ESSLLI 2007

19th European Summer School in Logic, Language and Information
August 6-17, 2007
http://www.cs.tcd.ie/esslli2007
Trinity College Dublin
Ireland

WORKSHOP PROCEEDINGS

ESSLLI is the Annual Summer School of FoLLI,
The Association for Logic, Language and Information
http://www.folli.org

Workshop on New Directions in Type-theoretic Grammars

August 6–10, 2007
Dublin, Ireland

Reinhard Muskens (ed.)

Preface

This volume collects the papers that were accepted for the Workshop on New Directions in Type-theoretic Grammars (NDTTG 2007), to be held from 6 to 10 August 2007, in Dublin, Ireland. The workshop is part of the European Summer School on Logic, Language and Information (ESSLLI 2007). I hope the reader of these papers will derive from them the intellectual pleasure that they gave me.

I wish to thank the members of the programme committee for their excellent reviewing. I also would like to gratefully acknowledge support from the Netherlands Organisation for Scientific Research (NWO).

June 2007

Reinhard Muskens
Workshop Organizer
NDTTG 2007

Workshop on New Directions in Type-theoretic Grammars

Workshop Purpose

In 1961 Haskell Curry published his by now famous paper on 'Some Logical Aspects of Grammatical Structure'. In this paper (large parts of which had already been written in the 1940's) he made a distinction between the 'tectogrammatics' and 'phenogrammatics' of language (a distinction similar to that between abstract syntax and concrete syntax in compiler theory), while also arguing against directionality in the type system used for language description. In 1953 Bar-Hillel had introduced a distinction between categories seeking material to their right and categories seeking material to the left. To date most categorial grammarians follow Bar-Hillel in this, but in Curry's architecture phenogrammatical structure can take care of word order, making directionality unnecessary.

Curry's proposal was part of a classical phase in categorial grammar that started with Ajdukiewicz's paper on syntactic connexity and also included Joachim Lambek's pivotal work on the introduction of hypothetical reasoning. It led to many follow-ups. For example, in Richard Montague's work the tectogrammatics/phenogrammatics distinction reappeared as one between analysis trees and surface strings, while Montague also added a level of meaning as a third component. The grammatical architecture thus became one in which a central abstract component is interpreted on two levels. An explicit connection between Montague's set-up and that of Curry was given in David Dowty's work in the 1980's. Also in the 1980's, Aarne Ranta used the idea in a constructive type theory setting, while Reinhard Muskens used it for his Partial Montague Grammar and Johan van Benthem explored the logical and linguistic implications of LP*, the undirected version of the Lambek Calculus, or, in other words, the logic of simply typed linear lambda terms. Later years brought Richard Oehrle's insight that the interpreting levels of the theory (not only semantics but also phenogrammar) can be represented with the help of lambda terms. Since the central abstract component consists of LP* derivations in Oehrle's set-up, equivalent with linear lambda terms, in fact all levels of the grammar can now be represented with the help of lambda terms and the typed lambda calculus becomes the central mechanism for grammatical description (as it had been in Cresswell's lambda-categorial languages).

Since the turn of the century there has been a heightened activity within a series of type-theoretical formalisms bearing a family resemblance to one another. All of these adopt the pheno/tecto distinction or undirectedness in one way or another and claim various descriptive and formal advantages. We mention Abstract Categorial Grammars (de Groote), De Saussure Grammar (Kracht), Minimalist Categorial Grammars (Lecomte, Retore), Lambda Grammars (Muskens), Higher Order Grammar (Pollard), and the Grammatical Framework (Ranta).

The workshop intends to bring together researchers in this now very active field. It aims to provide a forum for advanced PhD students and researchers, enabling them to present their work and to discuss it with colleagues who work in the broad subject areas represented at ESSLLI.

Workshop Organizer

Reinhard Muskens

Invited Speakers

David Dowty
Richard Oehrle

Programme Committee

Johan van Benthem
Nissim Francez
Philippe de Groote
Makoto Kanazawa
Marcus Kracht
Alain Lecomte
Glyn Morrill
Richard Oehrle
Carl Pollard
Aarne Ranta
Christian Retoré
Yoad Winter

Table of Contents

VI

TBA

David Dowty

Ohio State University

Dimensions of Grammatical Deduction

Dick Oehrle

Cataphora, Inc.
`rto@cataphora.com`

In my earlier work on multi-dimensional categorial type systems [1, 2], I described some systems of grammatical deduction in which each deductive step acts simultaneously, in accordance with general principles modeled directly on the Curry-Howard isomorphism, in three dimensions (corresponding to syntax, phonology, semantics). The immediate goal of this work was to provide an architecture in which quantifier scope distinctions arise as a result of two steps: (1) choosing a a set of compatible dimension-specific type systems, and (2) assigning an appropriate single type to each quantifier or determiner. Neither of these choices makes specific reference to scoping phenomena. Thus, this is a step (hardly the final step) toward a theory of grammatical composition in which scope ambiguities arise as an ineluctable form of proof indeterminism—not as an ad hoc widget of one kind or another that describes scope ambiguities without addressing the question of how they should arise in the first place.

The essential difference between the multi-dimensional categorial architecture that I described and more traditional systems of that time involves the interplay among different dimensions. In the traditional systems, grammatical deduction is defined with respect to the type logic of a single designated dimension (syntax). Properties of the other dimensions can be assigned with respect to syntactic proofs. Thus, while the other dimensions may support forms of type inference themselves, these properties are essentially passive: the syntactic dimension provides the proof; the proof determines the properties of the non-syntactic dimensions; in some cases, these properties might depend on a step by step correspondence like the Curry-Howard isomorphism; in other cases, the correspondence might rest on different principles.

In this paper, I first focus on these architectural differences. I then consider how these differences apply to a range of questions which bear on the fundamental notion of compositionality. Among the issues that arise are these:

- given the interaction of a number of type systems, is there just a single compositional driver? [I argue: no.]
- how many dimensions should there be? [I consider the compositional properties of sentences containing expressive elements like who left the stupid door open?, where the emotive term stupid need not be taken to be a modifier of door.]
- what kind of interaction across dimensions should we expect to find in various forms of quotation, names, and metalinguistic language? [I wish to be able to treat the fact that quotation is protected in some ways, but accessible in other ways, so that it is possible to draw on information inside quotations

for such phenomena as VP-ellipsis: Lee said, "I don't give a hoot", and he didn't.

References

1. Richard T. Oehrle. 1994. Term-Labeled Categorial Type Systems. *Linguistics & Philosophy.***17**.6.633-678.
2. Dick Oehrle. 1995. Some 3-Dimensional Systems of Labelled Deduction. *Bulletin of the IGPL.***3**.2-3.429-448.

The GF Grammar Compiler

Aarne Ranta

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University

Abstract. GF (Grammatical Framework) is a grammar formalism based on the distinction between abstract and concrete syntax. An abstract syntax is a free algebra of trees, and a concrete syntax is a mapping from trees to nested records of strings and features. These mappings are naturally defined as functions in a functional programming language; the GF language provides the customary functional programming constructs such as algebraic data types, pattern matching, and higher-order functions, which enable productive grammar writing and linguistic generalizations. Given the seemingly transformational power of the GF language, its computational properties are not obvious. However, all grammars written in GF can be compiled into a simple and austere core language, Canonical GF (CGF). CGF is well suited for implementing parsing and generation with grammars, as well as for proving properties of GF. This paper gives a concise description of both the core and the source language, the algorithm used in compiling GF to CGF, and some back-end optimizations on CGF.

1 Introduction

Grammar formalisms are *formal systems* used for *defining languages*. As formal systems, they can be reasoned about, so that their mathematical properties such as worst-case parsing complexity can be determined. To make rigorous mathematical reasoning feasible, a grammar formalism should be *austere*, i.e. have as few constructs as possible.

At the same time, grammar formalisms are also *programming languages* used for *writing grammars*. For this task, austerity is no more a virtue. If we think of general-purpose programming languages, such as C or Haskell, we can easily establish them as Turing-complete—thus they are, in a way, grammar formalisms in class 0 of the Chomsky hierarchy. But no-one would like to write programs in a Turing-complete language with as few constructs as possible, such as pure lambda calculus or Böhm's P" [3].

Practical programming languages have constructs that are *redundant*, i.e. not strictly necessary for writing programs. Typical redundant features are high-level control structures that can squeeze several lines of code to just one, and rich type systems, which help the programmer to keep the code consistent. Redundancies are also involved in various *abstractions* that human programmers want to do but which will get lost in the austere machine representation.

In the research on grammar formalisms, the formal system aspects have been more prominent than the programming language aspects. One reason is certainly that many mathematical properties are still an open question—for instance, what complexity class is appropriate for a grammar formalism. Working on austere formalisms is essential to keep track of these aspects. However, when actually writing grammars, redundant constructs that help grammarians are welcome. What is more, linguists have always strived after abstractions and generalizations. To support this, those grammar formalisms that are actually used by linguists typically provide mechanisms such as the following:

- *Reducible language extensions* For example, EBNF extends BNF by regular expressions over BNF items, which can always be eliminated but make grammar writing more compact.
- *Macros.* For example, the finite-state scripting language XFST [2] can be made to look almost like a functional language by the use of carefully chosen macros.

Such facilities can raise the abstraction level of grammars without sacrificing their mathematical properties, and they are straightforward to implement: only a syntax-based preprocessor is needed, rather than a real compiler that has to analyse the code semantically. In general-purpose programming languages, the compiler has to do more work. The following constructs are found in many modern languages, but seldom in grammar formalisms:

- *Type systems.* Grammar formalisms usually operate in an untyped universe of strings or atoms in the sense of Prolog or LISP.
- *Functions.* Grammar formalisms usually rely on macros, a kind of "poor man's functions". Replacing macros by proper functions contributes to type safety, but also permits the powerful technique of higher-order functions.
- *Module systems.* Grammar formalisms usually rely on inclusions of files, rather than on separately compiled modules.

These constructs are characteristic of GF, Grammatical Framework [19]. GF a grammar formalism first and foremost designed as a programming language. It is modelled after the typed functional languages ML and Haskell (types, functions, pattern matching), but has also been inspired by C++ (overloading, multiple inheritance). GF aims to be easy for ordinarily trained programmers (non-linguists) to use, but at the same time to give linguists a tool by which they can enjoy of powerful abstractions. GF supports the development of grammars in collaborative projects, where resource grammars written by linguists are used as libraries in application grammars written by programmers. Since the first implementation in 1998, hundreds of programmers have used GF to create linguistic resources, user applications, and of course toy programs testing the limits of GF. Grammars have been written for fragments of at least 100 languages, and 12 languages have extensive resource grammars [18].

The linguists' feel of GF is exemplified by a quote from Robin Cooper (personal communication): "using GF feels like having the power of a transformational grammar". The key to this power is functional programming. A syntactic

rule can use "transformations", i.e. functions that manipulate syntax trees, and even higher-order functions that take such functions as arguments. However, GF does not in the end have the transformational power: GF grammars reside in the class of polynomially parsable languages, as shown by Ljunglöf [10].

The "secret" of the controlled power of GF is the same thing that makes general-purpose programming languages work: compilation. "The GF grammar formalism" is actually two formalisms:

- Source GF, the rich language in which grammars are written.
- Canonical GF (CGF), the austere language to which grammars are compiled.

