Lecture Notes for 6.824, Artificial Itelligence
 Rev. November, 1992

Lifting

Lifting is a very general technique for improving the efficiency of programs that search. The basic intuition behind lifting is that symbolic reasoning is often more powerful than brute force search. For example, suppose we are asked to find integers x and y such that $x+y > y+x$. We can see immediately that no such pair of integers exists — there is no need to search all possible pairs of integers. In this example we have a constraint that is easily seen to be unsatisfiable. In many problems, however, there is a solution and finding that solution requires search. In cases where search is required, e.g., in NP-hard problems, symbolic computation can still be useful in pruning the search tree. For example suppose we want to find two integers x and y such that $x^3 + y^3 < 0$ and $y^2 - x^3 < 0$. These constraints are solvable and a solution can be found by search. However, rather than simply enumerate pairs of integers we can do a "binary search" on the values of x and y. We must either have $y \geq 0$ or $y < 0$. Symbolic reasoning (bounds propagation) can be used to show that the assumption $y \geq 0$ leads to a contradiction, and hence a branch of the search tree can be pruned.

Lifting introduces symbolic reasoning into search procedures. In this chapter lifting is formulated as a transformation on nondeterministic computer programs. Most search problems can be expressed concisely as nondeterministic "generate and test" programs. While such programs are short, and are easily verified to be correct, naive backtracking interpretation strategies result in extremely inefficient search. Lifting provides an alternative way of running these programs that automatically introduces symbolic variables and that uses symbolic computation to prune large fragments of the search tree.

In order to formulate lifting as a general way of interpreting nondeterministic programs we must first describe a particular nondeterministic programming language. The following section describes nondeterministic Lisp. Lifting is described in section 2.

1 Nondeterministic Lisp

Nondeterministic programs provide a natural way of characterizing many search problems. The complexity class NP is usually defined in terms of nondeterministic programs. A nondeterministic program is a computer program with different possible executions. In this chapter we investigate a nondeterministic version of the functional programming language Lisp. This chapter assumes prior experience with Lisp, although anyone with experience in some form of functional programming will probably be able to understand all of the examples. An introduction to functional programming in Lisp can be found in [Sussman and Abelson, 1985]. A complete specification of the Common Lisp programming language can be found in [Steele, 1990].

In an ordinary (deterministic) functional programming language each expression of the language either diverges (causes an infinite loop) or evaluates to a certain value. For example, in Lisp the expression `(+ 1 2)` evaluates to 3. In a nondeterministic programming language the evaluation of an expression involves nondeterministic choices. In nondeterministic Lisp these choices are declared using a new primitive called `EITHER`.[1] The expression `(EITHER 1 2)` has two possible values — 1 and 2. The expression `(+ 3 (EITHER 1 2))` also has two possible values — 4 and 5. Given this nondeterministic feature one can write simple recursive nondeterministic programs such as the following.

```
(DEFUN INTEGER-BETWEEN (MIN MAX)
  (IF (= MIN MAX)
      MIN
      (EITHER MIN (INTEGER-BETWEEN (1+ MIN) MAX))))
```

This defines a procedure `INTEGER-BETWEEN` which takes two numbers, `MIN` and `MAX`, and nondeterministically returns a number greater than or equal to `MIN` and less than or equal to `MAX`. Similarly, one can define a procedure that

[1] The primitive `EITHER` is essentially identical to the primitive `AMB` introduced by McCarthy in [McCarthy, 1967].

takes a list and nondeterministically returns some element of that list.

```
(DEFUN MEMBER-OF (L)
  (IF (NULL (REST L))
      (FIRST L)
      (EITHER (FIRST L)
              (MEMBER-OF (REST L)))))
```

The following procedure returns a list of n “bits” where each bit is either the Lisp value `NIL` or the Lisp value `T`.

```
(DEFUN LIST-OF-BITS (N)
  (IF (= N 0)
      NIL
      (CONS (EITHER T NIL)
            (LIST-OF-BITS (- N 1)))))
```

Nondeterministic Lisp also contains a primitive called `FAIL`. No values are generated by a call to `FAIL`. For example the expression `(EITHER 1 (EITHER (FAIL) 2))` has possible values 1 and 2. Using the new primitives `EITHER` and `FAIL` one can write simple generate and test programs. For example, one can write a program to nondeterministically generate a placement of eight queens on the chess board and then test to see if no two queens attack each other. Recall that a solution to the 8-queens problem involves placing one queen on each column and that a solution can be specified by a list of eight numbers where the i'th number specifies the row number of the queen on the i'th column.

```
(DEFUN QUEENS-SOLUTION (N)
  (REST-QUEENS N N))


(DEFUN REST-QUEENS (NQUEENS BOARDSIZE)
  (IF (= NQUEENS 0)
      NIL
```

```
      (LET ((REST (REST-QUEENS (- NQUEENS 1) BOARDSIZE))
            (NEXT (AN-INTEGER-BETWEEN 1 BOARDSIZE)))
        (IF (NOT-SAFE-TO-ADD NEXT 1 REST-QUEENS))
            (FAIL)
            (CONS NEXT REST-QUEENS))))


(DEFUN NOT-SAFE-TO-ADD (NEXT OFFSET REST)
  (AND REST
       (OR (= NEXT (CAR REST))
           (= (ABS (- NEXT (CAR REST)))
              OFFSET)
           (NOT-SAFE-TO-ADD NEXT (+ 1 OFFSET) (CDR REST)))))
```

