Deforesting LF

Anonymous

March 17, 2007

1 Introduction

As a preliminary, this paper shows (a) how a syntactic theory that incorporates a level of *logical form* (LF) can be applied fairly directly in a parser. This demonstration, while not strictly novel (Schubert and Pelletier, 1982; Shieber and Johnson, 1993) is nonetheless vital for cognitive scientists and others who wish to maintain the Competence Hypothesis (1) about the grammar-parser relationship.

Hypothesis 1 (Competence Hypothesis) *A reasonable model of language use will incorporate, as a basic component, the generative grammar that expresses the speaker-hearer's knowledge of the language (Chomsky, 1965, 9)*

The main result is that (b) LF can be eliminated in such a competence-based processor using a program-transformation technique called deforestation (Wadler, 1990). 'Eliminated' in this sense means that it is possible to derive another parser that does not build any LF representations — a program that does not allocate any data of type LF. Such a formal result suggests that any evidence for LF in processing could always be explained by an equivalent theory without LF. Representations at that level, therefore, should not be viewed as causally implicated in sentence processing, despite suggestions to the contrary in the literature.

> There is good evidence that the data structures or units of representation posited by theories of transformational grammar are actually implicated causally in online language processing.
>
> (Berwick and Weinberg, 1984, 197)

This paper's main argument is most clearly stated in a terminology that distinguishes between *computational/1st-level* theories and *algorithmic/2nd-level* theories (Marr, 1982; Stabler, 1983; Pylyshyn, 1984). Theories in the former category specify an information-processing task by defining the function that is to be computed. This is the conventional interpretation for theories of linguistic competence. Theories in the latter category are proposals about the way in which calculation of the function proceeds. This is the conventional interpretation of psycholinguistic theories, for instance, of human sentence comprehension. The claim that an LF-using parser can be deforested into an non-LF-using one is a claim about theories at this latter level. The claim is that for every *algorithmic/2nd-level* theory of natural language understanding that specifies an intermediate stopover at LF, there exists an equivalent theory at the same level that does not require such a stopover. To the extent that the argument is convincing, it casts doubt on two prominent hypotheses (2,3) about the role of grammar in processing

Hypothesis 2 (No Levels are Irrelevant) *To understand a sentence it is first necessary to reconstruct its analysis on each linguistic level (Chomsky, 1957, 87)*

Hypothesis 3 (LF Hypothesis) *LF is the level of syntactic representation that is interpreted by semantic rules (Larson and Segal, 1995, 248).*

by exhibiting a counterexample where understanding does not require reconstruction of the LF level, and where semantic rules would not apply at LF.

To make the argument, the paper develops a program that calculates formulas of a first-order logic for a fragment of English. The goal is to reflect as faithfully as possible the computations specified by transformational grammar along the deductive path from *surface structure* (SS) or *Spell-Out* to LF. Section 2 reviews how, in the functional programming language Caml (Weis and Leroy, 1999; Cousineau and Mauny, 1998) combinator-parser techniques (Burge, 1975; Frost and Launchbury, 1989) yield an SS parser whose definition mirrors the grammar it is a parser for. Section 3 expresses May's (1977) *Quantifier Raising* (QR) idea as a simple Caml function. In combination with section 2, this leads to a natural language understanding program that can handle the core data that motivated QR in the first place. Then, section 4 shows how deforestation leads to an equivalent natural language understanding program that does not use LF.

2 A combinator parser for surface structure

Parsers can be expressed as functions from lists of words (symbols, lexical items, et cetera) to new remainder-lists paired with tree-structure (Leermakers, 1993). This tree-structural output typically encodes the derivation, on the grammar, of whatever elements the parsing function was able to chew off of the given word-list argument, but in fact the technique itself can accommodate outputs of any type.