It would be hopeless for most humans to write grammars in CGF: it would feel like programming in machine code. At the same time, it would be hopeless to reason about source GF. There are too many language constructs to keep track of—a rough measure for this is that the BNF grammar used for parsing GF files has 247 productions. At the same time, the CGF format can be defined with less than 20 constructs.

The compiler that converts GF to CGF follows the compilation phases familiar from most modern compilers [1]:

1. Dependency analysis of the GF grammar to be compiled, determining what modules need compilation.
2. Lexing and parsing the GF code.
3. Type checking the parsed GF code with respect to the type system partly built within the code itself.
4. Simplifying the GF code so that it fits in a fragment of GF corresponding to CGF.
5. Generating the CGF code from the GF code.
6. Linking the grammar modules together into run-time grammar objects.
7. Optimizing the run-time grammar objects.

In this paper, we will first present the CGF formalism (Section 2) and give and outline of source GF (Section 3). The type checking phase is briefly discussed in Section 4, the simplifying phase in Section 5, the effect of simplification in grammar specialization in Section 6, generation of CGF in Section 7, and CGF optimizations in Section 8. A full presentation of source GF and its type system is given in [19]. The module system is introduced in [20].

2   Canonical GF

GF follows an architecture that divides a grammar into an *abstract syntax* and a *concrete syntax*. This division is commonplace in computer science, as a way of organizing compilers of programming languages. In linguistics, the same distinction was suggested by Curry [6] under the headings of *tectogrammatical* and *phenogrammatical* structure, respectively. It was implicitly followed by Montague [11] as observed by Dowty [9], but it has gained popularity only since the 1990's [17, 12, 19, 7, 15, 13].

A *run-time grammar* in GF is a grammar used for parsing and generation, as well as for type checking abstract syntax trees. It consists of one abstract syntax and one or more concrete syntaxes. The different concrete syntaxes typically model different languages, and the abstract syntax defines a common structure of those languages. GF grammars are thus *multilingual*.

To give a simplest possible example of a multilingual grammar, we first show a CGF grammar for constructing abstract greetings and linearizing them into two languages, English (*hello world*) and Italian (*ciao mondo*):

```
abstract Hello
  cat Greeting ; Addressee ;
  fun Hello : Addressee -> Greeting ;
  fun World : Addressee ;
concrete HelloEng
  lin Hello = [("hello",($0!0))] ;
  lin World = [("world")] ;
concrete HelloIta
  lin Hello = [("ciao",($0!0))] ;
  lin World = [("mondo")] ;
```

The abstract syntax has `cat` judgements defining the *categories* of the grammar, and `fun` judgementes definining the *functions* that construct *trees* of the categories. The functions can take zero or more arguments. For instance, `(Hello World)` is a tree formed by applying the one-argument function `Hello` to the zero-argument function `World`.

Each concrete syntax has `lin` judgements defining the *linearizations* of all functions. The linearization of a function with arguments may contain pointers to linearizations of the corresponding subtrees, denoted `$0, $1, $2`, etc. Linearization is thus a *compositional* operation, since it builds the linearization of a complex tree from the linearizations of its immediate subtrees.

The domain of linearization is a free algebra of syntax trees, inductively defined by an abstract syntax. The range of linearization is a universe whose elements are tokens, token lists, integers, and tuples. More formally, the universe T of tuples is built as follows:

- Tokens: `"foo"` : T.
- Token lists: $(t_1, \ldots, t_n)$: T for $t_1, \ldots, t_n : T$, $n \geq 0$.
- Integers: `0,1,2,...` : T.
- Tuples: $[\, t_1, \ldots, t_n \,]$: T for $t_1, \ldots, t_n : T$, $n \geq 0$.

Obviously, the universe T could be given a more discriminating type system, which would guarantee properties such as the impossibility to include integers in token lists. However, such restrictions are not needed in CGF, as long as we have a static type system for the GF source language and can thereby guarantee that the generated CGF expressions are meaningful.

Expressions for objects of the universe T also include *variables*, i.e. pointers to subtree linearizations, and *projections*, which select numbered components from tuples:

– Variables: $0, $1, $2,... : T.
– Projections: (t ! u) : T if $t : T$ and $u : T$.

Notice that projections are only needed because of the presence of variables. The values of variables are not known at compile time; but as soon as they get known, a well-formed projection can be brought into a form from which it can be eliminated:

$$([\ t_1,\ldots,t_n \ ] \ ! \ i \ ) \Longrightarrow t_i$$

In fact, this is the only computation rule needed in CGF, together with the rule of replacing variables with subtree linearizations. For this replacement, we maintain an array of these linearizations. The variable replacement rule is

$$\$i\{t_0,\ldots,t_n\} \Longrightarrow t_i$$

The top-level linearization t^* is defined compositionally as follows:

$$f \ t_1 \ldots t_n \Longrightarrow f^* \ \{t_1^*,\ldots,t_n^*\}$$

where f^* is the term defining the linearization of f in the grammar.

The tuple structure is exploited to express linguistic combinations that involve more than just string concatenation. Actually, tuples are just an austere representation of *feature structures*, in which all atomic features are encoded as integers. Tuples can moreover contain more than one string component, which gives a model of *discontinuous constituents*. The computational advantage of the integer representation of features is that tuples can be implemented as *arrays*, which can be stored compactly and have efficient lookup.

Here is an example showing how to deal with number agreement in English. A verb phrase (VP) is a tuple that contains both a singular and a plural form. A noun phrase (NP) is a tuple that contains both a string and a parameter, the latter indicating the grammatical number.

```
abstract Predic
  cat S ; NP ; VP ;
  fun Pred : NP -> VP -> S ;
  fun He : NP ;
  fun They : NP ;
  fun Talk : VP ;
  fun Walk : VP ;
concrete PredicEng of Predic
  lin Pred = [(($0!0),(($1!0)!($0!1)))] ;
  lin He = ["he",0] ;
  lin They = ["they",1] ;
  lin Talk = [["talks","talk"]] ;
  lin Walk = [["walks","walk"]] ;
```

The linearization of the tree (Pred They Walk) is computed as follows:

```
   [(($0!0),(($1!0)!($0!1)))] {["they",1], [["walks","walk"]]}
 = [((["they",1]!0),(((["walks","walk"]]!0)!(["they",1]!1)))]
 = [("they",(["walks","walk"]!1))]
 = [("they","walk")]
```

Again, it is crucial for the computation that the values held by the variables are of certain types. This is guaranteed when CGF is generated from GF source, where every category in the abstract syntax is assigned a *linearization type* in the concrete syntax. Here, for instance, it is assumed that every NP is a tuple holding a string and a parameter in range $\{0,1\}$, and that every VP is a tuple holding a tuple with two strings.

To reach the expressive power of GF, in the language-theoretic sense, nothing more is needed than CGF as presented above. For this formalism, we can define a linearization algorithm that is linear in the size of the tree, and a parsing algorithm that is polynomial in the size of the string. The parsing algorithm is obtained via a reduction to PMCFG [22], which indeed is another grammar formalism based on tuples. (This result, in [10], does not hold in general if dependent types appear in the abstract syntas; using them collapses GF to level-0 formalisms.)

In addition to reasoning, CGF is a good format for implementation. Interpreters for CGF have been built in C++, Haskell, Java, and Prolog [18], and make it possible to embed grammar components in those languages. An even more intimate embedding is achieved when CGF grammars are translated into a programming language; such translators have been written for C and JavaScript [18]. Moreover, CGF grammars can be translated (via context-free or finite-state approximations) to various formats used for defining language models in speech recognizers [4].

3 Source GF

Writing grammars in CGF manually is neither safe nor productive. The reasons are similar to the reasons why machine code is not a good medium for general-purpose programming:

- The lack of type checking makes it difficult to avoid errors when writing grammars.
- The selection of elements from tuples by position is error-prone.
- The use of integers to represent grammatical features charges human memory and makes it possible to confuse features of different types.
- Writing explicit tuples without any abstractions makes the grammars bulky.

The GF source formalism solves these problems by providing a higher-level language. A strict, static type system makes the language failure-safe, and abstraction mechanisms make grammars concise. The range of concrete syntax mappings is changed from the austere all-encompassing universe of tuples to systems of *records* and *tables*, whose layout can be defined by the programmer and is thus not given once and for all.

In the GF source language—from now on, just GF—linearizations are built from the following ingredients:

- Tokens: `"foo" : Str` (also used as one-element token lists).
- Empty token list: `[]`
- Concatenation of token lists: `s ++ t : Str` if `s,t : Str`.
- User-defined parameter types: `param Number = Sg | Pl`.
- Labelled records: $\{\ r_1 = t_1; \ldots; r_n = t_n\ \} : \{\ r_1 : T_1; \ldots; r_n : T_n\ \}$.
- Finite functions, i.e. tables: `table {Sg => "walks" ; Pl => "walk"} : Number => Str`.

Each category in an abstract syntax is given a linearization type (`lincat`) in the concrete syntas, and all linearization judgements are type-checked with respect to these types. Here is the second example from the previous section rewritten in GF.

```
abstract Predic = {
  cat S ; NP ; VP ;
  fun Pred : NP -> VP -> S ;
  fun He, They : NP ;
  fun NP ;
  fun Talk, Walk : VP ;
}
concrete PredicEng of Predic = {
  param Number = Sg | Pl ;
  lincat S  = {s : Str} ;
  lincat NP = {s : Str ; n : Number} ;
  lincat VP = {s : Number => Str} ;
  oper regVP : Str -> VP = \v -> {
    s = table {Sg => v + "s" ; _ => v}
    } ;
  lin Pred np vp = {s = np.s ++ vp.s ! np.n} ;
  lin He   = {s = "he" ; n = Sg} ;
  lin They = {s = "they" ; n = Pl} ;
  lin Talk = regVP "talk" ;
  lin Walk = regVP "walk" ;
}
```

The presence of variables means, as in CGF, that more expression forms are needed:

- Projections from records: $t.r$ returns the field labelled r from the record t.
- Selections from tables: $t\ !\ v$ returns the value assigned to v in the table `t`.
- Gluing of tokens: `"walk" + "s"` is computed to `"walks"`.

Note that the `case` expressions familiar from functional programming languages can be defined as syntactic sugar for table selections:

$$\text{case}\, t\, \text{of}\, \{\ldots\} \equiv \text{table}\, \{\ldots\}\, !\, t$$

This form of expression is convenient for programmers, and will also be used for presentation purposes below.

An important way in which GF is stronger than CGF is that variables are no longer only used for the arguments of linearization rules, but can appear anywhere in the concrete syntax:

- Auxiliary functions (`oper`) whose definitions bind variables.
- Local anonymous functions (lambda abstracts).
- Local definitions (`let` expressions).
- Pattern variables in table expressions used for case analysis.

It is the task of compilation from GF to CGF is to eliminate these "superfluous" variables. This can be done because their values are known at compile time. A technique based on *partial evaluation* is used. The usual evaluation rules of type theory are in this process applied to terms containing run-time variables.

Because of the way parsing algorithms work, the set of tokens must be known at compile-time (although it need not be finite). A token expression may thus not depend on run-time variables, which means that all instances of gluing tokens (`s + t`) are eliminated at compile time.