There are two special forms which turn nondeterministic expressions into deterministic expressions (expressions with at most one possible value). The expression (ALL-VALUES <EXP>) *deterministically* returns the list of all possible values of the (nondeterministic) expression <EXP>. The expression (ONE-VALUE <EXP>) deterministically returns the "first" value of the expression <EXP>.[2] The top level Lisp evaluator will only evaluate deterministic expressions. To compute values of nondeterministic expressions one must evaluate expressions of the form (ONE-VALUE <EXP>) and (ALL-VALUES <EXP>). For example, one can evaluate (ONE-VALUE (QUEENS-SOLUTION)) or (ALL-VALUES (QUEENS-SOLUTION)).

1.1 Backtrackable Side Effects

A nondeterministic program has a tree of different possible execution paths. The procedures ONE-VALUE and ALL-VALUES must search this tree of possible execution paths to find nonfailing values of the given expression. Ordinary Lisp side effects, such as those done with SETQ or REPLACD, introduce "cross talk" between the various possible execution paths. This cross talk is con-

[2]The special form ONE-VALUE is analogous to *cut* in Prolog — it prevents backtracking to a second value of the given expression.

fusing and should be avoided. Fortunately, it is possible to have side effects without cross talk. To avoid cross talk each side effect should be local to the execution path in which it is performed. To allow for local side effects we provide a special macro, LOCALLY-SETF, which has the same syntax and semantics as SETF except that cross talk between different possible nondeterministic executions is avoided. The following procedure shows the use of side effects inside a nondeterministic computation. This procedure nondeterministically returns a nonlooping (simple) path in a graph from a given start node to a given finish node. To avoid loops we mark the nodes that have been included in the path. We represent a graph as a set of node date structures. Each node data structure has two slots — NEIGHBORS and MARKED?. The NEIGHBORS slot contains a list of all nodes which neighbor the given node in the graph. The MARKED? slot is either T or NIL.

```
(DEFSTRUCT (NODE (:CONC-NAME NIL))
  NEIGHBORS
  MARKED?)

(DEFUN (NEIGHBOR-OF NODE)
  (MEMBER-OF (NEIGHBORS NODE)))

(DEFUN SIMPLE-PATH (START-NODE FINISH-NODE)
  (IF (EQ START-NODE FINISH-NODE)
      NIL
      (LET ((NEIGHBOR (NEIGHBOR-OF START-NODE)))
        (WHEN (MARKED? NEIGHBOR)
          (FAIL))
        (SETF-LOCALLY (MARKED? NEIGHBOR) T)
        (CONS NEIGHBOR
              (SIMPLE-PATH NEIGHBOR FINISH-NODE)))))
```

All side effects done with LOCALLY-SETF that are executed inside the body of ONE-VALUE or ALL-VALUES are undone by the time these procedures return a value. The locality property of these side effects also ensures that a search of the possible executions of the procedure SIMPLE-PATH will generate all, and only, the non-looping paths.

2 Lifting

Lifting, in the most general sense, is the use of symbolic reasoning to prune search. Consider a non-deterministic generate and test program. Most generate and test programs can be written in the following form.

```
(LET ((X (A-FOO)))
  (IF (NOT (P X))
      (FAIL)
      X))
```

The above expression can return any value of `(A-FOO)` that satisfies the predicate `P`. Lifting replaces the nondeterministic computation of `(A-FOO)` by a deterministic computation that simply returns a variable of "type" `(A-FOO)`. I will assume that the `(P X)` can be computed deterministically when `X` is bound to a particular value of `(A-FOO)`. In the lifted (symbolic) execution, `X` is bound to a variable. In the liftedcomputation the value of the test `(P X)` is not determined — it is true for some values of `X` and false for others. However, the lifted computation can nondeterministically compute a value for `(P X)` in a rather trivial way — it can nondeterministically either return true or false as the value of `(P x)`. If the lifted computation returns true for `(P X)` it must record a constraint on the variable `X` — the constraint that `(P X)` must be true. If the computation of `(P x)` returns false it must record the constraint that `(P X)` is false. In general, a lifted computation introduces variables to represent nondeterministic values and records constraints on those values when it returns true or false on a test involving variables. In complex lifted computations the systems accumulates a large set of constraints on variables. Symbolic reasoning can then be used to prune branches of the search tree by identifying cases in which the constraint set is unsatisfiable.

The general notion of lifting is formulated relative to an abstract data type. An abstract data type is defined by a family of primitive procedures that can return, or can be applied to, values of that type. Suppose that the type τ is defined by the primitives f_1, f_2, $\ldots$ f_n. We assume that no primitives other than the f_is can return or can be applied to values of type τ — any primitive other than the f_is will generate an error if applied to a value of type τ. In this section we will use the abstract type "expression" as our main example. We define the abstract type expression with the following nine primitives.

