Parsing and Canonical Unparsing

Gavin Mendel-Gleason

June 10, 2005

Abstract

Parsing and formatting have often been identified as problems that do not have elegant and widely accepted solutions in ANSI Common Lisp. Format strings tend to be parsimonious and can be very difficult to read. Destructuring primitives exist only for lists and their utility in parsing is further limited by the lack of non-deterministic features in Common Lisp. This paper demonstrates a simple implementation of non-deterministic parsing and canonical unparsing in Common Lisp. This method can provide a solution to both of these problems in a simple unified way. A single relation is constructed between an input string and a data-structure. This relation can provide a data-structure associated with a string, or construct a canonical string from the data-structure. While this type of solution may not meet performance requirements in some situations it is elegant, quite general and is easy to work with.

1 Introduction

Common Lisp uses `#'write` as a means of serializing objects to a string or stream and `#'read` for parsing characters from a stream into structured data. These two functions in conjunction provide a means for dealing with code migration and persistent data. Data that is not usually considered lisp code can also be read and written by providing read macros and custom printers. However there are fundamental limitations to this due to the way in which `#'read` is specified. It also requires users to carefully match the implementations of the serialization and parsing techniques.

In C, as well as many other languages, there exists a completely deterministic mechanism for parsing a sequence into the data-types of the language, as well as a mechanism for serializing them - namely, scanf() and printf(). Common Lisp has format directives which can mimic printf(), but does not have a mechanism akin to scanf()

Common Lisp is, however, well suited to the construction of sub-languages. Making use of a declarative logic sub-language which we call Twine, we can provide a specification that results in both the scanf() and printf() type behaviors as well as the parsing and unparsing of much more complex grammars than those specifiable strictly by concatenation.

2 The Sub-language

In order to achieve our goals we introduce the sub-language Twine. Twine is a declarative language in the spirit of prolog that interacts with the data-types of Common Lisp. The unification algorithm makes use of a slightly modified version of Marco Antoniotti's CL-Unification library. The code for Twine can be obtained from:

`http://www.evil-wire.org/~jacobian/`

2.1 Unification

Unification is a powerful form of pattern matching that can simplify many programming tasks, especially in the domain of parsing. Unification is in some ways similar and a generalization of the destructuring facilities such as destructuring-bind that are provided in Common Lisp.

CL-Unification is a Common Lisp library implementing unification on Common Lisp data-types. It uses a template sub-language for specifying destructuring and partial structures. For our purposes we

extend CL-Unification with the capacity to unify templates against other templates (it already has this feature for lists but has yet to include it for some other data-types) and we introduce the dual [1] of the destructuring operation which we will call `#'concretize`.

Let us take a quick look at how unify is used:

```
(setf *env* (unify '?x '(?y . ?z)))
(setf *env* (unify '?y 'a *env*))
(setf *env* (unify '?z '(a b c) *env*))

(report-environment *env*)

?x => (?y . ?z)
?y => a
?z => (a b c)
```

This is probably familiar to anyone who has used prolog or read through Norvig's PAIP [9].

The function `#'concretize` dereferences all bindings that were produced such that the most complete possible structure results. The function `#'report-environment` concretizes all values in an environment and displays them.

```
(report-environment *env*)

?x => (a a b c)
?y => a
?z => (a b c)
```

Some implementations of unification do a partial concretization as the unification algorithm proceeds which is allowable because the effects of unifying are strictly monotonic, that is to say that reordering a series of unifications has no effect on the outcome. This is often true with destructive unification algorithms.

Since we will be dealing with strings let us look at an example using the string template provided by CL-Unfication.

```
(setf *env* (unify "this is a test"
                   #T(string "this" &rest ?tail)))

(report-environment *env*)
?tail => " is a test"
```

One desirable feature (which is not very difficult to add) which is present in most newer unification algorithms is the ability to deal with circularity. Witness the following scenario:

```
(setf *env* (unify '?x '(f ?a ?b)))
(setf *env* (unify '?a '?x *env*))
(setf *env* (unify '?b '?x *env*))
```

The circularity can be consistently interpreted as meaning that the structure contains circularities. Since this does not have any effect on the monotonicity of unification and it allows a completely declarative model without side effects to create and destructure graphs, rather than limiting us strictly to trees, it is a highly desirable feature. It will, of course, not work on structures that are by definition linear e.g. strings.