Let us conclude the presentation of GF with an example of parameter types and pattern matching. Parameter type definitions in GF are like algebraic datatype definitions in ML and Haskell, except that recursion is not allowed. They are thus not just enumerations of atomic features, but may introduce constructors that take arguments. An example is given by the following system, defining French agreement features as combinations of gender, number, and person:

```
param Gender = Masc | Fem ;
param Number = Sg | Pl ;
param Person = P1 | P2 | P3 ;
param Agr    = Ag Gender Number Person ;
```

The linearization of verb phrases depends on a parameter of type `Agr`. In actual pattern matching, separate branches are not necessarily needed for all values, but pattern variables can be passed to the right-hand side. This is what happens when verb phrases are formed from adjectival phrases by using the copula.

```
cat VP ; AP ;
lincat VP = {s : Agr => Str} ;
lincat AP = {s : Gender => Number => Str} ;
fun CompAP : AP -> VP ;
lin CompAP ap =
  {s = table {Ag g n p => copula n p ++ ap.s ! g ! n}} ;
oper copula : Number -> Person -> Str = ...
```

4   Type checking of GF expressions

The role of type checking in compilers is not just to verify the consistency of the code and report on errors. It also informs later compilation phases by adding

type annotations to expressions. The most important annotations in GF are the following:

– Annotate tables with their argument types, to enable eta expansion.
– Annotate projections with integers indicating the positions of the projected fields.

The information needed for these operations is available at type checking time and would require the duplication of much of type checking work if performed later.

A recent addition to the type system is *overloading* of functions, which has proved useful in the presentation of large grammar libraries. In type checking, overloaded functions are replaced by their instances determined by the *overloading resolution algorithm*. The way overloading works in GF is inspired by C++ [23]. In comparison with C++, the possibility of partial applications makes the problem more complex, whereas the absence of type casts simplifies the problem.

5   Simplification of GF expressions

Simplification is performed as partial evaluation, which has several ingredients: eta expansion (5.1); application of evaluation rules (5.2); generalized reductions needed to guarantee the subformula property (5.3); elimination of variables from complex parameter expressions (5.4).

5.1   Eta expansion

Eta expansion is step that converts a term to the data form required by the type of the term. For function types, this form is that of a lambda abstract. For record types, it is a record with explicit fields appearing in the type. For table types, it is a table that has explicitly enumerated cases for each value of the argument type. Thus the expansion rules are as follows:

– $t : A \to B \implies \backslash x \to (t\,x)$
– $t : \{\ r_1 : T_1; \ldots; r_n : T_n\ \} \implies \{\ r_1 = t.r_1; \ldots; r_n = t.r_n\ \}$
– $t : P \Rightarrow T \implies$ `table` $\{\ V_0 \Rightarrow t\,!\,V_0; \ldots; V_n \Rightarrow t\,!\,V_n\ \}$

Eta-expanded forms correspond directly to CGF expressions of the corresponding types.

5.2   Evaluation rules

The evaluation rules of GF are completely standard: beta conversion of lambda abstracts, projection from records, selection from tables by pattern matching, and concatenation of tokens.

– $(\backslash x \to b)\,a \implies b\{x := a\}$
– $\{\ldots; r = t; \ldots\}.r \implies t$

- `table {...; p ⇒ t;...} ! u ⟹ tγ` for the first p that matches u with γ
- `"foo" + "bar" ⟹ "foobar"`

Pattern matching is similar to ML and Haskell, scanning the patterns from left to right. Its result is a term $t\gamma$ where γ is a *substitution* by which the pattern p matches the value u. The matched term u may not in general contain variables. If variables occur in u, the selection must be postponed until the variables receive values; for run-time variables this means that the selection is passed to the generated CGF code.

5.3 The subformula property

Eta-expanded records and tables can be easily converted to CGF, but functions can be converted only if they appear as top-level linearization terms; no terms of a function type may occur inside those terms.

Now, a linearization term t in

$$\mathtt{fun}\, f\, :\, A_1 \to \cdots \to A_n \to A\, ;\, \mathtt{lin}\, f\, =\, t$$

is a function from the linearization types of $A_1, \ldots, A_n$ to the linearization type of A. These types are built from strings, features, records, and tables: no functions appear in these types. By a version of Curry-Howard isomorphism, the term t can be seen as a *proof* of A depending on the hypotheses $A_1, \ldots, A_n$. In this isomorphism, record types correspond to conjunctions and parameter types to disjunctions. No function types occur in the hypotheses or the conclusion. That no terms of a function type need occur in t is an instance of the *subformula property* of intuitionistic propositional calculus in proof theory.

To prove the subformula property, it is not enough to use the ordinary evaluation rules. The problem is that the elimination of a function can be blocked by a variable. A case in point is a term of the form

$$(\mathtt{case}\, x\, \mathtt{of}\{A \Rightarrow f\, ;\, B \Rightarrow g\})\, c$$

where the function application cannot be performed because of x. This term is isomorphic to a proof in which disjunction elimination is followed by a modus ponens. For this constellation, Prawitz [16] introduced a generalized reduction rule:

$$
\cfrac{A \vee B \quad \cfrac{\overset{(A)}{C \to D} \quad \overset{(B)}{C \to D}}{C \to D} \quad C}{D}
\quad \Longrightarrow \quad
\cfrac{A \vee B \quad \cfrac{\overset{(A)}{C \to D} \quad C}{D} \quad \cfrac{\overset{(B)}{C \to D} \quad C}{D}}{D}
$$

Now the modus ponens has moved closer to the introductions of the implication, and can, as Prawitz proved, eventually be reduced away. The corresponding term transformation in the GF compiler is

$$(\mathtt{case}\, x\, \mathtt{of}\{A \Rightarrow f\, ;\, B \Rightarrow g\})\, c \implies \mathtt{case}\, x\, \mathtt{of}\{A \Rightarrow f\, c\, ;\, B \Rightarrow g\, c\}$$

5.4   Parameter constructors with variables in arguments

Parameter type definitions can introduce constructors that takes arguments, e.g. the constructor type `Ag` of French agreement features in Section 3. Now, an argument of `Ag` can be a run-time variable, as in

```
Ag Fem np.n P3
```

This expression has no translation in CGF. But we can first translate it to a case expression,

```
case np.n of {
  Sg => Ag Fem Sg P3 ;
  Pl => Ag Fem Pl P3
  }
```

which can then be translated to CGF by applying the compilation schemes of Section 7. This transformation has of course has be performed recursively, since there can be several occurrences of run-time variables in a constructor application.

6   Compilation and grammar specialization

A typical GF application is built on top of a *domain grammar*, whose abstract syntax is a type-theoretical model of a domain semantics. In the beginning, all GF grammars were such domain grammars implemented by writing concrete syntaxes from scratch. But gradually the evolution of the GF compiler permitted writing large-scale *resource grammars* and using them as libraries. The use of resource grammars in writing domain grammars is also known as *grammar specialization*.

As resource grammars are large and complex, they have high demands on both time and space. Properly implemented grammar specialization should eliminate all run-time penalty potentially caused by resource grammars. To show how this happens in GF, let us trace through the compilation of a simple domain grammar rule implemented using the GF resource grammar library [18]. The purpose of the application is to cover voice commands such as *I want to hear this song*. This is an example of how GF was used in the TALK project for building dialogue systems [14].

The abstract syntax covering the voice command is

```
cat Command ; Kind ;
fun want_hear_this : Kind -> Command ;
fun Song, Record, Singer : Kind ;
```

The concrete syntax assigns resource grammar categories as linearization types to the domain categories. Thus commands are utterances (`Utt`), kinds are common nouns (`CN`):

```
lincat Command = Utt ; Kind = CN ;
```

The linearization of the function want_hear_this is built by using constructor functions from the resource API. These constructors are either syntactic, having the form mk*C* for a function whose value category is *C*, or lexical, having the form *word*_C for a lexical unit of category *C*.

```
lin want_hear_this x = mkUtt (mkCl i_Pron
  (mkVP want_VV (mkVP hear_V2 (mkNP this_Quant x))))
```

The syntactic constructors are overloaded, for instance mkVP is here used for both V2 (NP-complement verbs) and VV (VP-complement verbs).

   In the resource grammar, a clause like our example has forms for different tenses and polarities and, e.g. in the case of German, also for different word orders. So the German variation includes 48 sentence forms, in the range

```
Pres Simul Pos Main: ich will dieses x hören
Pres Simul Pos Inv:  will ich dieses x hören
...
Cond Anter Neg Sub:  ich dieses x nicht würde hören wollen haben
```

Moreover, the form of dieses ("this") varies according to the gender of x: *diesen Sänger, dieses Lied, diese Platte*. All this variation gives initially 3*48 = 144 forms in the expanded tables. However, the linearization type Utt of the top-level category Command is a plain string, with no variation. The constructor mkUtt without explicit tense and polarity uses the values present, positive, and main clause. The category Kind is linearized to CN, which has a 3-valued gender parameter, so that the variation *diesen, dieses, diese* cannot be eliminated. So we have 3 forms that remain in the generated CGF grammar:

```
ich will diesen x hören
ich will dieses x hören
ich will diese  x hören
```

Further optimizations on the CGF code compactify this grammar by e.g. removing the repetitions of word strings (Section 8).

7   Translating GF to CGF

Once the GF grammar has been type-annotated and partial-evaluated, translation to CGF can be performed by compositional compilation schemes:

- $\text{lin } f \, \$0...\$n = t \implies \text{lin } f = t$
- $\{r_1 = t_1; \ldots; r_n = t_n\} \implies [t_1, \ldots, t_n]$
- $\text{table}\{V_1 \Rightarrow t_1; \ldots; V_n \Rightarrow t_n\} \implies [t_1, \ldots, t_n]$
- $s \mathbin{++} t \implies (s, t)$
- $[] \implies (s, t)$
- $t.r[i] \implies (t \mathbin{!} i)$

  – $t!u \implies (t!u)$
  – $C\,v_1 \dots v_n \implies \#(C\,v_1 \dots v_n)$

The expression $t.r[i]$ is a projection annotated by the position i of the label r. The expression $\#(v)$ denotes the integer value of the parameter value v.

The other forms of judgement—`param` and `lincat`—are not needed in run-time grammars and are hence omitted. However, when library grammars are separately compiled into CGF, these judgements are needed for type checking modules that use them.

Abstract syntax must be present in CGF, but it needs not be translated, apart from eliminating some syntactic sugar provided by the GF source language [19].

8   Back-end optimizations on CGF

Even though partial evaluation eliminates unnecessary rules from grammars (Section 6), the code bloat resulting from eta expansion can be significant. While eta expansion is indispensable for the compilation to work, some of its drawbacks can be relieved by back-end optimizations on CGF. This section gives a summary of two such optimizations, each of which uses a new expression form and thereby makes CGF less austere.

8.1   Common subexpression elimination

One and the same CGF term can appear several times in a grammar. Such terms can be captured by a standard technique of *common subexpression elimination*, which replaces the subterms with new constants and adds the definitions of those constants into the grammar. These expressions can contain run-time variables, and they are computed by simple syntactic replacement. The computation can be performed off-line in the whole grammar, but usually it is better to keep them in the run-time grammar and look them up at need.

Automatic subexpression elimination is often more powerful than hand-devised code sharing in the source code, because it catches all subexpressions that are used at least twice in the code. It is moreover iterated so that it takes into account the subexpressions in the definitions of global constants. The shrinkage of code size is typically by an order of magnitude.