- `SYMBOL->EXPRESSION` This primitive takes a Common Lisp symbol and returns an expression object that represents that symbol.
- `EXPRESSION-CONS` This primitive takes two expressions objects and returns the expression object representing the pair of the two given objects.
- `AN-EXPRESSION` This is primitive nondeterministically returns any expression object.
- `EXPRESSIONP` This primitive takes a value and returns `T` if that value is an expression object.
- `EXPRESSION-SYMBOLP` This primitive takes an expression and returns `T` if that expression is the representation of a symbol. i.e., was constructed with `SYMBOL->EXPRESSION`, and returns `NIL` otherwise.
- `EXPRESSION-EQUAL` This primitive takes two expression objects. It returns `T` if the two expression objects represent the same expression, i.e., if they either represent the same symbol or they both represent pairs with recursively equal respective parts. It returns `NIL` otherwise.
- `EXPRESSION-CAR` This primitive takes an expression object that represents a pair and returns the first component of the pair. It generates on error on any value that is not an expression or on an expression object that represents a symbol.
- `EXPRESSION-CDR` This is analogous to `EXPRESSION-CAR` but returns the second component of an expression pair.
- `EXPRESSION->LIST` This converts an expression object to normal Common Lisp list structure. If the expression object represents a symbol

then the represented symbol is returned. If the expression object represents a pair then the procedure is recursively applied to the two components of the pair and then returns a cons cell containing the two results.

Lifting involves two different implementations of the primitives that define an abstract data type — a ground implementation and a lifted implementation. The above descriptions of the nine primitives that define expressions correspond to the ground implementation. In the ground implementation we could have defined the primitive `AN-EXPRESSION` as follows.

```
(DEFUN AN-EXPRESSION ()
  (EITHER (SYMBOL->EXPRESSION (A-SYMBOL))
          (EXPRESSION-CONS (AN-EXPRESSION) (AN-EXPRESSION))))
```

However, each of the eight primitives for expressions behave differently in the lifted implementation. The lifted implementation of the expression data type can be described as follows.

- `SYMBOL->EXPRESSION` This is the same in ground and lifted implementations.
- `EXPRESSION-CONS` This is the same in the ground and lifted implementations.
- `AN-EXPRESSION` In the lifted implementation this primtive determistically returns a fresh expression variable (which is considered to be an instance of the abstract expression type).
- `EXPRESSIONP` This primitive takes a value and returns `T` if that value is an expression object. In the lifted implementation expression variables are considered to be expression objects.
- `EXPRESSION-SYMBOLP` This primitive behaves the same in the lifted and ground version when applied to any expression object other than an expression variable. When applied to an expression variable `X` it either

returns `T` and installs a symbol constraint stating that the variable must denote the expression representation of a symbol, or returns `NIL` and installs an equality constraint of the form `X` = `(EXPRESSION-CONS Y Z)` where `Y` and `Z` are new expression variables.

- `EXPRESSION-EQUAL` This primitive takes two expression objects e_1 and e_2 and nondeterministically either returns `T` or `NIL`. If it returns `T` it installs the equality constraint $e_1 = e_2$. If it returns nil it installs the disequality constraint $e_1 \neq e_2$.[3]

- `EXPRESSION-CAR` This primitive behaves the same in both the lifted and gound version when applied to an expression object other than an expression variable. When applied to an expression variable `X` it installs the equality constraint `X` = `(EXPRESSION-CONS Y Z)` where `Y` and `Z` are fresh expression variables and returns the expression variable `Y`.

- `EXPRESSION-CDR` This is analogous to `EXPRESSION-CAR`.

- `EXPRESSION->LIST` This converts an expression object to normal Common Lisp list structure. In the lifted implementation this procedure first finds a solution to all installed constraints and assigns each expression its value under that solution. After all variables have been assigned ground expression values, this procedure returns the list structure represented by its argument under that interpretation of the variables.

Before considering an example of ground verses lifted computation we define the following simple utility for converting traditional lisp expressions into objects that are instances of the abstract expression type defined above.

```
(DEFUN MAKE-EXP (X)
  (COND ((EXPRESSIONP X) X)
        ((SYMBOLP X)
         (SYMBOL->EXPRESSION X))
        ((CONSP X)
         (EXPRESSION-CONS (MAKE-EXP (CAR X))
```

[3]The term inequality already means something else.

```
                     (MAKE-EXP (CDR X))))
    (T (ERROR))))
```

Now consider the following simple program.

```
(LET ((X (AN-EXPRESSION)))
  (ENSURE (EXPRESSION-EQUAL
           X
           (MAKE-EXP '(THIS IS THE DESIRED EXPRESSION))))
  (EXPRESSION->LIST X))


(DEFUN ENSURE (X)
  (UNLESS X (FAIL)))
```

In the ground implementation, a call to AN-EXPRESSION has an infinite number of different possible nondeterministic values. A simple backtrack interpreter for the ground implementation would search for a very long time before finding the single possible nonfailing value. However, under the lifted interpretation there are only two possible executions, one of which immediately fails. The only nonfailing execution returns (THIS IS THE DESIRED EXPRESSION).