The resultant structure after concretizing `'?x` in the above is shown below.

```
?x => #1=(f #1# #1#)
```

[1] It is dual in the sense that destructuring can be seen as projective arrows and the dual the identical operation with arrows reversed.

We do not make use of higher order unification in this paper but it might interest some Common Lisp users that a special case of higher order unification is discussed by Henry Baker [1] where he remarks on the desirability of equality of a class of functions with η-convertible functions identified. Though the problem is in general undecidable A large class of lambdas can be identified efficiently [7] and it is even possible to find unique most general unifiers [2].

2.2 Logic Programming

Programs in Twine are written as conjunctions and disjunctions of bindings and predicates. A conjunction is denoted with the operator `&`, disjunction with the operator + and binding with the operator =.

Conjunction requires that all clauses are provable while disjunction requires that at least one clause is provable.

The empty query (+) is false and can be associated with the failed substitution, that is the impossibility of an environment existing with bindings that satisfies the expression.

The empty query (`&`) is true, and is associated with the empty substitution, that is an environment that does not require any bindings at all in order to satisfy the expression.

The binding operator extends the current environment to one in which its two arguments are identified or it becomes the failed substitution, a value synonymous with (+) or false.

Readers interested in relational algebra might note the close connection between these true and false environments (or substitutions) and C. J. Date's tabledee and tabledum [3]. In fact tuples in the relational algebra can be viewed as admissible substitutions and tables as disjunctions over these tuples to obtain a very close correspondence.

Predicates are named combinations of +,`&`,= and other predicates, and may include themselves in their body. We introduce new predicates with the defp macro. The following is an example of how a predicate that implements a simple database of facts might be written.

```
(defp parent (?child ?parent)
  (+ (& (= ?parent Chronos)
        (= ?child Zeus))
     (& (= ?parent Zeus)
        (= ?child Athena))))
```

In order to construct a query using this predicate we use the macro "top" which returns (continuation . environment). For example:

```
(report-environment (cdr (top (parent ?x ?y))))
?x => Chronos
?y => Zeus

(report-environment (cdr (top (parent ?x Athena))))
?x => Zeus
```

There are many mechanisms for implementing such declarative languages and many tradeoffs in their design. The Twine implementation focuses on simplicity of design and uses techniques that are similar to those that are used in Kanren [4].

3 Parsing and Unparsing

We can now look at a simple parsing and unparsing problem in Twine. Imagine that we want to output the concatenation described by the string `"~Dfoo~D"`. We can write this in the following way:

```
(defp int-foo-int (?int ?s0 ?sn)
  (& (int<->string ?int ?s0 ?si)
     (= ?si #T(string "foo" &rest ?sj))
     (int<->string ?int ?sj ?sn)))
```

The `?s0 ?sn` are difference lists. They help us keep track of how much of the problem we are currently working on. The head ?s0 and tail ?`?sn` do not need to be bound in order for regions in the middle to be computed. We may now query int-foo-int in the following way.

```
(report-environment (cdr (top (int-foo-int 3 ?s0 ""))))
?s0 => "3foo3"

(report-environment (cdr (top (int-foo-int ?int "3foo3" ""))))
?int => 3

(top (int-foo-int ?int "3foo4" "")) => False
```

We have defined both a reader and a writer for our deterministic language. If in fact there is ambiguity in the predicate `int<->string` which we have glossed over, it can make itself known as a non-determinism. If it is required that the application have no ambiguity, it is simply necessary to check that calling the returned continuation results in a falsehood, which is identical to establishing a uniqueness quantification over the result.

```
(funcall (car (top (int-foo-int ?int "3foo3" "")))) => false
```

3.1 Macros

In order to make our parsing routines easier to write we make use of Twine's macro-system to define the "alphabetic" and "non-alphabetic" macros.

```
(defparameter *alphabetic*
  (mapcar #'code-char (append (loop for elt from (char-code #\a)
    below (char-code #\Z)
    collect elt)
      (loop for elt from (char-code #\A)
    below (char-code #\|)
    collect elt))))

(defm alphabetic (char)
  `(+ ,@(loop for c in *alphabetic*
      collect `(= ,char ,c))))

(defm non-alphabetic (char)
  `(+ ,@(loop for c in (set-difference *characters* *alphabetic*)
      collect `(= ,char ,c))))
```

The macro system uses Common Lisp as the language by which to construct the code forms in a way familiar to Common Lisp programmers.