8.2   Prefix-suffix tables

Suppose we have in a grammar the rule

```
Walk = [["walk", "walks", "walked", "walking"]]
```

The *prefix-suffix table* representation divides an array of words to the longest common prefix and an array of suffixes.

```
Walk = [[("walk" + ["","s", "ed", "ing"])]]
```

The power of this representation comes from the fact that suffix arrays tend to be repeated in a language, and can therefore be collected by common subexpression elimination. After this, a grammar may look as follows:

```
__a18 = ["","s", "ed", "ing"]
Destroy = [[("destroy" + __a18)]]
Talk = [[("talk" + __a18)]]
Walk = [[("walk" + __a18)]]
```

Thus this optimization in fact identifies a set of concatenative inflection paradigms of the language.

9   Implementation

The GF grammar compiler is a central component of the GF grammar development system, i.e. the program called `gf` and available from [18] as open-source software. The austere CGF format described in this paper is a second-generation target language for GF. At the moment, the GF system still uses a richer target language called GFC. GFC is a compromise between run-time simplicity and the concerns of separate compilation. The current GF system does produce CGF as a back-end format (under the name GFCC), but separate compilation and parser generation still depend on GFC.

10   Related work

Static type checking and library specialization are not very common in grammar formalisms. HPSG has type checking of typed feature structures [5], and Regulus [21] uses a technique called *explanation-based learning* for specializing large resource grammars to domain-specific run-time grammars. Regulus moreover compiles high-level unification grammars into lower-level context-free grammars by expanding rules depending on finite feature sets into sets of context-free rules.

In the new wave of grammar formalisms inspired by Curry [6], ACG has an implementation where generation is treated as term rewriting and parsing as higher-order linear matching [7]. These techniques are direct and elegant applications of the metatheory of ACG, but they do not give narrow complexity bounds. However, the equivalence results for fragments of ACG and classes such as context-free and mildly context-sensitive grammars [8] could serve as basis of efficient implementations via compilation.

References

1. A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools. Second Edition.* Addison-Wesley, 2006.
2. K. Beesley and L. Karttunen. *Finite State Morphology.* CSLI Publications, 2003.

3. C. Böhm. On a family of Turing machines and the related programming language. *ICC Bulletine*, 3:185–194, 1964.

4. B. Bringert. Speech Recognition Grammar Compilation in Grammatical Framework. In *SPEECHGRAM 2007: ACL Workshop on Grammar-Based Approaches to Spoken Language Processing, June 29, 2007, Prague*, 2007.

5. A. Copestake and D. Flickinger. An open-source grammar development environment and broad-coverage English grammar using HPSG. *Proceedings of the Second conference on Language Resources and Evaluation (LREC-2000)*, 2000.

6. H. B. Curry. Some logical aspects of grammatical structure. In Roman Jakobson, editor, *Structure of Language and its Mathematical Aspects: Proceedings of the Twelfth Symposium in Applied Mathematics*, pages 56–68. American Mathematical Society, 1963.

7. Ph. de Groote. Towards Abstract Categorial Grammars. In *Association for Computational Linguistics, 39th Annual Meeting and 10th Conference of the European Chapter, Toulouse, France*, pages 148–155, 2001.

8. Ph. de Groote. Tree-Adjoining Grammars as Abstract Categorial Grammars. In *TAG+6, Proceedings of the sixth International Workshop on Tree Adjoining Grammars and Related Frameworks*, pages 145–150. Università di Venezia, 2002.

9. D Dowty. *Word Meaning and Montague Grammar*. D. Reidel, Dordrecht, 1979.

10. P. Ljunglöf. *The Expressivity and Complexity of Grammatical Framework*. PhD thesis, Dept. of Computing Science, Chalmers University of Technology and Gothenburg University, 2004.

11. R. Montague. *Formal Philosophy*. Yale University Press, New Haven, 1974. Collected papers edited by Richmond Thomason.

12. R. Muskens. *Meaning and Partiality*. PhD thesis, University of Amsterdam, 1989.

13. R. Muskens. Lambda Grammars and the Syntax-Semantics Interface. In R. van Rooy and M. Stokhof, editors, *Proceedings of the Thirteenth Amsterdam Colloquium*, pages 150–155, Amsterdam, 2001.

14. N. Perera and A. Ranta. Dialogue System Localization with the GF Resource Grammar Library. In *SPEECHGRAM 2007: ACL Workshop on Grammar-Based Approaches to Spoken Language Processing, June 29, 2007, Prague*, 2007.

15. C. Pollard. Higher-Order Categorial Grammar. In M. Moortgat, editor, *Proceedings of the Conference on Categorial Grammars (CG2004), Montpellier, France*, pages 340–361, 2004.

16. D. Prawitz. *Natural Deduction*. Almqvist & Wiksell, Stockholm, 1965.

17. A. Ranta. *Type Theoretical Grammar*. Oxford University Press, 1994.

18. A. Ranta. Grammatical Framework Homepage, 2002. www.cs.chalmers.se/~aarne/GF/.

19. A. Ranta. Grammatical Framework: A Type-theoretical Grammar Formalism. *The Journal of Functional Programming*, 14(2):145–189, 2004.

20. A. Ranta. Modular Grammar Engineering in GF. *Research on Language and Computation*, 2007. To appear.

21. M. Rayner, B. A. Hockey, and P. Bouillon. *Putting Linguistics into Speech Recognition: The Regulus Grammar Compiler*. CSLI Publications, 2006.

22. H. Seki, T. Matsumura, M. Fujii, and T. Kasami. On multiple context-free grammars. *Theoretical Computer Science*, 88:191–229, 1991.

23. B. Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley, 1998.

Type-theoretic Extensions of Abstract Categorial Grammars

Philippe de Groote[1] and Sarah Maarek[2]

[1] LORIA & INRIA-Lorraine
Philippe.de.Groote@loria.fr

[2] LORIA & Université Nancy 2
Sarah.Maarek@loria.fr

1   Introduction

Abstract Categorial Grammars (ACG), in their original definition [3], are based on the linear λ-calculus. This choice derives from the traditional categorial grammars, which are based on resource sensitive logics [8].

From a language-theoretic standpoint, this linearity constraint does not result in a weak expressive power of the formalism [4, 5]. In particular, the string languages generated by the second-order ACGs (whose parsing is known to be polynomial [12]) corresponds to the class of mildly context sensitive languages. From a more practical point of view, however, it would be interesting to increase the intentional expressive power of the formalism by providing high level constructs. For instance, one would like to provide the ACGs with feature structures, as it is the case in most current grammatical formalisms.

In [3], a possible way of extending the ACGs is proposed. It consists of enriching the type system of the formalism with new type constructors. The present paper, which elaborates on this proposal, is organized as follows:

- In the next section, we remind the reader of the definition of an ACG. Then, we explain why the ACG architecture is well-suited for type-theoretic extensions.
- In Section 3, we briefly review and motivate possible extensions based on the following type constructors: non-linear functional types, cartesian product, disjoint union, unit type, and dependent product.
- In Section 4, we define formally the type system underlying the extensions proposed in Section 3.
- Finally, we illustrate the resulting system by providing a toy example.

2   The ACG architecture

An Abstract Categorial Grammar consists of two signatures together with a morphism that allows the types and the terms built on the first signature to be interpreted as types and terms built on the second signature. This morphism

(which is called the *lexicon* of the grammar) is required to commute with the typing relation, which ensures that well-typed terms are interpreted as well-typed terms.

More formally, let $\mathscr{T}(A)$ denotes the set of linear functional types[3] built from the set of atomic types A, and define a higher-order linear signature Σ to be a triple $\langle A, C, \tau \rangle$, where A is a finite set of atomic types, C is a finite set of constants, and $\tau : C \rightarrow \mathscr{T}(A)$ is a function that assigns a linear functional type to each constant. Given such a signature Σ, define $\Lambda(\Sigma)$ to be the set of (well-typed) linear λ-terms built on Σ. Then, given two higher-order linear signatures $\Sigma_1 = \langle A_1, C_1, \tau_1 \rangle$ and $\Sigma_2 = \langle A_2, C_2, \tau_2 \rangle$, define a lexicon from Σ_1 to Σ_2 to be a pair $\mathscr{L} = \langle \eta, \theta \rangle$ where $\eta : A_1 \rightarrow \mathscr{T}(A_2)$ and $\theta : C_1 \rightarrow \Lambda(\Sigma_2)$ are such that:

$$\vdash_{\Sigma_2} \theta(c) : \eta(\tau_1(c)) \text{ for all } c \in C_1$$

This condition ensures that the homomorphic extensions of η and θ are such that well-typed terms are interpreted by well-typed terms.

Using the above definitions,, an abstract categorial grammar is defined to be a quadruple $\mathscr{G} = \langle \Sigma_1, \Sigma_2, \mathscr{L}, s \rangle$ where:

1. Σ_1 and Σ_2 are two higher-order linear signatures;
2. $\mathscr{L}$ is a lexicon form Σ_1 to Σ_2;
3. s is an atomic type of Σ_1 called the *distinguished type* of the grammar.

From a methodological point of view, the first signature (which is called the *abstract vocabulary*) is used to specify the abstract parse structures of the grammar, while the second signature (which is called the *object vocabulary*) is used to express the surface forms of the grammar.[4] More precisely, an Abstract Categorial Grammar generates two languages: an abstract language, which corresponds to Curry's tectogrammatics, and an object language, which corresponds to Curry's phenogrammatics [2]. From a formal point of view, the abstract language is simply defined to be the set of closed terms, built on the abstract vocabulary, that are of the given distinguished type:

$$\mathcal{A}(\mathscr{G}) = \{t \in \Lambda(\Sigma_1) \mid \vdash_{\Sigma_1} t : s\}$$

Then, the object language is defined to be the image of the abstract language under the lexicon:

$$\mathcal{O}(\mathscr{G}) = \{t \in \Lambda(\Sigma_2) \mid \exists u \in \mathcal{A}(\mathscr{G}).\ t = \theta(u)\}$$

It is important to note that this architecture is in fact independent of the underlying type system. This provides us with a systematic way of extending the expressive power of the ACG framework, simply by enriching the underlying type theory.

---

[3] We use the connective $\multimap$ —i.e., the linear implication symbol— to denote the linear functional type constructor.

[4] It may also be the case that the object vocabulary is used to express the logical forms of the grammar. In the present setting, the notions of syntax, semantics, parse structure, surface form, logical form, ... are purely methodological. These notions do not belong to the theory of ACGs.

3   The need for type-theoretic extensions

In [3], we suggest that all the connectives of linear logic (seen, through the Curry-Howard isomorphism, as type constructors) are natural candidates for extending the ACG typing system. In this paper, we propose extensions based on the following type constructors:

- Non-linear functional type. This corresponds, by the Curry-Howard isomorphism, to intuitionistic implication. It is not a primitive connective of linear logic, but may be defined by means of the linear implication and the "of course" modality.
- Cartesian product, disjoint union, and unit type. These corresponds to the additive conjunction and disjunction, and the multiplicative unit.
- Dependent product. This type construct does not correspond to any linear logic primitive. It comes from Matin-Löf type-theory [7] and has been used in a linear logic context by Pfenning and Cervesato [1].

This choice is not a matter of doctrine. It rather results from experiments in writing toy grammars.