3 An Example: Lifting a Planner

In the formalization of blocks world planning presented in earlier chapters each blocks world operation is named by an expression such as (MOVE B7 B3 P2) which names the operation that moves the block B7 from the block B3 to the place on the table named by P2. If there are n blocks and places then there are order n^3 ground operations of this type. If there is an infinite set of blocks then there is an infinite set of operations. In this section we show how the lifting transformation can be used to reduce a planning search space with infinite branching factor to a much more tractable space with finite branching. We given the following implementation of a linear planner.

```
(DEFUN A-BLOCKS-OPERATOR ()
  (LET ((B1 (AN-EXPRESSION))
        (B2 (AN-EXPRESSION))
        (B3 (AN-EXPRESSION)))
    (MAKE-OPERATOR :NAME (MAKE-EXP `(MOVE ,B1 ,B2 ,B3))
                   :PREREQUISITES (LIST (MAKE-EXP `(ON ,B1 ,B2))
                                        (MAKE-EXP `(CLEAR ,B1))
                                        (MAKE-EXP `(CLEAR ,B3)))
                   :DELETE-LIST (LIST (MAKE-EXP `(ON ,B1 ,B2))
                                      (MAKE-EXP `(CLEAR ,B3)))
                   :ADD-LIST (LIST (MAKE-EXP `(CLEAR ,B2))
                                   (MAKE-EXP `(ON ,B1 ,B3))))))
```

The above procedure allows any expression to be used as the name of a block. Under the ground implemenation of expressions the generating expression (A-BLOCKS-OPERATOR) has an infinite set of possible values each of which is a ground operator specification corresponding to a particular move operation. Now consider the following simplified (linear) backward chaining planning procedure implemented in nondeterministic Lisp.

```
(DEFUN FIND-PLAN (SIGMA GAMMA LENGTH-LIMIT)
  (COND ((SUBSET? GAMMA SIGMA))
         *THE-EMPTY-PLAN*)
        ((= LENGTH-LIMIT 0)
         (FAIL))
        (T
         (LET ((OP (A-BLOCKS-OPERATOR)))
              (ENSURE (INTERSECTS? (ADD-LIST OP) GAMMA))
              (ENSURE (NOT (INTERSECTS? (DELETE-LIST OP)
                                        GAMMA)))
              (LET ((GAMMA2 (APPEND (PREREQUISITES OP)
                                    (SET-DIFF GAMMA
                                              (ADD-LIST OP)))))
                   (APPEND (FIND-PLAN SIGMA GAMMA2 (1- LENGTH-LIMIT))
```

```
                              (LIST (OPERATOR-NAME OP)))))))


(DEFUN ENSURE (X)
  (UNLESS X (FAIL)))


(DEFUN SUBSET? (S1 S2)
  (COND ((NULL S1) T)
        ((NOT (MEMBER? (CAR S1) S2)) NIL)
        (T (SUBSET? (CDR S1) S2))))


(DEFUN INTERSECT? (S1 S2)
  (COND ((NULL S1) NIL)
        ((MEMBER? (CAR S1) S2) T)
        (T (INTERSECTS? (CDR S1) S2))))


(DEFUN SET-DIFF (S1 S2)
  (COND ((NULL S1) NIL)
        ((MEMBER? (CAR S1) S2)
         (SET-DIFF (CDR S1) S2))
        (T (CONS (CAR S1)
                 (SET-DIFF (CDR S1) S2)))))


(DEFUN MEMBER? (X S)
  (COND ((NULL S) NIL)
        ((EXPRESSION-EQUAL X (CAR S)) T)
        (T (MEMBER? X (CDR S)))))
```

It is fairly easier to verify that the above planner is correct (although ineffecient) under the ground implementation of the expression data type. The procedure defines the backward chaining search space for a blocks world planning problem. The *only* source of nondeterminism under the ground implementation is the call to A-BLOCKS-OPERATOR which selects the next operation to be added to the plan. One might think that by using iterative deepening on the length limit one could use the above procedure to find a solution. Unfortunately, the procedure A-BLOCKS-OPERATOR as defined above has an infinite set of possible values. If there is no solution of length 1 then a call to the FIND-PLAN procedure with depth limit 1 will enumerate the set of possible blocks operators and will not terminate. The search space defined by the FIND-PLAN procedure has an infinite branching factor. Search trees with infinite branching factor can not be searched by iterative deepening.

Under the lifted implementation of the expression data type the expression (a-blocks-operator) is deterministic. All nondeterminism in the lifted procedure arises from the nondeterminism of equality tests in the procedure MEMBER?.

4 Lifting Lemmas

The ground and lifted implementations are related by three lifting lemmas. Unfortunately, we have defined the semantics of the programming language in enough detail to make the following definitions and lemmas completely rigorous.

> **Definition:** A *search problem* is a pair $<D,\ E>$ where D is a set of definitions in nondeterminsitic lisp and E is a nondeterministic lisp expression. A *ground solution* to a search problem $<P,\ E>$ is a nonfailing value v of E that can be computed under the ground implementation of abstract data types using the definitions in P. A *lifted result* to a problem $<P,\ E>$ is a pair $<u,\ \Sigma>$ where u is a nonfailing value of E that can be computed using the lifted interpretation of abstract data types and where Σ is the set of all constraints installed in computing the value u. A *solution via lifting* of $<P,\ E>$ is a value v such that there is some lifted result $<u,\ \Sigma>$ such that v can be derived from u by replacing all abstract variables by their value under a substitution that satisfies all the constraints in Σ.

Now let <P, E> be any search problem. We have the following lifting lemmas.

> **Lifting Soundness Lemma:** All solutions via lifting are solutions.

> **Lifting Completeness Lemma:** All solutions are solutions via lifting.

