

# Principles of Programming Languages

Logic Programming Interpreter

---

We describe an interpreter for Logic Programming written in Scheme. The interpreter re-uses many of the tools we developed when developing interpreters for Functional Programming (the sequence  $L1$  to  $L7$  we introduced in previous chapters).

In particular, we reuse directly:

- The Substitution abstract data type
- The Equation abstract data type
- The unification algorithm

We also implement the abstract syntax for LP using the same methodology we used for  $L_i$  - as a disjoint union type of expression types.

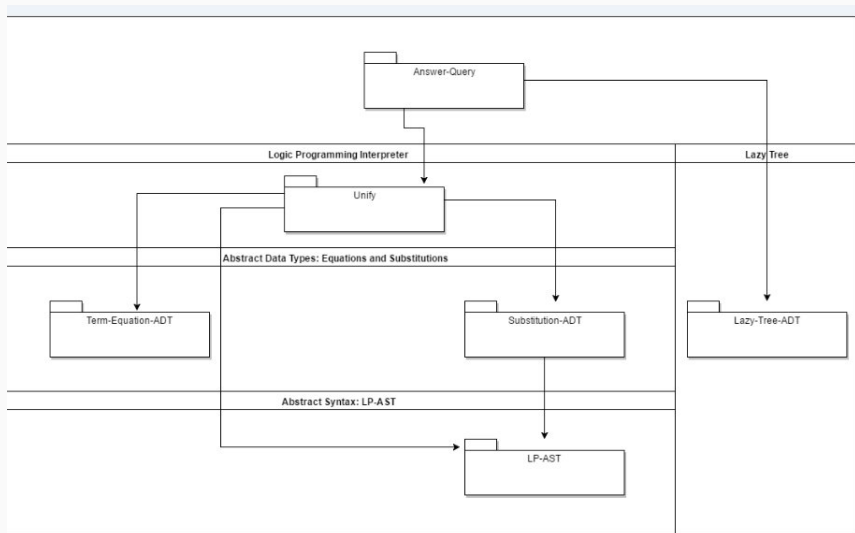
The new part of the LP interpreter is the implementation of the **answer-query** algorithm. Recall that the **answer-query** relies on two key ingredients:

- Unification
- The construction of a proof-tree given a program and a query.

We describe the proof-tree construction algorithm through the usage of a data type we call **lazy tree** which is a generalization of the lazy list we introduced in Chapter 4 to the case of a possibly infinite tree.

The overall system architecture is given in the following architecture diagram:

# Architecture of the Logic Programming Interpreter



1. **Syntax:** This layer includes, so far, only an abstract syntax module LP-AST, which defines a convenient interface to all syntactic elements in an LP program.
2. **ADTs:** The LP related ADTs are Substitution and Term-Equation, which are used for implementing a unification operation in Unify.

3. **lazy-tree-ADT** represents an n-ary labeled tree (has labels on internal nodes), whose depth might be infinite. The constructor of lazy trees wraps the child-branches of a node with a lambda abstraction that enables laziness: it delays the construction of children nodes until requested.

4. **LP-solver**: The Answer-Query module, defines the LP **Gsel** and **Rsel**, and the algorithm for proof-tree construction and search.

We implement this interpreter in Scheme. The programming style we use is extremely close to the one we used in TypeScript:

## Scheme Programming Style

- Data types are implemented as functional interfaces with a value constructor, type predicate and accessors. We do not use type annotations and type inference - but take care to describe the data structures we use as disjoint union types.
- The naming convention we use for constructors are **make-*TYPE***, type predicate is ***TYPE*?**, type accessors are functions named ***TYPE*->*FIELDNAME***.

# Scheme Programming Style

- Disjoint union types are defined only by a type predicate - as was the case in TypeScript - for example:

```
;; Term: Symbol | Number | Variable | Compound-term

;; Signature: term?(x)
;; Type: [T -> Boolean]
;; Purpose: Type predicate for terms
;; Pre-conditions: -
(define term?
  (lambda (x)
    (or (symbol? x)
        (number? x)
        (variable? x)
        (compound-term? x))))
```

- The structure of the recursive functions that traverse AST values follows the structure of the AST datatypes. The typical structure of the functions that we have used in TypeScript such as:

```
const typeTraversal = (x: TYPE): T =>  
  isSubTYPE1(x) ? do1(x) :  
  isSubTYPE2(x) ? do2(x) :  
  do3(x);
```

- ...is implemented in Scheme as:

```
(define type-traversal
  (lambda (x)
    (cond ((sub-type1? x) (do1 x))
          ((sub-type2? x) (do2 x))
          (else (do3 x)))))
```

- We use in general Scheme lists instead of TypeScript arrays and Scheme symbols instead of TypeScript strings.