Non-linearity.   Categorial grammars are based on linear type-theories because grammatical composition has been identified as a resource-sensitive process. This does not mean, however, that grammatical composition is always linear. It rather means that categorial type logics should allow for linearity.

Non linear combinators are quite usual in semantics. For instance, the lexical semantics of an intersective adjective such as *red* is given by the following non-linear λ-term:[5]

$$(1) \qquad\qquad \lambda^\circ P. \lambda x. (P\, x) \wedge (\mathbf{red}\, x) : (\iota \to o) \multimap (\iota \to o)$$

Another typical use of non-linearity is provided, on the syntactic side, by feature agreement. This is well exemplified in Ranta's GF [11].

In the ACG framework, both syntax and semantics are modeled using the same primitives. As a consequence, a λ-term such as (1) must be allowed as a possible lexical entry. Consequently, there is a crucial need for non-linear λ-abstraction.

From a technical point of view, mixing linear and non-linear functional types results in a typing system with two kinds of contexts: linear contexts, and non-linear contexts. Such a system will be defined in Section 4.

Cartesian product, disjoint union, and unit.   As we already mentioned in the introduction of this paper, we feel a need for providing ACGs with feature structures. These may be defined using records, variants, and enumerated types

---

[5] We write $\lambda^\circ x. t$ for a linear λ-abstraction, and $\lambda x. t$ for a non-linear (i.e., usual) λ-abstraction.

(which may appear to be particular cases of variants). For instance, the following feature matrix:

$$\begin{bmatrix} \text{inflection} = \begin{bmatrix} \text{gender} & = \text{masc} \\ \text{number} = \text{sing} \end{bmatrix} \\ \text{function} \ \ = \text{object}\,(\text{indirect}\,(\text{a\_Prep})) \end{bmatrix}$$

may be expressed as a well-typed term of the following signature:[6]

$$\begin{aligned} Gender &= \{\text{masc} \mid \text{fem}\} : \text{type} \\ Number &= \{\text{sing} \mid \text{plur}\} : \text{type} \\ Inflection &= [\text{gender} : Gender;\ \text{number} : Number] : \text{type} \\ Preposition &= \{\text{a\_Prep} \mid \text{de\_Prep}\} : \text{type} \\ Direction &= \{\text{direct} \mid \text{indirect}\ \mathtt{of}\ Preposition\} : \text{type} \\ Function &= \{\text{subject} \mid \text{object}\ \mathtt{of}\ Direction\} : \text{type} \\ Features &= [\text{inflection} : Inflection;\ \text{function} : Function] : \text{type} \end{aligned}$$

From a type theoretic standpoint, records, variants, and enumerated types may be defined using cartesian products, disjoint unions, and unit types.

Dependent product   Dependent product is a quite powerful type construction. It allows ones to specify types that depend upon terms. This is exactly what is needed if one wants to define generic syntactic categories (for instance, NP for *noun phrase*) that can be instantiated according to the value of some feature (for instance, $(NP\,f)$ for *feminine noun phrase*, $(NP\,m)$ for *masculine noun phrase*, etc.)

Dependent products are not that much popular in the type logical grammar community, even if some weak form of dependent function is mentioned by both Morrill [9] and Moortgat [8]. There is, however, quite an exception to this matter of fact, namely, Ranta's type-theoretical grammars [10]. If Ranta's Grammatical Framework is definitively an achievement that speaks for itself, it also speaks for the use of dependent products.

4   The extended typing system

In the presence of dependent products, the well-formedness of types may depend upon the well-typedness of terms. The usual solution to this is to consider a third level of expressions: the kinds. These are to the types what the types are to the terms. Consequently, the extended typing system we propose relies on four sorts of typing judgements that are defined by mutual induction:

– The judgement that a signature is well-formed.
– The judgement that a kind is well-formed.

---

[6] We use square brackets for records and curly brackets for variants. We hope that the reader will find this syntax self-explanatory. If this is not the case, we refer him/her to Section 4.

- The judgement that a type is well-kinded.
- The judgement that a term is well-typed.

These judgements manipulate expressions that obey the following raw syntax.

Signatures

$$\Sigma ::= ()\qquad\qquad\qquad\qquad (Empty\ signature)$$

$\Sigma ::= ()$	(*Empty signature*)
$\mid\ \Sigma; a : \mathsf{K}$	(*Atomic type declaration*)
$\mid\ \Sigma; a = \mathrm{R} : \mathsf{type}$	(*Record type declaration*)
$\mid\ \Sigma; a = \mathrm{V} : \mathsf{type}$	(*Variant type declaration*)
$\mid\ \Sigma; c : \tau$	(*Constant declaration*)

Kinds

$\mathsf{K} ::= \mathsf{type}$	(*Kind of types*)
$\mid\ (\tau)\mathsf{K}$	(*Kind of dependent types*)

Types

$\tau ::= a$	(*Atomic type*)
$\mid\ (\lambda x.\, \tau)$	(*Type abstraction*)
$\mid\ (\tau\, t)$	(*Type application*)
$\mid\ (\tau_1 \multimap \tau_2)$	(*Linear functional type*)
$\mid\ (\Pi x : \tau_1)\, \tau_2)$	(*Dependent product*)

Record types

$\mathrm{R} ::= [l_1 : \tau_1; \ldots; l_n : \tau_n]$	(*Record type*)

Variant types

$\mathrm{V} ::= \{c_1\ \mathsf{of}\ \tau_1 \mid \ldots \mid c_n\ \mathsf{of}\ \tau_n\}$	(*Variant type*)

Terms

$t ::= c$	(*Constant*)
$\mid\ x$	(*Variable*)
$\mid\ (\lambda^\circ x.\, t)$	(*Linear abstraction*)
$\mid\ (\lambda x.\, t)$	(*Non-linear abstraction*)
$\mid\ (t_1\, t_2)$	(*Application*)
$\mid\ [l_1 = t_1; \ldots; l_n = t_n]$	(*Record*)
$\mid\ t.l$	(*Selection*)
$\mid\ \{(c_1\, x_1) \rightarrow t_1 \mid \ldots \mid (c_n\, x_n) \rightarrow u_n\}$	(*Case analysis*)

In this grammar, a, c, x, and l (possibly with subscripts) range over type constants, λ-term constants,[7] λ-variables, and record labels, respectively. In a variant type the "$\mathsf{of}\ \tau_i$" part is optional. In which case, the corresponding variant

---

[7] Variant constructors are considered as a special case of constants.

constructor c_i amounts to a constant. This allows enumerated types to be seen as particular cases of variants.

Given a (raw) signature Σ, we define three partial functions. The first one, $kind_\Sigma$, takes a type constant as an argument and possibly yields a kind as a result. It is inductively defined as follows:

$$kind_{()}(a) \text{ is undefined}$$

$$kind_{\Sigma;\, a_1:\mathsf{K}}(a) = \begin{cases} \mathsf{K} & \text{if } a = a_1 \\ kind_\Sigma(a) & \text{otherwise} \end{cases}$$

$$kind_{\Sigma;\, a_1=\mathrm{R}:\mathsf{type}}(a) = \begin{cases} \mathsf{type} & \text{if } a = a_1 \\ kind_\Sigma(a) & \text{otherwise} \end{cases}$$

$$kind_{\Sigma;\, a_1=\mathrm{V}:\mathsf{type}}(a) = \begin{cases} \mathsf{type} & \text{if } a = a_1 \\ kind_\Sigma(a) & \text{otherwise} \end{cases}$$

$$kind_{\Sigma;\, c:\tau}(a) = kind_\Sigma(a)$$

Similarly, $type_\Sigma$ assigns types to λ-term constants:

$$type_{()}(c) \text{ is undefined}$$

$$type_{\Sigma;\, a_1:\mathsf{K}}(c) = type_\Sigma(c)$$

$$type_{\Sigma;\, a_1=\mathrm{R}:\mathsf{type}}(c) = type_\Sigma(c)$$

$$type_{\Sigma;\, a_1=\mathrm{V}:\mathsf{type}}(c) = type_\Sigma(c)$$

$$type_{\Sigma;\, c_1:\tau}(c) = \begin{cases} \tau & \text{if } c = c_1 \\ type_\Sigma(c) & \text{otherwise} \end{cases}$$

Finally, $binding_\Sigma$, takes a type constant as an argument and possibly yields a record or variant type:

$$binding_{()}(a) \text{ is undefined}$$

$$binding_{\Sigma;\, a_1:\mathsf{K}}(a) = \begin{cases} \text{undefined} & \text{if } a = a_1 \\ binding_\Sigma(a) & \text{otherwise} \end{cases}$$

$$binding_{\Sigma;\, a_1=\mathrm{R}:\mathsf{type}}(a) = \begin{cases} \mathrm{R} & \text{if } a = a_1 \\ binding_\Sigma(a) & \text{otherwise} \end{cases}$$

$$binding_{\Sigma;\, a_1=\mathrm{V}:\mathsf{type}}(a) = \begin{cases} \mathrm{V} & \text{if } a = a_1 \\ binding_\Sigma(a) & \text{otherwise} \end{cases}$$

$$binding_{\Sigma;\, c:\tau}(a) = binding_\Sigma(a)$$

As we already said, the rules of the typing system involve four sorts of judgements. They are of the following forms:

$\mathsf{sig}\,(\Sigma)$ $(\Sigma$ *is a well-formed signature*$)$

$\vdash_\Sigma \mathsf{K} : \mathsf{kind}$ $(Given\ the\ signature\ \Sigma,\ \mathsf{K}\ is\ a\ well\mbox{-}formed\ kind)$

$\Gamma \vdash_\Sigma \alpha : \mathsf{K}$ $(Given\ the\ signature\ \Sigma,\ \alpha\ is\ a\ type\ of\ kind\ \mathsf{K}\ according\ to$
                                            $the\ non\mbox{-}linear\ typing\ context\ \Gamma)$

$\Gamma; \Delta \vdash_\Sigma t : \alpha$ $(Given\ the\ signature\ \Sigma,\ t\ is\ a\ term\ of\ type\ \alpha\ according$
                                            $to\ the\ non\mbox{-}linear\ typing\ context\ \Gamma\ and\ the\ linear\ typing$
                                            $context\ \Delta)$

We are now in a position of giving the very rules of the typing system.

Well-formed signatures

$$\mathsf{sig}\,(\,)$$

$$\frac{\mathsf{sig}\,(\Sigma) \quad \vdash_\Sigma \mathsf{K} : \mathsf{kind}}{\mathsf{sig}\,(\Sigma; a : \mathsf{K})}$$

$$\frac{\mathsf{sig}\,(\Sigma) \quad \vdash_\Sigma \alpha_1 : \mathsf{type} \quad \cdots \quad \vdash_\Sigma \alpha_n : \mathsf{type}}{\mathsf{sig}\,(\Sigma; a = [l_1 : \alpha_1; \ldots; l_n : \alpha_n] : \mathsf{type})}$$

$$\frac{\mathsf{sig}\,(\Sigma) \quad \vdash_\Sigma \alpha_1 : \mathsf{type} \quad \cdots \quad \vdash_\Sigma \alpha_n : \mathsf{type}}{\mathsf{sig}\,(\Sigma; a = \{c_1\ \mathtt{of}\ \alpha_1 | \ldots | c_n\ \mathtt{of}\ \alpha_n\} : \mathsf{type})}$$

$$\frac{\mathsf{sig}\,(\Sigma) \quad \vdash_\Sigma \alpha : \mathsf{type}}{\mathsf{sig}\,(\Sigma; c : \alpha)}$$

In the above rules, all the introduced symbols $(a, l_1, \ldots, l_n, c_1, \ldots, c_n, c)$ must be fresh with respect to Σ.