We can also state a general systematicity property for lifting. A precise statement of the systamiticity lemma would require formalizing the notion of a "computation". Informally a computation is a series of machine states $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_n$. A ground computation $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_n$ is an instance of a lifted computation $u_0 \rightarrow u_1 \rightarrow u_2 \rightarrow \ldots \rightarrow u_n$ if for each i there is a substitution satisfying the constraints present in the state u_i such that applying that subsitution maps the lifted state u_i to the ground state s_i. We can now (somewhat informally) state the systematicity lemma for lifting.

> **Lifting Systematiticity Lemma:** Each ground computation is a substititon instance of exactly one lifted computation.

The systematicity lemma ensures that if a search procedure is systematic under the ground implementation then it is also systematic under the lifted computation.

5 Solving Constraints: Unification

Lifting for the expression data type has the nice property that it only generates tractable constraint sets — for any set of constraints generated by lifting (only) the expression data type one can determine in linear time whether or not there exists a solution to those constraints. There are only three kinds of constraints generated by expression lifting. First, there are constraints that state that two expressions are equal, called equality constraints. Second, there are constraints that state that two expressions are not equal, called disequality constraints. And finally, there are constraints that state that a given expression variable must be a symbol, called symbol constraints. It turns out that if any set of constraints of this form has a solution then it has a "most general" solution. This most general solution is a substitution that maps expression variables to expressions (which may themselves contain variables). The most general solution, if it exists, can be computed from the equality constraints alone. Before giving the algorithm for determining satisfiability and computing the most general solution some terminology is needed.

> **Definition:** An *expression* is either an expresison symbol, an expression variable, or the expression cons of two expressions.

> **Definition:** A *substitution* is a mapping from expression variables to expressions. If s is a expression and ρ is a substitution then $\rho(s)$ denotes the result of simultaneously replacing all the variables in s by their image under the mapping ρ.

For example, if the substitution ρ maps the variable x to the expression `(f a)` then $\rho($`(h x)`$)$ is `(h (f a))`. A substitution can be represented by a finite enumeration of pairs $<x, s>$ where x is a variable and s is the expression that x maps to. If a substitution ρ is defined by a finite list of such pairs, and y is a variable that is not mapped by one of given pairs, then we define $\rho(y)$ to be y. This allows ρ to be defined on all variables — a technical convenience. Formally, we simply say that a substitution is a mapping from (all) variables to expressions.

> **Definition:** Let Σ be a set of equations between expressions. A *unifier* of Σ is a substitution ρ such that for each equation $s = t$ in Σ we have that $\rho(s)$ is the same expression as $\rho(t)$.

For example, if Σ contains the equation (f x b) = (f a y) then any substitution that maps x to a and y to b is a unifier of these two expressions. As another example consider the equation (f x y y) = (f x x y). A substitution is a unifier for this equation if and only if it interprets x and y as the same expression. Also note that the constraint (f x) = x is unsatisfiable — for any interpretation of x as a finite expression, the expression x must be different from the expression (f x). Some unifiers are more general than others. For example, consider the equation (f x y y) = (f x x y). The substitution that maps x to y and all other variables to themselves is a unifier for this equation — it maps both expressions to (f y y y). The substitution that maps both x and y to (h z) is also a unifier for the given equation — it maps both expressions to (f (h z) (h z) (h z)). However, the first unifier is "more general" than the second.

> **Definition:** A *refinement* of a substitution ρ is a substitution σ such that there exists a substitution δ such that for any variable x we have that $\sigma(x)$ equals $\delta(\rho(x))$.

For example, consider the substitution ρ that is the identity on all variables other than x and such that $\rho(x)$ is (f y). Also consider the substitution σ that is the identity on all variables other than x and y and such that $\sigma(y)$ is (h z) and $\sigma(x)$ is (f (h z)). In this case σ is a refinement of ρ. To see this consider the substitution δ that is the identity on all variables other than y and which maps y to (h z). For any variable w we have that $\sigma(w)$ equals $\delta(\rho(w))$.

> **Definition:** Let Σ be a set of equations between expressions. A *most general unifier* of Σ is a unifier ρ of Σ such that any other unifier of Σ is a refinement of ρ.

Consider a set of equations Σ. It is possible that Σ does not have any unifiers — Σ may contain an equation between distinct ground symbols or an equation of the form x = (f x). Fortunately one can determine in linear time in the size of Σ whether any unifiers exist. If unifiers do exist then there is always a most general unifier and a most general unifier can be computed in linear time in the size of Σ. A unification procedure (procedure for computing a most general unifier) is given below. The procedure given here is exponential rather than linear. However it seems that in practice the exponential algorithm is actually faster than existing implementations of the linear algorithm.

To determine if a set of equations, disequations, and symbol constraints is satisfiable one simply computes a most general unifier ρ of the equations and then checks that for each disequation $s \neq t$ the expression $\rho(s)$ is different from the expression $\rho(t)$, and for each symbol constaint on a variable x that $\rho(x)$ is either a symbol or a variable. Note that the unifier ρ is itself an interpretation of the variables that is a solution to the equations. If ρ does not violate any of the disequations or symbol constraints then it is a solution to the entire constraint set so the constraints are satisfiable. On the orher hand, if ρ violates a disequation or symbol constraint then, since all solutions are refinements of ρ, that constraint will be violated in all solutions so the constraint set is unsatisfiable.

