text{prg}(\text{KBase Query}) \). Clauses can be either statements \( \text{st}(\text{Pred}) \), where \text{Pred} is a function application (a predicate), or “cules” (clause rules) represented by \( \text{cl}(\text{Predicate} \ \text{PredList}) \), where \text{PredList} is a list of predicates forming the assertion part of the cule. To illustrate how a program is encoded let’s look at our example from section 2.1:

\[
\text{declare}
\text{Family} = \text{prg}([\text{st}(p(\text{"father"} \ [c(\text{"tom"}) \ c(\text{"bob"})]))}
\text{st}(p(\text{"mother"} \ [c(\text{"mary"}) \ c(\text{"bob"})]))}
\text{st}(p(\text{"father"} \ [c(\text{"john"}) \ c(\text{"mary"})]))}
\text{st}(p(\text{"father"} \ [c(\text{"bob"}) \ c(\text{"bill"})]))}
\text{cl}(p(\text{"grandfather"} \ [v(\text{"X"}) \ v(\text{"Y"})]))
[p(\text{"father"} \ [v(\text{"X"}) \ v(\text{"Z"})])]
[p(\text{"father"} \ [v(\text{"Z"}) \ v(\text{"Y"})])])
\text{cl}(p(\text{"grandfather"} \ [v(\text{"X"}) \ v(\text{"Y"})]))
[p(\text{"father"} \ [v(\text{"X"}) \ v(\text{"Z"})])]
[p(\text{"mother"} \ [v(\text{"Z"}) \ v(\text{"Y"})])])]
\text{st}(p(\text{"grandfather"} \ [v(\text{"X"}) \ v(\text{"Y"})])))
\]

Note that the query part of a correct program should be a single statement.

\(^3\) Which we certainly do have in Oz.
\(^4\) The tree starts at level 0.
\(^5\) Both are provided in SCIPIO.
The interpreter is written in a pure functional style, except for minor issues such as thread counting and search termination. The core of the control is based on the concurrent ForAll mechanism:

```prolog
proc {CForAll Xs P ...}
case Xs of nil then skip
[] X | Xr then
  thread {P X ...} end
  {CForAll Xr P ...}
end
end
```

which takes a list and a procedure name P as arguments (... stands for extra arguments passed to P). For every element of the list CForAll calls P with the element passed as the first argument. Its operation is similar to that of ForAll except that it creates a new thread for every procedure call.

Prolog control can be implemented in numerous ways. For clarity we used mutually recursive procedures Solve and Prove shown below. Procedure Solve takes resolvent list RL, substitution list SL, and current tree level N. First it checks whether RL is empty; if so then it has obtained a solution and displays it. If not, then it checks if it has reached the maximum tree level in which case it does nothing. Otherwise it calls Prove with every clause of the knowledge base (or environment in this case) concurrently by applying CForAll. The maximum tree depth limit and the environment are accessible as global variables implemented by cells.

```prolog
proc {Solve RL SL N}
case RL of nil then ... % Display answer
[] R | RLr then case N > {Access MaxN} then skip % Tree limit reached
else {CForAll {Access Env} Prove R RLr SL N}
end
end
end
```

```prolog
proc {Prove E G RL SL N}
... (*) % Increment thread counter here
local NE = {Rename E N} in
  local [Res1 SL1] = {Unify G {Statement NE} SL} in
    case Res1 of false then skip
     else {Solve {Append RL {PredList NE}} {MFlatten SL1} N+1}
     end
   end
... (*) % Decrement thread counter after all children have been spawned
end
```

Procedure Prove takes a clause E and attempts to unify it with goal G. Before unification it renames all variables of E by attaching the current tree level to their names. If unification fails, then the thread of Prove terminates (before that it decrements the thread counter). Otherwise Solve is called with the new resolvent list by joining the current resolvent list RL with the assertion part of E (if it is a statement, then the assertion part is an empty list). Before the new substitution list obtained from the unification is passed, we flatten it to help with unifications go faster at the
lower levels of the tree. This might be left out though, whether it helps or not is highly problem dependent.

Somehow we have to determine when the search terminates. We could use the built-in wait construct of Oz, but we don't want to have a large number of blocking threads. Therefore we explicitly increment a counting semaphore before a new thread is spawned, and decrement it before the thread is terminated. Reentrant locks prove to be useful to implement mutual exclusion:

```oz
proc {IncNumThreads}
local Success in
  lock NumLock then
    local O Q in
      {Exchange NumThreads O thread O+1 end}
      {Exchange Born Q thread Q+1 end}
    end
    Success = 1
  end
  case {Value.isFree Success} then {IncNumThreads}
    else skip end
end
```

Procedure `IncNumThreads` increments two counters placed in cells. Since loops are barely supported by Oz, we need to define this atomic operation recursively. A serious drawback of this approach is that the counter becomes a bottleneck in the search, since it creates a race condition. Proper thread scheduling should take care of it avoiding starvation.

The increment and decrement operations have to be placed at the beginning and at the end of of `Prove` (denoted by `(*)`) to preserve the semantics of the counters. Then we can just have a thread looking at the value of `Numthreads` every second; termination is guaranteed when the counter reaches zero.

Explicit thread termination is done by storing the identifier of the root thread in a cell. A procedure is assigned to a Tk button to retrieve the content of this cell and terminate the main thread. This also means that all children of the root thread, therefore the whole search process is terminated. In some situations, this may take a while, and sometimes we can see the terminate message appear earlier than some outputs coming from threads processing the lower levels of the proof tree.

6 Conclusions

We have implemented a concurrent Prolog interpreter in Oz exploiting its multi-paradigm features. The interpreter can retrieve answers to Prolog queries in parallel, and depending on the tree depth limit and the problem formulation we obtain all or part of the answers. Interesting to note that the search uses a lot of case constructs, so depending on whether using the sequential (`elseof`) or the concurrent (`[ ]`) version, we get different levels (coarse or fine grain) of parallelism.

A serious problem with concurrent resolution is that when the size of the problem is large, it requires an enormous amount of time to get at least one solution. A good example is the arithmetic program distributed with SCIPIO; if we set the maximum tree depth too large, we end up waiting

---

6This statement requires some formal verification, though. We can intuitively see that the counter is modified at the even levels of the tree, exactly where concurrent processing is done.
for minutes for the search to terminate. Therefore, we added the possibility of explicit termination, which does not seem to be always efficient. The user can also control the number of solutions by setting the maximum tree depth appropriately. The search could be improved by introducing either special language constructs or some formalized extra knowledge about the problem. Both techniques are out of the scope of this project and still subject of current research.

Critical issues of Oz such as proper scheduling and garbage collection can be well tested using SCIPIO. Oz turns out to be a wonderful and easy-to-use development environment, but its efficiency still needs to be improved. This project has been a very good opportunity to get familiar with some aspects of the language besides experimenting with concurrent resolution.

References