Well-formed kinds

$$\vdash_\Sigma \mathsf{type} : \mathsf{kind}$$

$$\frac{\vdash_\Sigma \alpha : \mathsf{type} \quad \vdash_\Sigma \mathsf{K} : \mathsf{kind}}{\vdash_\Sigma (\alpha)\,\mathsf{K} : \mathsf{kind}}$$

Well-kinded types

$$\vdash_\Sigma a : kind_\Sigma(a) \quad (\text{type const.})$$

$$\frac{\vdash_\Sigma \; \alpha : \mathsf{type} \quad \Gamma \; \vdash_\Sigma \; \beta : \mathsf{K}}{\Gamma, x : \alpha \; \vdash_\Sigma \; \beta : \mathsf{K}} \quad \text{(type weak.)}$$

$$\frac{\Gamma, x : \alpha \; \vdash_\Sigma \; \beta : \mathsf{K}}{\Gamma \; \vdash_\Sigma \; \lambda x.\, \beta : (\alpha)\,\mathsf{K}} \quad \text{(type abs.)}$$

$$\frac{\Gamma \; \vdash_\Sigma \; \alpha : (\beta)\,\mathsf{K} \quad \Gamma; \; \vdash_\Sigma \; t : \beta}{\Gamma \; \vdash_\Sigma \; \alpha\, t : \mathsf{K}} \quad \text{(type app.)}$$

$$\frac{\Gamma \; \vdash_\Sigma \; \alpha : \mathsf{type} \quad \Gamma \; \vdash_\Sigma \; \beta : \mathsf{type}}{\Gamma \; \vdash_\Sigma \; \alpha \multimap \beta : \mathsf{type}} \quad \text{(lin. fun.)}$$

$$\frac{\Gamma \; \vdash_\Sigma \; \alpha : \mathsf{type} \quad \Gamma, x : \alpha \vdash_\Sigma \; \beta : \mathsf{type}}{\Gamma \; \vdash_\Sigma \; (\Pi x : \alpha)\, \beta : \mathsf{type}} \quad \text{(dep. prod.)}$$

In Rule (type weak.), x must be fresh with respect to Γ.

Well-typed terms

$$; \; \vdash_\Sigma \; c : type_\Sigma(c) \quad \text{(const.)}$$

$$\frac{\Gamma \; \vdash_\Sigma \; \alpha : \mathsf{type}}{\Gamma; \, x : \alpha \vdash_\Sigma \; x : \alpha} \quad \text{(lin. var.)}$$

$$\frac{\Gamma \; \vdash_\Sigma \; \alpha : \mathsf{type}}{\Gamma, x : \alpha; \; \vdash_\Sigma \; x : \alpha} \quad \text{(var.)}$$

$$\frac{\Gamma \; \vdash_\Sigma \; \alpha : \mathsf{type} \quad \Gamma; \Delta \vdash_\Sigma \; t : \beta}{\Gamma, x : \alpha; \Delta \vdash_\Sigma \; t : \beta} \quad \text{(weak.)}$$

$$\frac{\Gamma; \Delta_1, x : \alpha, y : \beta, \Delta_2 \vdash_\Sigma \; t : \gamma}{\Gamma; \Delta_1, y : \beta, x : \alpha, \Delta_2 \vdash_\Sigma \; t : \gamma} \quad \text{(perm.)}$$

$$\frac{\Gamma; \Delta, x : \alpha \vdash_\Sigma \; t : \beta}{\Gamma; \Delta \vdash_\Sigma \; \lambda^\circ x.\, t : \alpha \multimap \beta} \quad \text{(lin. abs.)}$$

$$\frac{\Gamma; \Delta_1 \vdash_\Sigma t : \alpha \multimap \beta \quad \Gamma; \Delta_2 \vdash_\Sigma u : \alpha}{\Gamma; \Delta_1, \Delta_2 \vdash_\Sigma t\, u : \beta} \quad \text{(lin. app.)}$$

$$\frac{\Gamma, x : \alpha; \Delta \vdash_\Sigma t : \beta}{\Gamma; \Delta \vdash_\Sigma \lambda x.\, t : (\Pi x : \alpha)\, \beta} \quad \text{(abs.)}$$

$$\frac{\Gamma; \Delta \vdash_\Sigma t : (\Pi x : \alpha)\beta \quad \Gamma; \vdash_\Sigma u : \alpha}{\Gamma; \Delta \vdash_\Sigma t\, u : \beta[x{:=}u]} \quad \text{(app.)}$$

$$\frac{binding_\Sigma(a) = [l_1 : \alpha_1; \ldots; l_n : \alpha_n] \quad \Gamma; \Delta \vdash_\Sigma t_i : \alpha_i \;\; (1 \le i \le n)}{\Gamma; \Delta \vdash_\Sigma [l_1 = t_1; \ldots; l_n = t_n] : a} \quad \text{(rec.)}$$

$$\frac{binding_\Sigma(a) = [l_1 : \alpha_1; \ldots; l_n : \alpha_n] \quad \Gamma; \Delta \vdash_\Sigma t : a}{\Gamma; \Delta \vdash_\Sigma t.l_i : \alpha_i} \quad \text{(sel.)}$$

$$\frac{binding_\Sigma(a) = \{c_1 \text{ of } \alpha_1 | \ldots | c_n \text{ of } \alpha_n\}}{; \vdash_\Sigma c_i : \alpha_i \multimap a} \quad \text{(inj.)}$$

$$\frac{binding_\Sigma(a) = \{c_1 \text{ of } \alpha_1 | \ldots | c_n \text{ of } \alpha_n\} \quad \Gamma; \Delta, x_i : \alpha_i \vdash_\Sigma t_i : \beta \;\; (1 \le i \le n)}{\Gamma; \Delta \vdash_\Sigma \{(c_1\, x_1) \to t_1 | \ldots | (c_n\, x_n) \to t_n\} : a \multimap \beta} \quad \text{(case)}$$

In Rules (lin. var.) and (var.), x must be fresh with respect to Γ. In Rule (weak.), x must be fresh with respect to both Γ and Δ. Moreover, t must be either a λ-variable, a constant, or a variant constructor. In Rule (abs.), x cannot occur free in Δ.

Both the abstract and the object language of an ACG are defined modulo an appropriate notion of equality between λ-terms. In the original definition of an ACG, this notion of equality is the usual relation of $\beta\eta$-conversion. In the present setting, an equivalence relation akin to β-conversion may be defined as the reflexive, transitive, symmetric closure of the reduction relation induced by the following rules.

$$
\begin{array}{rll}
(\lambda^\circ x.\, t)\, u \to t[x{:=}u] & \quad & (\textit{Linear } \beta\textit{-reduction}) \\
(\lambda x.\, t)\, u \to t[x{:=}u] & \quad & (\beta\textit{-reduction}) \\
[l_1 = t_1; \ldots; l_n = t_n].l_i \to t_i & \quad & (\textit{Record selection}) \\
\{(c_1\, x_1) \to t_1 | \ldots | (c_n\, x_n) \to t_n\}\, (c_i\, u) \to t_i[x_i{:=}u] & \quad & (\textit{Case analysis})
\end{array}
$$

This reduction relation satisfies the properties of confluence, subject reduction, and strong normalization. Hence, the corresponding equivalence relation is

decidable. Nevertheless, it amounts to a rather weak form of equality. In order to obtain a stronger notion of equality, one should also consider η-like conversion rules and permutative conversion rules. We will not discuss these in the present paper.

Other theoretical questions concern the decidability and the tractability of parsing. It is not difficult to show that ACGs based on this extended typing system have an undecidable membership problem. This follows from the fact that the Edinburgh logical framework [6] appear to be a subsystem of the present typing system.[8] This raises the problem of isolating interesting fragments that have a decidable membership problem.

5   A small example

In order to illustrate the expressive power of the extensions we have introduced, we provide a toy example related to gender agreement in French. This example implements the rule saying that, in the plural, the masculine is used by default when referring to a group of mixed genders.

We first give an abstract signature:

$$
\begin{aligned}
gender &= \{\mathrm{m}\,|\,\mathrm{f}\} : \mathrm{type} \\
number &= \{\mathrm{s}\,|\,\mathrm{p}\} : \mathrm{type} \\
m\_feat &= [\mathrm{g}:gender;\ \mathrm{n}:number] : \mathrm{type} \\
N, NP &: (m\_feat)\mathrm{type} \\
S &: \mathrm{type} \\
\text{PIERRE, JEAN} &: NP[\mathrm{g}=\mathrm{m};\ \mathrm{n}=\mathrm{s}] \\
\text{MARIE, ALICE} &: NP[\mathrm{g}=\mathrm{f};\ \mathrm{n}=\mathrm{s}] \\
\text{ETRE} &: (\Pi x:m\_feat) \\
&\quad ((\Pi y:m\_feat)(N\,y) \multimap NP\,x \multimap S) \\
\text{ET} &: (\Pi xy:m\_feat) \\
&\quad (NP\,x \multimap NP\,y \multimap NP[\mathrm{g}=C\,x\,y;\ \mathrm{n}=\mathrm{p}]) \\
\text{MATHEMATICIEN} &: (\Pi x:m\_feat)N\,x
\end{aligned}
$$

where

$$
C\,x\,y = \{\mathrm{m} \to \mathrm{m}\,|\,\mathrm{f} \to \{\mathrm{m} \to \mathrm{m}\,|\,\mathrm{f} \to \mathrm{f}\}\,(y.\mathrm{g})\}\,(x.\mathrm{g})
$$

is the term specifying that the gender of a conjunction is feminine if and only if both conjuncts are of feminine gender.