The remainder of this section presents a simple unification procedure. Unification can be implemented using a form of constraint propagation on variables. Each variable is implemented as a data structure with a slot to hold the "binding" of that variable. The constraint propagation process fills in the binding slots of variables. A variable x can be bound to an expression s where s contains variables which may also be bound. The following can be used as an implementation of the `ASSERT-EQUAL!` primitive described in the previous section.[4]

```
(DEFUN ASSERT-EQUAL! (EXP1 EXP2)
  (COND ((EXPRESSION-VARIABLE? EXP1)
         (BIND! EXP1 EXP2))
        ((EXPRESSION-VARIABLE? EXP2)
         (BIND! EXP2 EXP1))
```

[4]The code given here is not quite adequate — it does not fail when a newly installed equality violates a previously installed disequality. As part of the specification of the lifting transformation we require that the lifted computation fail as soon as the equality and disequality constraints become unsatisfiable. The full implementation of this specification is not presented here.

```
        ((AND (CONSP EXP1) (CONSP EXP2))
         (ASSERT-EQUAL! (CAR EXP1) (CAR EXP2))
         (ASSERT-EQUAL! (CDR EXP1) (CDR EXP2)))
        (T
         (UNLESS (AND (ATOM EXP1)
                      (ATOM EXP2)
                      (EQ EXP1 EXP2))
           (FAIL)))))


(DEFUN BIND! (VARIABLE EXPRESSION)
  (LET ((BINDING (VARIABLE-BINDING VARIABLE)))
    (COND ((NOT (NULL BINDING))
           (ASSERT-EQUAL! BINDING EXPRESSION))
          (T
           (WHEN (OCCURS-IN? VARIABLE EXPRESSION)
             (FAIL))
           (LOCALLY-SETF (VARIABLE-BINDING VARIABLE)
                         EXPRESSION)))))


(DEFUN OCCURS-IN? (VAR EXPRESSION)
  (COND ((EQ VAR EXPRESSION)
         T)
        ((AND (EXPRESSION-VARIABLE? EXPRESSION)
              (NOT (NULL (BINDING EXPRESSION))))
         (OCCURS-IN? VAR (BINDING EXPRESSION)))
        ((CONSP EXP)
         (OR (OCCURS-IN? VAR (CAR EXPRESSION))
             (OCCURS-IN? VAR (CDR EXPRESSION))))
        (T NIL)))
```

Note that asserting an equality constraint between (ON x y) and (ON A B) will cause x to be bound to A and y to be bound to B. Also note that the binding function prevents an expression from being bound to an expression that contains itself (or contains itself through implicit chains of binding). To determine if a set of equations has a most general unifier, and to compute a most general unifier if it exists, one simply calls `ASSERT-EQUAL!` on each equality. If this does result in a failure then there exists a unifier for the given equations. Furthermore, the substitution implicit in the generated bindings represents a most general unifier.

6 Numerical Lifting

In general, lifting is a transformation between two different implemenations of an abstract data type. So far we have only given one example of lifting — expression lifting. In this section we consider a second example — numerical lifting. Numerical lifting is formulated here as two different implementations of an abstract data type representing floating point numbers. The data type used in the lifting transformation is called `LFLOAT` (for lifted floating point number). It should be remembered that the data type used in lifting is different from the standard notion of floating point number used in Common Lisp. The primitives defining this type are given below. For each primitive we describe both the ground and the lifted implementation.

- `FLOAT->LFLOAT` This procedure takes a Common Lisp floating point number and returns an LFLOAT object representing that number. This procedure has the same implementation in both the ground and lifted implementation.
- `AN-LFLOAT` In the ground implementation this nondeterministically generates a floating point number. In the lifted implementation this deterministically generates a variable ranging over floating point numbers.
- `LFLOAT-+` In the ground version this adds objects of type LFLOAT by adding the floating point numbers they represent. The result is an object of type LFLOAT. In the lifted implementation this behaves the same as in the ground implementation unless one of the arguments is an `LFLOAT` variable. When applied to `LFLOAT` objects x and y where one of x and y is a variable, this procedure returns a fresh LFLOAT variable z and installs the constraint $z = x + y$.

- `LFLOAT-*` and `LFLOAT-MINUS` These are analogous to `LFLOAT-+` except that they are for multiplication and subtraction respectively rather than addition.

- `LFLOAT-ENSURE-<=` This is a test that can cause failure, and hence backtracking. In the ground version it fails unless its first argument is less than or equal to its second argument, in which case it does nothing. In the lifted version, when applied to x and y, it installs the constraint $x \leq y$.

- `LFLOAT->FLOAT` This converts an LFLOAT object to a floating point number. In the ground implementation this is simply returns the floating point number represented by its argument. In the lifted implementation this nondeterministically assigns each `LFLOAT` variable a particular floating point value such that the resulting assignment is a solution to all constraints. It then returns the value of its argument.