3.2 More Complex Grammars

As an example of a more complex grammar we use an extreme subset of HTML where no tags are verified. The idea is simply to make a relation R between two entities xRy such that we can supply either x and obtain y or the reverse or supply both in order to verify that in fact the relation holds. The relation that we are concerned with is the predicate "document".

```
(defp chars (?s0 ?sn ?t0 ?tn)
  (+ (& (= ?s0 "")
        (= ?sn "")
        (= ?t0 "")
        (= ?tn ""))
     (& (= ?s0 #T(string ?char &rest ?sm))
```

```
        (= ?t0 #T(string ?char &rest ?tm))
        (alphabetic ?char)
        (chars ?sm ?sn ?tm ?tn))
     (& (= ?sn #T(string ?char &rest ?sm))
        (= ?s0 ?sn)
        (non-alphabetic ?char)
        (= ?t0 "")
        (= ?tn ""))))

(defp word (?s0 ?sn ?word)
  (& (chars ?s0 ?sn ?word "")
     (= ?word #T(string ?char &rest ?anything))))

(defp tag (?s0 ?sn ?tag)
  (& (= ?s0 #T(string #\< &rest ?sj))
     (word ?sj ?sk ?tag)
     (= ?sk #T(string #\> &rest ?sn))))

(defp endtag (?s0 ?sn ?tag)
  (& (= ?s0 #T(string #\< &rest ?sj))
     (= ?sj #T(string #\/ &rest ?sk))
     (word ?sk ?sl ?tag)
     (= ?sl #T(string #\> &rest ?sn))))

(defp elements (?s0 ?sn ?list)
  (+ (& (= ?list (?head . ?rest))
        (element ?s0 ?sj ?head)
        (elements ?sj ?sn ?rest))
     (& (= ?list nil)
        (= ?s0 ?sn))))

(defp element (?s0 ?sn ?list)
  (+ (& (= ?list (?word))
        (word ?s0 ?sn ?word))
     (& (= ?list (?tag . ?rest))
        (tag ?s0 ?sj ?tag)
        (endtag ?sk ?sn ?tag)
        (elements ?sj ?sk ?rest))))

(defp document (?string ?list)
  (elements ?string "" ?list))
```

This simple program defines a mechanism for parsing HTML into structured data, using the predicate "document". The serialization of the structured lists exists naturally as a consequence of the definition, with the caveat that some care is needed to make sure that termination conditions are met in both directions.

The code basically forms a declarative clause grammar extended by an argument which holds the structural information and therefor could be written in an even more terse fashion abstracting over the fact that the string structure is completely linear. The macros for performing this transformation are left as an exercise to the interested reader.

An example of the execution of "document" is as follows.

```
(report-environment
 (cdr
  (top
   (document
```

```
    "<body><head>asdf</head><title>fdsa</title></body>"
    ?list))))

?LIST => (("body" ("head" ("asdf")) ("title" ("fdsa"))))

(report-environment
 (cdr
  (top
   (document
    ?s0
    (("body" ("head" ("asdf")) ("title" ("fdsa")))))))))

?S0 => "<body><head>asdf</head><title>fdsa</title></body>"
```

Running the "document" query in order to produce our serialization, guarantees that our answer belongs to the language as described by our parsing rules. As long as we are careful with our parsing rules, we can safely produce correct serializations.