---

[8] Actually, the notion of dependent type we use is slightly weaker. This, however, does not affect the undecidability result

We then consider the following object signature:

$$gender = \{\mathrm{m} \mid \mathrm{f}\} : \mathrm{type}$$
$$number = \{\mathrm{s} \mid \mathrm{p}\} : \mathrm{type}$$
$$m\_feat = [\mathrm{g} : gender; \mathrm{n} : number] : \mathrm{type}$$
$$s : \mathrm{type}$$
$$\text{/Pierre/, /Jean/, /Marie/, /Alice/, /est/, /sont/, /et/,}$$
$$\text{/mathématicien/, /mathématicienne/,}$$
$$\text{/mathématiciens/, /mathématiciennes/} : s \multimap s$$

Finally, we define the following lexicon:[9]

$$gender := gender$$
$$number := number$$
$$m\_feat := m\_feat$$
$$N, NP := \lambda x.\, s \multimap s$$
$$S := s \multimap s$$
$$\textsc{pierre} := \text{/Pierre/}$$
$$\textsc{jean} := \text{/Jean/}$$
$$\textsc{marie} := \text{/Marie/}$$
$$\textsc{alice} := \text{/Alice/}$$
$$\textsc{etre} := \lambda m.\, \lambda^\circ xy.\, y + (\{\, \mathrm{s} \to \text{/est/} \mid \mathrm{p} \to \text{/sont/} \,\}\, m.\mathrm{n}) + x\, m$$
$$\textsc{et} := \lambda m_1 m_2.\, \lambda^\circ xy.\, x + \text{/et/} + y$$
$$\textsc{mathematicien} := \lambda m.\, \{\, \mathrm{m} \to \{\, \mathrm{s} \to \text{/mathématicien/}$$
$$\mid \mathrm{p} \to \text{/mathématiciens/} \,\}\, m.\mathrm{n}$$
$$\mid \mathrm{f} \to \{\, \mathrm{s} \to \text{/mathématicienne/}$$
$$\mid \mathrm{p} \to \text{/mathématiciennes/} \,\}\, m.\mathrm{n} \,\}\, m.\mathrm{g}$$

Let us write [ms], [mp], [fs], and [fp] as abbreviations for [g = m; n = s], [g = m; n = p], [g = f; n = s], [g = f; n = p], respectively. We then have that the following terms are well-typed λ-terms belonging to the abstract language of the grammar:

<div style="text-align:center">

etre [ms] mathematicien pierre

etre [fp] mathematicien (et [fs] [fs] marie alice)

etre [mp] mathematicien (et [fs] [ms] marie pierre)

</div>

Their images by the lexicon yield respectively the following object terms:

/Pierre/ + /est/ + /mathématicien/

/Marie/ + /et/ + /Alice/ + /sont/ + /mathématiciennes/

/Marie/ + /et/ + /Pierre/ + /sont/ + /mathématiciens/

---

[9] As usual, strings are identified with λ-terms of type $s \multimap s$. Then, in case t and u are strings, $t + u$ stands for $\lambda x.\, t\, (u\, x)$.

References

1. Cervesato, I. and Pfenning, F. A linear logical framework. *Information & Computation*, 179(1): 19–75, 2002.
2. Curry, H.B. Some logical aspects of grammatical structure. In: *Proceedings of symposia in applied mathematics*, volume XII, pp. 56–68. 1961.
3. de Groote, Ph. Towards abstract categorial grammars. In: *Association for Computational Linguistics, 39th Annual Meeting and 10th Conference of the European Chapter, Proceedings of the Conference*, pp. 148–155, 2001.
4. de Groote, Ph. Tree-Adjoining Grammars as Abstract Categorial Grammars. In: *TAG+6, Proceedings of the sixth International Workshop on Tree Adjoining Grammars and Related Frameworks*, pp. 145–150, 2001.
5. de Groote, Ph. and Pogodalla, S. On the Expressive Power of Abstract Categorial Grammars: Representing Context-Free Formalisms. *Journal of Logic, Language and Information* 13(4): 421–438, 2004.
6. Harper, R., Honsel, F., and Plotkin, G. A framework for defining logics *Proceedings of the second annual IEEE symposium on logic in computer science* 194–204, 1987.
7. Martin-Löf, P. An Intuitionistic Theory of Types: Predicative Part. In: *Logic Colloquium '73*, North-Holland, pp. 73–118, 1975.
8. Moortgat, M. Categorial type logic. In J. van Benthem and A. ter Meulen, editors, *Handbook of Logic and Language*, chapter 2. Elsevier, 1997.
9. Morrill, G. *Type Logical Grammar: Categorial Logic of Signs.* Kluwer Academic Publishers, 1994.
10. Ranta, A. *Type theoretical grammar.* Oxford University Press, 1994.
11. Ranta, A. Grammatical Framework: A Type-Theoretical Grammar Formalism. *Journal of Functional Programming*, 14(2): 145–189, 2004.
12. Salvati, S. *Problèmes de filtrage et problèmes d'analyse pour les grammaires catégorielles abstraites.* Thèse de Doctorat. Institut National Polytechnique de Lorraine, 2005

Second-Order Abstract Categorial Grammars as Hyperedge Replacement Grammars

Makoto Kanazawa[*]

National Institute of Informatics
2–1–2 Hitotsubashi, Chiyoda-ku, Tokyo, 101–8430, Japan

Abstract. Second-order *abstract categorial grammars* (de Groote 2001) and *hyperedge replacement grammars* (see Engelfriet 1997) are two natural ways of generalizing "context-free" grammar formalisms for string and tree languages. It is known that the string generating power of both formalisms is equivalent to (non-erasing) *multiple context-free grammars* (Seki et al. 1991) or *linear context-free rewriting systems* (Weir 1988). In this paper, we give a simple, direct proof of the fact that second-order ACGs are simulated by hyperedge replacement grammars, which implies that the string and tree generating power of the former is included in that of the latter. The normal form for tree-generating hyperedge replacement grammars given by Engelfriet and Maneth (2000) can then be used to show that the tree generating power of second-order ACGs is exactly the same as that of hyperedge replacement grammars.

1 Introduction

When an *abstract categorial grammar* (de Groote 2001) has a second-order abstract vocabulary, terms in its abstract language do not contain any λ-abstraction and can hence be regarded as trees. Since in this case the abstract language is a *local* tree language, second-order ACGs belong to a long list of grammar formalisms that have "context-free" derivations. De Groote and Pogodalla (2004) showed that such well-known "context-free" grammar formalisms as *linear context-free rewriting systems* (Weir 1988), or non-erasing *multiple context-free grammars* (Seki et al. 1991), and linear and non-deleting *context-free tree grammars* (Rounds 1970, Engelfriet and Schmidt 1977) can be faithfully encoded by second-order ACGs. Subsequently, Salvati (2007) showed that the string generating power of second-order ACGs exactly matches that of LCFRSs using the fact that the latter coincides with the class of output languages of *deterministic tree-walking transducers* (Weir 1992). The ability of second-order ACGs to capture the above-mentioned "context-free" grammar formalisms is explained by the fact that linear λ-terms—the data structure used by ACGs—generalize both strings and trees and can express "linear and non-deleting" operations on them, such as concatenation and second-order tree substitution.

Another very general "context-free" grammar formalism, encompassing both string grammars and tree grammars, is that of *hyperedge replacement grammars* (see Engelfriet 1997), which generate sets of *hypergraphs*. Hypergraphs are a generalization of graphs that allow edges to be incident on any number of nodes, and can represent strings and trees in a straightforward way. The operation of *hyperedge replacement* can express many operations on strings and trees, including concatenation and second-order tree substitution. It is known that the string generating power of hyperedge replacement grammars coincides with the class of output languages of deterministic tree-walking transducers (Engelfriet and Heyker 1989). By Salvati's (2007) result, this implies the equivalence of second-order ACGs and hyperedge replacement grammars in string generating power.

Since second-order ACGs and hyperedge replacement grammars are two natural ways of generalizing "context-free" string grammars and tree grammars, it is interesting to know whether their tree generating power is also the same. In this paper, we give a very simple proof of the fact that second-order ACGs are "simulated" by hyperedge replacement grammars, which implies that the the generating power of the former is included in that of the latter both in the case of strings and in the case of trees. The normal form for tree-generating hyperedge replacement grammars given by Engelfriet and Maneth (2000) can then be used to show that the tree generating power of the two formalisms is exactly the same.

2 Preliminaries

2.1 Abstract Categorial Grammars

We assume that the reader is familiar with Kanazawa 2006.[1] For standard notions in simply typed λ-calculus, see Hindley 1997 or Sørensen and Urzyczyn 2006. Briefly, a *higher-order signature* is $\Sigma = (A, C, \tau)$, where A is a finite set of *atomic types*, C is a finite set of *constants*, and τ is a type assignment to constants. A λ-term M over Σ with free variables $x_1, \ldots, x_n$ may be assigned a type α under a *type environment* $x_1 : \alpha_1, \ldots, x_n : \alpha_n$, or in symbols: $x_1 : \alpha_1, \ldots, x_n : \alpha_n \vdash_\Sigma M : \alpha$. An *abstract categorial grammar* (de Groote 2001) is $\mathscr{G} = (\Sigma, \Sigma', \mathscr{L}, s)$, where $\Sigma = (A, C, \tau)$ (*abstract vocabulary*) and $\Sigma' = (A', C', \tau')$ (*object vocabulary*) are higher-order signatures, the *lexicon* $\mathscr{L}$ maps each atomic type in A to a type built upon A' and each abstract constant c in C to a closed linear λ-term $\mathscr{L}(c)$ over Σ' such that $\vdash_{\Sigma'} \mathscr{L}(c) : \mathscr{L}(\tau(c))$. The *abstract language* of $\mathscr{G}$, written $\mathcal{A}(\mathscr{G})$, is the set of β-normal closed linear λ-terms over Σ of type s, and the *object language* of $\mathscr{G}$ is $\mathcal{O}(\mathscr{G}) = \{ |\mathscr{L}(P)|_\beta \mid P \in \mathcal{A}(\mathscr{G}) \}$, where $|\mathscr{L}(P)|_\beta$ is the β-normal form of $\mathscr{L}(P)$. An ACG is *n-th order* if the *order* of the type of each abstract constant does not exceed n. The notation $\mathbf{G}(n, m)$

---

[1] The errata for this paper is available at
  `http://research.nii.ac.jp/~kanazawa/publications/afacl_corrections.pdf`.

denotes the class of n-th order ACGs whose lexicon maps each atomic abstract type to a type of order $\leq m$.

Recall that a *ranked alphabet* is a finite set $\Delta = \bigcup_{n \in \mathbb{N}} \Delta^{(n)}$, where $\Delta^{(m)} \cap \Delta^{(n)} = \varnothing$ if $m \neq n$. If $f \in \Delta^{(n)}$, n is the *rank* of f, written $\mathrm{rank}_\Delta(f)$. A *tree* over Δ is an expression $f T_1 \ldots T_n$, where $\mathrm{rank}_\Delta(f) = n$ and $T_1, \ldots, T_n$ are trees over Δ. The set of trees over Δ is denoted $\mathbb{T}_\Delta$. Let $Y_k = \{y_1, \ldots, y_k\}$ be a set of k variables. The notation $\mathbb{T}_\Delta(Y_k)$ denotes the set $\mathbb{T}_{\Delta \cup Y_k}$, where $\mathrm{rank}_{\Delta \cup Y_k}(y_i) = 0$ for all $y_i \in Y_k$.

If Δ is a ranked alphabet, $\mathrm{inc}(\Delta)$ is the ranked alphabet such that $(\mathrm{inc}(\Delta))^{(0)} = \varnothing$ and $(\mathrm{inc}(\Delta))^{(n+1)} = \Delta^{(n)}$. If Δ is a ranked alphabet with $\Delta^{(0)} = \varnothing$, then $\mathrm{dec}(\Delta)$ is the ranked alphabet such that $(\mathrm{dec}(\Delta))^{(n)} = \Delta^{(n+1)}$.

A ranked alphabet Δ can be represented by a second-order signature $\Sigma_\Delta = (\{o\}, \Delta, \tau_\Delta)$, where for each $f \in \Delta^{(n)}$, $\tau_\Delta(f) = o^n \to o$. A tree in $\mathbb{T}_\Delta(Y_k)$ can be identified with a β-normal λ-term T over Σ_Δ such that $\mathrm{FV}(T) \subseteq Y_k$ and $\{\, y_i : o \mid y_i \in \mathrm{FV}(T) \,\} \vdash_{\Sigma_\Delta} T : o$.