We simplify error handling in the interpreter by using exceptions in Scheme. Exceptions are thrown in Racket using the **error** primitive - which interrupts the current code and returns to the toplevel execution with an error value.

For example:

```
(define program->procedure
  (lambda (program predicate)
    (let ((procedure (assoc predicate program)))
      (if procedure
          procedure
          (error 'program->procedure
                 "Program does not include predicate ~s"
                 predicate))))))
```

To simplify the code of the interpreter, we do not implement tagged disjoint types the way we did in TypeScript. We also do not implement a parser / unparser to read logic programs in concrete syntax and convert them to their AST representation. Instead, we use a *readable S-exp-based AST encoding*.

In this approach, a programs is represented as a list of the abstract representations of its procedures. Note that this list actually represents a set.

For example, the program:

```
append([], Xs, Xs).  
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).  
  
member(X, Ys) :- append(Zs, [X|Xs], Ys).
```

is represented as the list:

```
( (append 3)
  (0 ((append empty (var Xs) (var Xs))
      true))
  (1 ((append (cons (var X) (var Xs))
                (var Ys)
                (cons (var X) (var Zs))))
      (append (var Xs) (var Ys) (var Zs)))) )

((member 2)
  (0 ((member (var X) (var Ys))
      (append (var Zs)
                (cons (var X) (var Xs))
                (var Ys)) )) )
```

Note the following conventions in this encoding:

- A program is encoded as a list of procedures.

- Each procedure is encoded as a list:
  - starting with the (**predicate arity**) tag - for example **append/3** is encoded as (**append 3**) and **member/2** is encoded as (**member 2**).
  - Then a list of clauses (rules or facts).
  - Facts are encoded as rules with a body containing only the **true** atomic formula - so that all clauses are represented as rules.
  - Rules are numbered, starting at 0.

- Rules are represented as a list  
(`<number> (<head> . <body>)`) within  
each procedure.
- An atomic formula `pred(arg1, arg2)` is  
represented as `(pred arg1 arg2)`.

- Terms are represented as:
  - Constants as Scheme constants (symbol or number)
  - Variables as a pair (**var** <Varname>)
  - Composite terms  
**functor(arg1, arg2)** as  
**(functor arg1 arg2)** (In particular, lists are represented as nested **cons** terms)

- Program (list of procedures):

```
(make-program procedures-set)
```

```
(program->predicates program)
```

```
(program->procedure program predicate)
```

```
(program->procedure->numbered-rules program  
                                     predicate)
```

```
(program->procedure->rule program  
                           predicate  
                           number)
```

- procedure (a list of rules with heads for the same **pred/n**):  
    (make-procedure rules)  
    (procedure->predicate procedure)  
    (procedure->numbered-rules procedure)

- rule:  
    (make-rule head body)  
    (rule->head rule)  
    (rule->body rule)

- atomic-formula:  
 (make-predication predicate terms)

- variable:
  - (make-variable name)
  - (variable->name var)
  - (rename-variable var number)

- compound term:  
 (make-compound-term functor terms)

- query:  
    (make-query atomic-formulas)  
    (query->goals query)

A frequent operation in the interpreter consists of retrieving a rule from a program, and renaming all the variables in the rule. To facilitate this operation, variables can be encoded in two ways:

- Regular variable, as read from the program.
- Renamed variable - a variable annotated with a version number (which according to our convention corresponds to the layer of the proof-tree as we expand it breadth-first).

For example, the rule:

```
member(X, Ys) :- append(Zs, [X|Xs], Ys).
```

is encoded originally as follows (within the **member/2** procedure):

```
((member (var X) (var Ys))  
  (append (var Zs)  
          (cons (var X) (var Xs))  
          (var Ys)))
```

When this rule is retrieved from the program (using the **Rsel** procedure) - it will be renamed as:

```
((member (var X 1) (var Ys 1))  
  (append (var Zs 1)  
           (cons (var X 1) (var Xs 1))  
           (var Ys 1))))
```

and the variable counter (**1**) will be incremented each time the rule is retrieved.

To support the renaming operation, each compound AST type has a method that computes the list of variable terms it contains.

- `rule->vars`
- `atomic-formula->vars`
- `predication->vars`
- `term->vars`
- `query->vars`

All of these return a list of variables without repetition.