This view leads naturally to a conception of a parser as a mutually-recursive set of functions, each named for a distinct nonterminal. As in Definite Clause Grammars, one must acknowledge that nonterminals are fundamentally *relations* between input strings and data structures that aggregate their suffixes with outputs. The Caml type definition in Listing 1 acknowledges this by specifying that an analyzer's output type is a sequence (Seq.t) of outcomes. The backquote indicates a type variable in a polymorphic data type.

```
(* an outcome is a result and a remaining symbol list *)
type ('symbol,'result) outcome = 'result * ('symbol list)

(* an analyzer is a function from remaining symbols to a sequence of outcomes *)
type ('symbol,'result) analyzer = 'symbol list -> ('symbol, 'result) outcome Seq.t
```

Listing 1: Parser type

Higher-order functions – combinators – combine simpler parsers into more complex ones. These modes of combination are not necessarily restricted to those expressible in context-free grammar as van Eijck (2003) demonstrates. However, Listing 2 exhibits just the standard complement of alternation, sequencing and result-conversion. These combinators can be given shorter, infixable names |. , &. and >. respectively.

```
(* either p1 can parse the input, or p2 can *)
let (alternative : ('a,'b) analyzer -> ('a,'b) analyzer -> ('a,'b) analyzer) = fun p1 p2 inp
  -> Seq.append ((p1 inp),(p2 inp))

(* use p2 to parse all the remainders p1 leaves behind. pair-up the results *)
let (sequence : ('a,'b) analyzer -> ('a,'c) analyzer -> ('a,('b * 'c)) analyzer) = fun p1 p2 inp ->
          Seq.multiply (function (result1,remainder1) ->
                              Seq.map
                                (function (result2,remainder2) -> ((result1,result2),remainder2))
                                (p2 remainder1)
                             ) (* outermost function takes individual outcomes to sequences of outcomes *)
            (p1 inp)

(* map the function f : 'b->'c over the result of parsing inp with p *)
let (gives : ('a,'b) analyzer -> ('b -> 'c) -> ('a,'c) analyzer)= fun p f inp ->
  Seq.map (function (result,remainder) -> (f result,remainder)) (p inp)
```

Listing 2: Combinators

Adding a tree data type `type 'a tree = Node of 'a * 'a tree list` and some constructor abbreviations branch0-branch3 (not listed) yields[1] a parser (Listing 3) that finds the two analyses depicted in Figure 1.

```
let rec s words = (dp &. vp >. branch2 "s") words
and dp words = ((d &. np >. branch2 "dp") |. (pn >. branch1 "dp")) words
and d = term ["every";"some"] >. branch1 "d"
and pn = term ["cecil";"dexter";"john"] >. branch1 "pn"
and np words = (((kstar adj) &. nbar) >. function (premodifiers,n1) -> Node("np",premodifiers@[n1])) words
and nbar words = (adjoin2 pp "nbar" n) words
and n = term ["city";"body";"scale";"man";"woman"] >. branch1 "n"
and adj = term ["italian"] >. branch1 "adj"
and vp words = (((v &. dp) >. branch2 "vp") |. ((v &. dp &. pp) >. branch3 "vp")) words
and pp words = (((p &. dp) >. branch2 "pp")) words
and p = term ["in"] >. branch1 "p"
and v = term ["met";"saw";"played";"loves"] >. branch1 "v"
```

Listing 3: Combinator parser

3 Quantifier Raising

Quantifier Raising is an adjunction transformation proposed by May (1977) as part of a syntactic account of quantifier scope ambiguities. An economical theory accounting for both quantifier scope and WH-movement could postulate a single transformational rule — Move-α — that is uniformly subject to the same constraints, but links different levels of representation in a grammar.

[1]The kstar and adjoin2 combinators are not difficult but would take up too much space in this extended abstract.