In the lifted implementation of the procedure `LFLOAT->FLOAT` the constraints are solved using interval methods [Moore, 1966], [Hansen, 1968], [Hansen and Walster, 1991]. The procedudure maintains upper and lower numerical bounds on each variable. It iteratively refines these bounds as follows. It selects x to be the `LFLOAT` variable with the largest the largest range of values, i.e., with the maximimum difference between the upper and lower bounds. If two variables have the same maximum range then it picks the "first" such variable (this is not a backtrackable choice). Let m be the midpoint of the interval for x, i.e., m is the numerical average of the known upper and lower bounds for x.[5] The procedure then nondeterministically (and backtrackibly) either installs the constraint $x \leq m$ or installs the constraint $x > m$. The new constraint provides a new bound for x. This new bound can be propagated through constraints to establish new bounds for other variables. If any variable can be assigned an upper bound that is lower than its lower bound a failure occurs. This process is repeated until all variables have well defined values.

[5] Either the upper or lower bound, or both, may be infinite. There are various simple strategies for selecting m in this case.

As an example of floating point lifting consider the following.

```
(DEFUN SOLUTION ()
  (LET ((X (AN-LFLOAT))
        (Y (AN-LFLOAT)))
    (LFLOAT-ENSURE-= Y (LFLOAT-+ (LFLOAT-* X X)
                                 (FLOAT->LFLOAT -30.0)))
    (LFLOAT-ENSURE-= X (LFLOAT-* Y Y))
    (LIST (LFLOAT->FLOAT X) (LFLOAT->FLOAT Y))))
```

```
(DEFVAR EPSILON (FLOAT->LFLOAT .00001))
(DEFVAR MINUS-EPSILON (FLOAT->LFLOAT -.00001))

(DEFUN LFLOAT-ENSURE-= (X Y)
  (LET ((DIFF (LFLOAT-MINUS X Y)))
    (LFLOAT-ENSURE-<= DIFF EPSILON)
    (LFLOAT-ENSURE-<= MINUS-EPSILON DIFF))))
```

Clearly the ground implemenation of the above procedure would do an unacceptable amount of backtracking. The lifted implementation, however, can find an acceptable value fairly quickly.

7 CSP Lifting

The constraint propagation techniques used for solving constraint satisfaction problems (CSPs) can be formulated as a lifting transformation on an abstract data type. The objects of this abstract type will be called CSP objects. The type is defined by the following three primitives.

- `CSP-ONE-OF` This takes a list. In the ground implementation it non-deterministically returns one of the elements of the list. In the lifted implementation it deterministically returns a CSP variable whose range of possible values is the set of elements of the given list.

- `CSP-ENSURE-CONSTRAINT` This takes a predicate (any Common Lisp procedure) and a list of CSP objects. In the ground implementation it applies the predicate to the values represented by the CSP objects and fails if this application returns `NIL`. In the lifted implementation it installs the constraint that predicate must be true of the values of the CSP objects in the list.

- `CSP-VALUE` This procedure takes a CSP object and returns the value of that object. In the lifted implementation each CSP variable is (nondeterministically) assigned a value such that all constraints are satisfied. The corresponding value of the argument is then returned.

The lifted implementation of `CSP-VALUE` uses restricted GFC to prune the search for a solution. The n-queens problem can now be solved as follows.

```
(DEFUN QUEENS-SOLUTION (N)
  (MAPCAR 'CSP->VALUE (REST-QUEENS N N)))


(DEFUN REST-QUEENS (NQUEENS BOARDSIZE)
  (IF (= NQUEENS 0)
      NIL
      (LET ((REST (REST-QUEENS (- NQUEENS 1) BOARDSIZE))
            (NEXT (CSP-ONE-OF (NUMBER-LIST 1 BOARDSIZE))))
        (ENSURE-SAFE-TO-ADD NEXT 1 REST-QUEENS)
        (CONS NEXT REST-QUEENS))))


(DEFUN ENSURE-SAFE-TO-ADD (NEXT OFFSET REST)
  (WHEN REST
    (CSP-ENSURE-CONSTRAINT #'(LAMBDA (NEXT PREVIOUS)
                               (NOT (OR (= NEXT PREVIOUS)
                                        (= (ABS (- NEXT (CAR REST)))
                                           OFFSET))))
                           (LIST NEXT (CAR REST))))
```

```
    (ENSURE-SAFE-TO-ADD NEXT (+ 1 OFFSET) (CDR REST))))


(DEFUN NUMBER-LIST (MIN MAX)
  (IF (> MIN MAX)
      NIL
      (CONS MIN (NUMBER-LIST (+ MIN 1) MAX))))
```

In the ground implementation of the CSP data type the above program behaves identically to the n-queens program given in section 1. In the lifted implementation, however, the procedure `REST-QUEENS` is completely deterministic. It returns a list of CSP variables subject to a set of installed constraints. Search is postponed until all of the constraints have been installed. Restricted GFC is then used to prune the search for a solution. The lifted implementation is more efficient.

8 Problems

1. Consider rewrite rules such as the following.

$$(* \; x \; (+ \; y \; z)) \rightarrow (+ \; (* \; x \; y) \; (* \; x \; z))$$

$$(+ \; (+ \; x \; y) \; z) \rightarrow (+ \; x \; (+ \; y \; z))$$

$$x \rightarrow (+ \; x \; (+ \; y \; (- \; y)))$$

Each rewrite rule specifies that an instance of the left hand side can be rewritten as an instance of the right hand side. For example, the first rule specifies that $(* \; (f \; a) \; (+ \; c \; d))$ can be rewritten to $(+ \; (* \; (f \; a) \; c) \; (* \; (f \; a) \; d))$. The last rule specifies that $(f \; a)$ can be rewritten as $(+ \; (f \; a) \; (+ \; (g \; b) \; (- \; (g \; b))))$.