The Substitution ADT is a direct adaptation of the substitution-ADT module from the type inference system we saw in Chapter 3. It is translated in Scheme and adapted to the datatypes of LP: a substitution is a finite mapping from Variable to Term with occurs-check.

The ADT consists of:

- Constructor: `make-sub(variables, terms)`, which also checks for circularity (occurs-check).
- Getters: `sub->variables`, `sub->terms`, `sub->get-var(sub, var)` which returns the value of `var`, if defined, or error otherwise.
- Predicates: `sub?`, `empty-sub?`, `non-empty-sub?`, `sub-equal?`

### Operations:

- `extend-sub(sub, var, term)` which extends `sub` with the binding `var = term`
- Application of a substitution to LP terms, atomic formulas and queries:  
`sub-apply(sub, term)`
- Restriction of a substitution:  
`sub-restrict(sub, vars)`
- Substitution combination:  
`sub-combine(sub1, sub2)`

## Examples:

```
;; {T7 = Number, T8 = f(m(T5, Number), T3)}  
;; o {T5 = T7, T8 = Boolean}  
;; => {T5 = T7, T7 = Number, T8 = f(m(T7, Number), T3)}  
  
(define sub1  
  (make-sub '((var T7) (var T8))  
             '(Number (f (m (var T5) Number) (var T3)))))  
  
(define sub2  
  (make-sub '((var T5) (var T8))  
             '((var T7) Boolean)))  
  
(sub-combine sub1 sub2)  
;; '(sub ((var T5) (var T7) (var T8))  
        ((var T7) Number (f (m (var T7) Number) (var T3))))
```

## Examples:

```
;; f(X) o {X = 1} => f(1)
(sub-apply (make-sub '((var X)) '(1))
  (make-compound-term 'f '((var X))))
;; '(f 1)
```

The unifier is implemented as a term equations solver - that is, to unify 2 atomic-formulas, we split the atomic-formulas into a sequence of equations which pair the terms of each atomic formula one by one, and then solve the equations iteratively. When complex terms are met, they are split into more equations.

The Terms Equations ADT is just a pair data structure for two terms:

- Constructor:  
`make-equation(term1, term2)`
- Getters: `equation->left`,  
`equation->right`
- Predicates: `equation?`

The unification operation operates over atomic formulas and terms. This is an adaptation of the solve module from the type inference system. The unification algorithm uses the equation solving method:

1. For atomic elements – either compares if equal, different, or can create a substitution (non-circular);
2. For compound arguments with the same predicate or functor and the same arity, creates equations from corresponding elements, and repeats unification.
3. Main procedure: **unify-formulas**

4. Equation solvers:

```
solve-equations(equation-list),  
solve(equations, substitution)
```

5. Helpers:

```
unifiable-structure(equation),  
split-equation(equation)
```

A lazy tree is represented as a *lazy tree-list* whose head is the root-node and whose tail is a regular list of lazy-trees:

```
(root (lambda () (list lzt1 lzt2 ... lztn)))
```

This is a lazy representation for labeled trees with finite branching, but possibly infinite depth.

- `empty-lzt` represents the empty lazy-tree
- A leaf is represented by:  
`(root (lambda () empty-lzt))`

The ADT consists of:

- Constructors: `make-lzt`, `make-lzt-leaf`,  
`empty-lzt`,  
`expand-lzt(node, node-expander)`.
- Getters: `lzt->root`, `leaf-data`,  
`lzt->branches`, `lzt->first-branch`,  
`lzt->rest-branches`,  
`lzt->take-branches(lzt, n)`,  
`lzt->nth-level(lzt, n)`

The ADT consists of:

- Predicates: `empty-lzt?`, `lzt?`,  
`composite-lzt?`

The key operation to understand is  
**expand-lzt:**

```
(define expand-lzt
  (lambda (root node-expander)
    (let ((child-nodes (node-expander root)))
      (make-lzt root
        (lambda ()
          (map (lambda (node)
                  (expand-lzt node node-expander))
               child-nodes))))))
```

`expand-lzt` is the natural way to construct lazy trees. Consider the following example of a finite lazy tree:

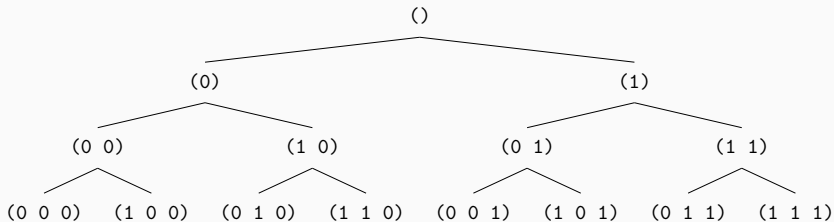
```
(define ft01
  (expand-lzt '()
    (lambda (node)
      (if (> (length node) 2)
          empty-lzt
          (map (lambda (n) (cons n node))
               '(0 1))))))
```

- The nodes in this lazy tree are labeled with a list of numbers (**0** or **1**).
- The root contains a list of length 0.
- The children of a node are of the form (**0 . parent**) and (**1 . parent**)
- When a node has a label with length 3, it has no children.