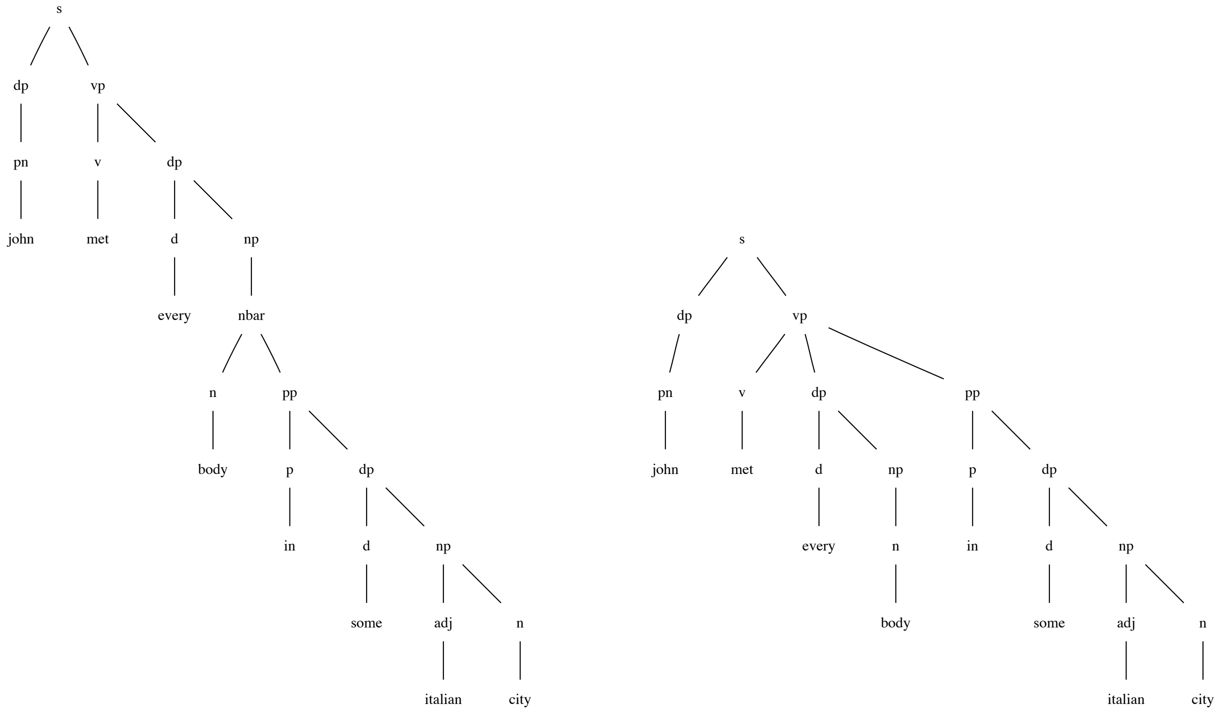

Figure 1: Two analyses returned by the parser in Listing 3

May's QR rule applies freely to adjoin quantified phrases to the root, subject to the constraint that the moved phrase must c-command its trace. However, viewing QR as a function from lists of quantified nodes to new linear orderings for those nodes (at the front of the sentence) makes it clear that the c-command condition can equivalently be enforced on lists of Gorn addresses. One need only avoid permutations that order a super-constituent before its sub-constituents (Listing 4).

```
let rec improperly_bound = function
    [] -> false
  | x::xs -> (List. fold_left (fun prev cur -> prev || ( is_prefix x cur)) (improperly_bound xs) xs)

let candidate_quantifier_orderings ss =
  Seq.map (function x -> (ss,x))
    (Seq. filter (function x -> not(improperly_bound x)) (Seq.fromList (permutations (where_are_QDPs ss))))

let rec qr (ss, places) = match places with
    [] -> ss
  | p::ps ->
      let id = global_counter () in
      let remnant = substitute (indexwith id (Node("t",[]))) p ss in
      Node ( (label ss ),    (* adjunction: same label as old root *)
             (indexwith id (at ss p)) (* adjunction: becomes new first daughter *)
             ::
               [(qr (remnant,ps))] ) (* process the other daughter *)
```

Listing 4: Proper binding enforced on permutations of Gorn addresses

The functions sketched in this section can be combined in an end-to-end natural language understander as in Listing 5.

```
let analyze ssToLFprime inp =
    (* parser returns only analyses that are finished=span all given input*)
  let surfaceStructures pf = Seq.map Pervasives.fst (Seq. filter finished (s (lex pf))) in
    (* logical expression simplifier *)
  let like_prenex lfprime = Seq.map simplify lfprime in
    like_prenex (Seq.multiply ssToLFprime (surfaceStructures inp))

let withLF ss = Seq.map convert (Seq.map qr (candidate_quantifier_orderings ss))
let withoutLF ss = Seq.map f (candidate_quantifier_orderings ss)
```

Listing 5: Abstract analyzer with plug-in that uses LF (section 4 derives f)

The program in Listing 5 faithfully calculates interactions between the constraints that May's competence theory specifies:

```
# Seq.iter formula (analyze withLF "john met every body in some italian city");;
‘exists v25[forall v26[(italian-city(v25) & (body-in(v26,v25) -> john-met(v26)))]]’
‘forall v27[exists v28[(body(v27) -> (italian-city(v28) & john-met-in(v28,v27)))]]’
‘exists v29[forall v30[(italian-city(v29) & (body(v30) -> john-met-in(v29,v30)))]]’
- : unit = ()
# Seq.iter formula (analyze withLF "every body in some italian city met john");;
‘exists v31[forall v32[(italian-city(v31) & (body-in(v32,v31) -> met-john(v32)))]]’
- : unit = ()
```

Executing the analyze withLF function causes intermediate data structures to be created. In particular, it allocates a sequence of quantifier-raised syntactic structures. Is there an equivalent program that does not allocate this intermediate data? Section 4 answers this question in the affirmative.