The above rules are only meant as an example — in this problem we consider the general concept of rewrite rules. Let R be a set of rewrite rules and let s and w be ground expressions (expressions that do not contain variables). Suppose that we use the following procedure to determine if there exists a way of rewriting s to w using rewrite rules in R using at most d rewriting steps.

Procedure to determine if s can be rewritten to w in d steps

1. If s is the same expression as w then return "yes".
2. If d is 0 then fail.
3. Select (nondeterministically) be a *ground instance* $u \rightarrow v$ of a rewrite rule in R.
4. If u does not equal s then fail.
5. If u equals s then return the value of a recursive call to the procedure on v, w and $d-1$.

The above procedure operates on ground expressions (step 3 selects a ground instance of a rewrite rule). The general lifting transformation can be applied to this procedure. The following questions concern the differences between the ground procedure and the lifted procedure.

In the following questions we assume that at least one rule in the rule set R contains a variable and that there are n different rules in R.

In the ground procedure given above, how many ways are there of executing step 3? In the lifted version of the procedure, what is the maximum number of ways that the sequence of step 3 followed by step 4 can execute without failure? Consider a call to the lifted version of the procedure. As a function of n (the number of rules) and d (the allowed number of steps), give the maximum possible number of leaves in the search tree, i.e., the maximum number of times some recursive call to the procedure can fail at step 2.

2. This problem is on the lifted nonlinear planner. Consider the nonlinear planning procedure `FIND-COMPLETION` given below. This procedure takes a nonlinear plan β and searches for a completion.

The Procedure `FIND-COMPLETION`(β n)

1. If the nonlinear plan β is order inconsistent, or the number of the steps in β is greater than n, then fail.

2. If there is an unsafe causal link in β then let $s \xrightarrow{P} w$ be the first such causal link and let v be the first threat to this link such that β does not contain either $v < s$ or $v > w$.

3. If an unsafe causal link was found in step 2 then return either `FIND-COMPLETION`$(\beta + (v < s), c)$ or `FIND-COMPLETION`$(\beta + (v > w), c)$.

4. If there exists a step in the symbol table and some prerequisite of that step which is not established by a causal link, then let w be the first such step and P be the first of w's prerequisites that is not established by a causal link.

5. If an unestablished precondition P of step w was found in step 4 do either of the following.

   (a) i. Let s be a step name in the symbol table.
       ii. Unless s adds P fail.
       iii. Return the plan `FIND-COMPLETION`$(\beta + s \xrightarrow{P} w, c)$.

   (b) i. Select an operator o_i and create a new step s with operation o_i.
       ii. Unless s adds P fail.
       iii. Return the plan `FIND-COMPLETION`$(\beta + s \xrightarrow{P} w, c)$.

6. If no unsafe causal links or open preconditions were found in steps 2 or 4 then return the plan β.

For each of the 11 steps and substeps of the above procedure state whether that step is deterministic or nondeterministic in both the ground and the lifted version of the planner *for blocks world planning.* Give a *brief* explanation of each answer. Several answers are different when the planner is restricted to the blocks world than when we consider arbitrary STRIPS planning. You should assume that the step names in the symbol table are ground symbols (no two step names can ever be equal). You should also assume that all sets are represented by ordered lists so that one can always talk about the "first" element of a set. This allows certain steps, such as step 2 in the ground procedure, to be unambiguously deterministic. Also, a step is consdiered to be nondeterministic if and only if there is more than one way of completeing the step without failing (the step has a nonfailing branching factor of at most one).

References

[Hansen and Walster, 1991] E. R. Hansen and G. W. Walster. Nonlinear equations and optimization. *Control and Games of Comput. Math. Appl.*, 1991. To appear in the second special issue on Global Optimization.

[Hansen, 1968] E. R. Hansen. On the solution of linear algebraic equations using interval arithmetic. *Mathematical Computation*, 22:153–165, 1968.

[Jaffar and Lassez, 1987] J. Jaffar and J. L. Lassez. Constraint logic programming. In *Proceedings of POPL-87*, pages 111–119, 1987.

[Knuth and Bendix, 1969] Donald E. Knuth and Peter B. Bendix. *Computational Problems in Abstract Algebra*, chapter Simple Word Problems in Universal Algebras, pages 263–297. Pergamon Press, Oxford, England, 1969.

[Lawler and Woods, 1966] E. L. Lawler and D. Woods. Branch-and-bound methods: A survey. *Operations Research*, 14, 1966.

[McCarthy, 1967] John McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg, editors, *Computer Programing and Formal Systems*. North-Holland, 1967.

[Moore, 1966] R.E. Moore. *Interval Analysis*. Prentice-Hall, 1966.

[Robinson, 1965] J. A. Robinson. A machine-oriented logic based on the resolution principle. *JACM*, 12(1), January 1965.

[Steele, 1990] G. L. Steele. *Common Lisp: The Language (Second Edition)*. Digital Press, 1990.

[Sussman and Abelson, 1985] Gerald Sussman and Herald Abelson. *Structure and Interpretation of Computer Programs*. MIT Press, 1985.

[van Hentenryck, 1989] Pascal van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.