If we print all the node labels in this tree, we obtain:

```
'((  
  ((0) ((0 0) ((0 0 0)) ((1 0 0))) ((1 0) ((0 1 0)) ((1 1 0)))  
  ((1) ((0 1) ((0 0 1)) ((1 0 1))) ((1 1) ((0 1 1)) ((1 1 1)))
```

Which translates to:



`expand-lzt` describes a tree by providing a function that computes the direct children given a root node. Given this function, `expand-lzt` produces a lazy tree which generates the nodes level by level.

This method can be used to produce infinite trees naturally. Consider the following simple variation:

```
(define it01
  (expand-lzt '()
    (lambda (node)
      (map (lambda (n) (cons n node))
           '(0 1)))))
```

There are three procedures for scanning a lazy tree:

`lzt-filter(lzt, filterP)` – returns a list of nodes that satisfy the filter predicate; does not terminate on infinite lazy trees.

# LZT Operations

```
(define lzt-filter
  (lambda (lzt filterP)
    (letrec ((collect
              (lambda (lzt)
                (let ((children (flatmap collect
                                           (lzt->branches lzt))))
                  (if (filterP (lzt->root lzt))
                      (cons (lzt->root lzt) children)
                      children))))))
      (if (empty-lzt? lzt)
          empty
          (collect lzt)))))
```

`lzt-find-first(lzt, filterP)` – returns the first node that satisfies the filter predicate. Might not terminate for infinite lazy trees (if the path to the first success node is infinite).

`lzt-filter->lzl(lzt, filterP)` –  
returns a lazy list of all nodes that satisfy the  
filter predicate.

```
> (lzt-find-first it01 (lambda (node) (> (length node) 4)))  
'(0 0 0 0 0)
```

```
> (lzt-filter it01 (lambda (node) (< (length node) 4)))  
--> Infinite loop
```

```
> (lzt-filter->lzl it01 (lambda (node) (= (length node) 4)))  
--> A lazy list which returns 16 elements then loops forever
```

We are now ready to review the implementation of the `answer-query` algorithm in the LP Interpreter: The main functions of the `answer-query.rkt` module are:

- `answer-query`, which has two variants: `answer-query-first` and `answer-query-lzl`
- `LP-node-expander`, `expand-query`
- `Gsel`, `Rsel`

**answer-query** creates a proof tree as a lazy tree, whose nodes are labeled by a list of query and substitution. The substitution is the combination of all substitutions on the tree branches.

The nodes of the proof tree are defined in the data structure **PT-node**, with the getters:

- `PT-node->query`
- `PT-node->sub`

The proof tree is created using the `expand-lzt` constructor of lazy trees, using the procedure **LP-node-expander**, which performs the main actions of the LP interpreter:

1. Applying **Gsel** on the query.
2. Applying **Rsel** on the selected goal.
3. Creating the new queries for the child node.
4. Creating the new combined substitutions for the child nodes.

The code for **LP-node-expander** can be found [here](#).

The key functions of **Gsel** and **Rsel** rely on the ADTs for queries and rules defined in the AST.

The key functions of **Gsel** and **Rsel** rely on the ADTs for queries and rules defined in the AST.

Code can be found [here](#).

Observe that `Rsel` performs renaming of the rules when they are retrieved from the program. This is made easy by the encoding of renamed-variables as `(var X <n>)` and by the accessors `rule->vars` defined in the AST module.

The code for `rename-rule` can be found [here](#).

Eventually, given a lazy-tree data structure representing a proof-tree, we read off the answers to a query by producing a lazy-list of substitutions:

## Computing Answers Given a Proof-Tree

- The proof-tree is started using a root node with the query and an empty substitution
- **answer-query** expands the tree into a lazy tree representing the full proof-tree
- We extract from the proof-tree the success leaves, and for each one, read the substitution associated to the leaf and project it to keep only the variables that were present in the query.

The code for `answer-query`,  
`answer-query-first` and  
`answer-query-lzl` can be found [here](#).