In general either the AST or the string may not have unique instantiations that satisfy the relation. This is elegantly dealt with via non-determinism. Further instantiations can be obtained by calling the continuation that is returned from the query. The canonical unparse of a structure is the first result that occurs from running the query. This element is determined uniquely as code in Twine executes using a deterministic algorithm. This of course is not the case with all interpreters - notably those that make use of implicit parallelization. In these interpreters it may be difficult to obtain a canonical form.

4 Future Directions

Our declarative language has several deficiencies some of which have quite elegant solutions but provide a much more complex and more difficult to understand implementation. Perhaps most importantly, left-recursive definitions do not terminate in Twine. This problem can be easily solved with the addition of a technique known as tabling [10]. This makes evaluation of rules which are written left recursively even more efficient then our current implementation as it mimics qualities of Earley's parser algorithm, providing a mix of top down and bottom up evaluation.

Also of concern is the lack of "fair choice" as Twine always attempts to try the most local choices which can result in a larger class of non-terminating queries than languages which exhibit the property of "fair choice". Again this is solvable [6] but requires a slightly less transparent implementation.

The use of higher order techniques is also desirable as many features can be abstracted. Kleene Star for instance is easily represented if predicates can be passed in logic variables. This however leads one naturally into considering the results of unification between two predicates bound as logic variables which requires higher order unification.

In a real world implementation of a non-deterministic language that is used for serious parsing it is essential that some form of determinism detection or annotation is provided which can reduce the space usage that can result from unbridled use of non-determinism. Every + clause in Twine grows the stack in a manner which can not be removed without careful analysis. This can lead to performance degradation on large problems. For a brief survey of pruning techniques see [8].

The implementation of `int<->string` does not actually exist. It can be written in principle using an inductive construction of the integers but it would be desirable to implement it in terms of the lisp integer data-type. In order to do this it is necessary to either provide it as a primitive, provide arithmetic predicates as primitives or provide a mechanism for introducing computed queries. The latter is probably the most desirable but it also introduces the need for a more sophisticated evaluation engine. Namely more complex evaluation strategies such as residuation would become desirable [5]. It is likely that some mechanism for declaring or automatically determining the cardinality of predicates based on their signatures would also be desirable.

It is intended that the next update of Twine will include an implementation will include a common-lisp reader/writer relation as a primitive that will solve the particular problem of writing `int<->string` and predicates like it.

References

[1] Henry Baker. Equal rights for functional objects or, the more things change, the more they are the same. *OOPSM: OOPS Messenger (Special Interest Group on Programming Languages)*, 4, 1993.

[2] Michael Beeson. Unification in lambda calculus with if-then-else. In *Proceedings of the Fifteenth Conference on Automated Deduction (CADE-15)*, pages 96–111. Springer-Verlag, 1998.

[3] C. J. Date and Hugh Darwin. *Foundation for Future Database Systems: The Third Manifest, 2nd edition*. Addison-Wesley, Reading, Massachusetts, 2000.

[4] Daniel P. Friedman and Oleg Kiselyov. A logic system with first class relations. unpublished manuscript, 2004.

[5] Michael Hanus, Sergio Antoy, Herbert Kuchen, Francisco J. Lopez-Fraguas, Wolfgang Lux, Juan Jose Moreno Navarro, and Frank Steiner. Curry, an integrated functional logic language, 2003.

[6] Oleg Kiselyov, Chung chieh Shan, Daniel P. Friedman, and Amr Sabry. Functional pearl: backtracking, interleaving, and terminating monad transformers. In *Proceedings of the International Conference on Functional Programming*, 2005.

[7] Gopalan Nadathur. A treatment of higher order features in logic programming. *To appear in Theory and Practice of Logic Programming (TPLP)*, 2004.

[8] Lee Naish. Pruning in logic programming, 1995. advanced tutorial at International Conference on Logic Programming in Japan, 13-16 June, 1995.

[9] Peter Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common LISP*. Morgan Kaufmann, San Francisco, CA, 1991.

[10] I. V. Ramakrishnan, Prasad Rao, Konstantinos F. Sagonas, Terrance Swift, and David Scott Warren. Efficient tabling mechanisms for logic programs. In *International Conference on Logic Programming*, pages 697–711, 1995.