4 How deforestation applies to the parser with QR

The function candidate_quantifier_orderings multiplies out all nondeterminism in advance so that the definition of withLF (Listing 5) has the form map g (map h sequence_of_things) which is equivalent to mapping the composition of g with h. The composition would be: map $\underbrace{\lambda x.\,[g(h\,x)]}_{DeforestMe}$ sequence_of_things

In the case at hand, x pairs a surface structure and a Gorn-addressed list of quantified nodes. Thus, the program transformation begins with

```
let f = function (ss,places) -> convert (qr (ss,places))
```

Listing 6: Caml definition of the form $DeforestMe$ with $x = (\texttt{ss}, \texttt{places})$, $g = \texttt{convert}$, $h = \texttt{qr}$

Rule numbers in this section refer to Wadler's program transformation rules.

```
(* 3: replace convert with its definition *)
let f = function (ss,places) -> match (qr (ss,places)) with
    Node (_,(Node (_,(Node ("d",[Node ("every" ,[])])):: np::_) as k1)::k2::_) ->
      begin
        match (indexof k1) with (* k1 is an indexed [XP [D every ] ... ], k2 her right-sister *)
            Some numericstring -> Quant(Universal,numericstring,Impl ((predicateify [numericstring] np),(convert k2)))
          | None -> (* first daughter exhaustively dominates "every" but is not indexed *)
              let numericstring = "novariable" in Quant(Universal,numericstring,Impl ((predicateify [] np),(convert k2)))
      end
  | here -> predicateify [] here (* any node whose 1st daughter is not a quantified DP *)

(* 6: replace match on a function application with its definition *)
let f  = function (ss,places) -> match (match places with
            [] -> ss
          | p::ps ->
              let id = global_counter () in
              let remnant = substitute (indexwith id (Node("t",[]))) p ss in
                Node ( (label ss),
                       (indexwith id (at ss p))
                       ::
                         [(qr (remnant,ps))]
                     )
      ) with
      Node (_,(Node (_,(Node ("d",[Node ("every" ,[])])):: np::_) as k1)::k2::_) -> ...
    | here -> predicateify [] here

(* 7: broadcast inner case statement outwards *)
let f = function (ss,places) -> match places with
     [] -> match ss with
            Node (_,(Node (_,(Node ("d",[Node ("every" ,[])])):: np::_) as k1)::k2::_) -> ...
          | here -> predicateify [] here
  | p::ps -> match (let id = global_counter () in
                let remnant = substitute (indexwith id (Node("t",[]))) p ss in
                  Node ( (label ss),    (* adjunction: same label as old root *)
                         (indexwith id (at ss p)) (* adjunction: becomes new first daughter *)
                         ::
                           [(qr (remnant,ps))] (* process the other daughter *)
                       )
              ) with
            Node (_,(Node (_,(Node ("d",[Node ("every" ,[])])):: np::_) as k1)::k2::_) -> ...
          | here -> predicateify [] here

(* hoisting lets reveals we are matching on a constructor *)
(* 5:  matching on a constructor. use fact that NP is second daughter of DP *)
let f = function (ss,places) -> match places with
```

```
    [] −> match ss with
              Node (_,(Node (_,(Node ("d",[Node ("every" ,[])])):: np::_) as k1)::k2::_) −> ...
            | here −> predicateify [] here
    end
  | p::ps −> let id = global_counter () in
               let remnant = substitute (indexwith id (Node("t",[]))) p ss in
               match (indexof (indexwith id (at ss p))) with
               Some numericstring −>
                  Quant(Universal,numericstring,Impl ((predicateify [numericstring] (at ss (p@[1]))),(convert (qr (remnant,ps)))))
             | None −> let numericstring = "novariable" in
                  Quant(Universal,numericstring,Impl ((predicateify [] (at ss (p@[1]))),(convert (qr (remnant,ps)))))
(* indexof (indexwith n t)  = Some (string_of_int n) for all trees t and integers n *)
(* 5: matching on the Some constructor *)
let f = function (ss,places) −> match places with
    [] −> match ss with
              Node (_,(Node (_,(Node ("d",[Node ("every" ,[])])):: np::_) as k1)::k2::_) −> ...
            | here −> predicateify [] here
  | p::ps −> let id = global_counter () in
               let remnant = substitute (indexwith id (Node("t",[]))) p ss in
               Quant(Universal,(string_of_int id),Impl ((predicateify [(string_of_int id)] (at ss (p@[1]))),(convert (qr (remnant,ps)))))
        end
(* knot−tie in the nonempty case of outermost match: realize that we started by translating (convert (qr (ss,places))) *)
let rec f = function (ss,places) −> match places with
    [] −> match ss with
              Node (_,(Node (_,(Node ("d",[Node ("every" ,[])])):: np::_) as k1)::k2::_) −> ...
            | here −> predicateify [] here
  | p::ps −>
        begin
          let id = global_counter () in
          let remnant = substitute (indexwith id (Node("t",[]))) p ss in
            Quant(Universal,(string_of_int id),Impl ((predicateify [(string_of_int id)] (at ss (p@[1]))),f (remnant,ps)))
        end
(* if there are any tree nodes in ss labeled with DP, candidate_quantifier_orderings will have listed them in places
   and the second clause of the outermost match will have replaced them with traces by the time the first clause matches
   thus there cannot ever be a match on the first clause of the inner match inside [] −> *)
let rec f = function (ss,places) −> match places with
    [] −> match ss with
             here −> predicateify [] here
  | p::ps −>
        begin
          let id = global_counter () in
          let remnant = substitute (indexwith id (Node("t",[]))) p ss in
            Quant(Universal,(string_of_int id),Impl ((predicateify [(string_of_int id)] (at ss (p@[1]))),f (remnant,ps)))
        end
(* simplify: match imposes no constraint and has only one case *)
let rec f = function (ss,places) −> match places with
    [] −> predicateify []  ss
  | p::ps −>
        begin
          let id = global_counter () in
          let remnant = substitute (indexwith id (Node("t",[]))) p ss in
            Quant(Universal,(string_of_int id),Impl ((predicateify [(string_of_int id)] (at ss (p@[1]))),f (remnant,ps)))
        end
```

Listing 7: Derivation of function f : string tree ∗ int list list → formula that does not use LF

References

Berwick, Robert C. and Amy S. Weinberg. 1984. The Grammatical Basis of Linguistic Performance. MIT Press.

Burge, William H. 1975. Recursive Programming Techniques. Addison-Wesley.

Chomsky, Noam. 1957. Syntactic Structures. Mouton de Gruyter.

Chomsky, Noam. 1965. Aspects of the Theory of Syntax. MIT Press.

Cousineau, Guy and Michel Mauny. 1998. The Functional Approach to Programming. Cambridge University Press. First published in French by Ediscience in 1995.

Frost, R. and J. Launchbury. 1989. Constructing natural language interpreters in a lazy functional language. The Computer Journal, 32(2):108–121.

Larson, Richard and Gabriel Segal. 1995. Knowledge of Meaning. MIT Press.

Leermakers, René. 1993. The Functional Treatment of Parsing. Kluwer.

Marr, David. 1982. Vision: A computational investigation into the human representation and processing of visual information. W.H. Freeman and Company.

May, Robert. 1977. The grammar of quantification. Ph.D. thesis, MIT.

Pylyshyn, Zenon W. 1984. Computation and Cognition: Toward a Foundation for Cognitive Science. MIT Press.

Schubert, Lenhart K. and Francis Jeffry Pelletier. 1982. From English to logic: context-free computation of 'conventional' logical translation. Computational Linguistics, 8(1):26–44.

Shieber, Stuart and Mark Johnson. 1993. Variations on incremental interpretation. Journal of Psycholinguistic Research, 22(2):287–318.

Stabler, Jr., Edward P. 1983. How are grammars represented? Behavioral and Brain Sciences, 6:391–421.

van Eijck, Jan. 2003. Parser combinators for extraction. In Proceedings of the 14th Amsterdam Colloquium.

Wadler, Philip. 1990. Deforestation: transforming programs to eliminate trees. Theoretical Computer Science, 73:231–248.

Weis, Pierre and Xavier Leroy. 1999. Le Langage Caml. Dunod.