2.2 Hyperedge Replacement Grammars

For the most part we follow Engelfriet and Heyker (1991) and Engelfriet (1997).

A *hypergraph* over a ranked alphabet Δ is a tuple $H = (V, E, \mathrm{nod}, \mathrm{lab}, \mathrm{ext})$, where V and E are disjoint finite sets, $\mathrm{nod}\colon E \to V^*$, $\mathrm{lab}\colon E \to \Delta$, and $\mathrm{ext} \in V^*$. Elements of V and E are nodes and *hyperedges*, respectively, nod is the *incidence function*, lab is the *labeling function*, and ext is the sequence of *external nodes*. It is required that for every $e \in E$, $\mathrm{rank}_\Delta(\mathrm{lab}(e)) = |\mathrm{nod}(e)|$. If $\mathrm{nod}(e) = (v_1, \ldots, v_m)$, then we write $\mathrm{nod}(e, i)$ for v_i, and if $\mathrm{ext} = (v_1, \ldots, v_m)$, we write $\mathrm{ext}(i)$ for v_i. The *type* of H is defined as $\mathrm{type}(H) = |\mathrm{ext}|$. We often refer to the components of H as $V_H, E_H, \mathrm{nod}_H, \mathrm{lab}_H, \mathrm{ext}_H$. It is customary to identify hypergraphs that are isomorphic to each other.

Let $H = (V_H, E_H, \mathrm{nod}_H, \mathrm{lab}_H, \mathrm{ext}_H)$ be a hypergraph and $R \subseteq V_H \times V_H$. The result of identifying every pair of nodes $(v, v') \in R$ in H is defined as $H/R = (V_H/\!\equiv_R, E_H, \mathrm{nod}, \mathrm{lab}_H, \mathrm{ext})$, where $\equiv_R$ is the smallest equivalence relation on V_H extending R, $\mathrm{nod}(e, i) = [\mathrm{nod}_H(e, i)]_{\equiv_R}$, and $\mathrm{ext}(i) = [\mathrm{ext}_H(i)]_{\equiv_R}$. Note that if we think of the nodes in V_H as drawn from some universal set U and a mapping σ on U is a most general unifier of the pairs in R, then H/R is isomorphic to $H' = (V', E_H, \mathrm{nod}', \mathrm{lab}_H, \mathrm{ext}')$, where $V' = \{\, v\sigma \mid v \in V_H \,\}$, $\mathrm{nod}'(e) = \mathrm{nod}_H(e)\,\sigma$ and $\mathrm{ext}' = \mathrm{ext}_H \sigma$.

Let $K = (V_K, E_K, \mathrm{nod}_K, \mathrm{lab}_K, \mathrm{ext}_K)$ and $H = (V_H, E_H, \mathrm{nod}_H, \mathrm{lab}_H, \mathrm{ext}_H)$ be hypergraphs and let $e \in E_K$ be such that $|\mathrm{nod}_K(e)| = \mathrm{type}(H)$. Then the result of substituting H for e in K is $K[e := H] = \bar{K}/R$, where

$$\bar{K} = (V, E, \mathrm{nod}, \mathrm{lab}, \mathrm{ext}),$$
$$V = V_K \cup V_H,$$
$$E = (E_K - \{e\}) \cup E_H,$$
$$\mathrm{nod} = (\mathrm{nod}_K \cup \mathrm{nod}_H){\restriction}E,$$
$$\mathrm{lab} = (\mathrm{lab}_K \cup \mathrm{lab}_H){\restriction}E,$$

$$\text{ext} = \text{ext}_K,$$
$$R = \{\,(\text{nod}_K(e, i), \text{ext}_H(i)) \mid 1 \leq i \leq \text{type}(H)\,\}.$$

In the definition of V and E, it is assumed that $V_K \cap V_H = E_K \cap E_H = \varnothing$ (taking isomorphic copies when necessary).

If e_1 and e_2 are distinct edges of K, one can show that $K[e_1 := H_1][e_2 := H_2] = K[e_2 := H_2][e_1 := H_1]$. The simultaneous substitution $K[e_1 := H_1, \ldots, e_m := H_m]$ is defined as $K[e_1 := H_1] \ldots [e_m := H_m]$.

A *hyperedge replacement grammar* is a tuple $G = (N, \Delta, P, S)$, where N and Δ are disjoint ranked alphabets, $S \in N$, and P is a finite set of *productions*, each of the form

$$B \to H,$$

where $B \in N$, H is a hypergraph over $N \cup \Delta$, and $\text{type}(H) = \text{rank}_N(B)$. Elements of N and Δ are called *nonterminals* and *terminals*, respectively, and S is called the *initial nonterminal*. If H is a hypergraph over $N \cup \Delta$, $\text{nont}(H)$ is the list consisting of the edges in H labeled with nonterminals, in some fixed order. If $\text{nont}(H) = (e_1, \ldots, e_m)$, we write $\text{nont}(H, i)$ for e_i.

We associate with G a second-order signature $\Sigma_G = (N, P, \tau)$, where $\tau(\pi) = B_1 \to \cdots \to B_n \to B$ if $\pi = B \to H$ and $\text{nont}(H) = (e_1, \ldots, e_n)$ with $\text{lab}(e_i) = B_i$.[2] A *derivation tree* of G is a β-normal closed λ-term of atomic type over Σ_G. A derivation tree T is *complete* if it is of type S. If $\pi = B \to H$, $\text{nont}(H) = (e_1, \ldots, e_n)$, and $T = \pi T_1 \ldots T_n$ is a derivation tree of G, then $\text{yield}_G(T) = H[e_1 := \text{yield}_G(T_1), \ldots, e_n := \text{yield}_G(T_n)]$. The language of G is defined as

$$L(G) = \{\,\text{yield}_G(T) \mid T \text{ is a complete derivation tree of } G\,\}.$$

The *string graph* of a string $a_1 \ldots a_n$ is $\text{sgr}(a_1 \ldots a_n) = (V, E, \text{nod}, \text{lab}, \text{ext})$, where

$$V = \{0, \ldots, n\},$$
$$E = \{e_1, \ldots, e_n\},$$
$$\text{nod}(e_i) = (i - 1, i) \quad \text{for } i = 1, \ldots, n,$$
$$\text{lab}(e_i) = a_i,$$
$$\text{ext} = (0, n).$$

A tree $f T_1 \ldots T_n \in \mathbb{T}_\Delta$ is represented by a *tree graph* of type 1, namely, $\text{gr}(f T_1 \ldots T_n) = (V, E, \text{nod}, \text{lab}, \text{ext})$, where

$$V = \{v_0\} \cup \bigcup_{i=1}^{n} V_{\text{gr}(T_i)},$$
$$E = \{e_0\} \cup \bigcup_{i=1}^{n} E_{\text{gr}(T_i)},$$

[2] A more standard treatment uses a context-free grammar rather than a second-order signature.

$$\mathrm{nod}(e) = \begin{cases} (v_1, \ldots, v_n, v_0) & \text{if } e = e_0, \text{ where } \mathrm{ext}_{\mathrm{gr}(T_i)} = (v_i) \text{ for } i = 1, \ldots, n, \\ \mathrm{nod}_{\mathrm{gr}(T_i)}(e) & \text{if } e \in E_{\mathrm{gr}(T_i)}, \end{cases}$$

$$\mathrm{lab}(e) = \begin{cases} f & \text{if } e = e_0, \\ \mathrm{lab}_{\mathrm{gr}(T_i)}(e) & \text{if } e \in E_{\mathrm{gr}(T_i)}, \end{cases}$$

$$\mathrm{ext} = (v_0).$$

In the definition of V and E, the union is assumed to be over $n+1$ disjoint sets. If $T \in \mathbb{T}_\Delta$, then $\mathrm{gr}(T)$ is a hypergraph over $\mathrm{inc}(\Delta)$.

The notation $STR(HR)$ denotes the class of all string languages L such that there exists a hyperedge replacement grammar that generates $\{\,\mathrm{sgr}(w) \mid w \in L\,\}$. Similarly, $TR(HR)$ is the class of tree languages L such that there exists a hyperedge replacement grammar that generates $\{\,\mathrm{gr}(T) \mid T \in L\,\}$.

Example 1. The following grammar, taken from Engelfriet and Maneth (2000), with the ranked alphabet of nonterminals $N = N^{(1)} \cup N^{(4)} = \{S\} \cup \{A\}$, generates $\{\,\mathrm{gr}\big(f(a^n(b^n e))(a^n(b^n e))\big) \mid n \geq 0\,\}$. Here, dots represent nodes, and dots with numbers attached are external nodes. Hyperedges are represented by boxes with labels inside and *tentacles* connecting them to the nodes that they are incident on. The tentacles are ordered counterclockwise starting from the 9 o'clock position, with the exception of the A-labeled hyperedges, where the order is indicated by numbers in small type.
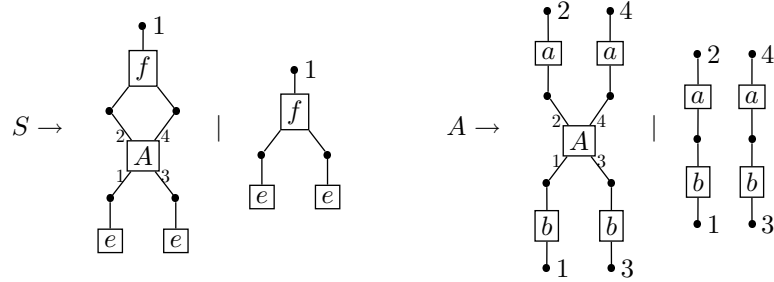
3   From Second-Order ACGs to Hyperedge Replacement Grammars

If α is a type, we let $\overline{\alpha}$ denote the sequence of atomic types occurring in α, listed from left to right. Given a closed λ-term M, the notation $\overrightarrow{\mathrm{Con}}(M)$ denotes the sequence $(d_1, \ldots, d_m)$ of constants occurring in M; $\widehat{M}[y_1, \ldots, y_m]$ is the pure λ-term (i.e., λ-term without constants) with $y_1, \ldots, y_m$ as its free variables such that $\widehat{M}[d_1, \ldots, d_m] = M$. Note that $\vdash_\Sigma M : \alpha$ if and only if $y_1 : \tau(d_1), \ldots, y_m : \tau(d_m) \vdash \widehat{M}[y_1, \ldots, y_m] : \alpha$.

Let M be a λ-term over $\Sigma = (A, C, \tau)$. Assume that $\overrightarrow{\mathrm{Con}}(M) = (d_1, \ldots, d_m)$, and

$$y_1 : \beta_1, \ldots, y_m : \beta_m \vdash \widehat{M}[y_1, \ldots, y_m] : \alpha$$

is a *principal typing* of $\widehat{M}[y_1, \ldots, y_m]$.[3] The hypergraph $\mathrm{graph}(M)$ is defined as follows:

$$\mathrm{graph}(M) = (V, E, \mathrm{nod}, \mathrm{lab}, \mathrm{ext}),$$

where

$$V = \text{the set of atomic types in } \beta_1, \ldots, \beta_m, \alpha,$$
$$E = \{y_1, \ldots, y_m\},$$
$$\mathrm{nod}(y_i) = \overline{\beta_i},$$
$$\mathrm{lab}(y_i) = d_i,$$
$$\mathrm{ext} = \overline{\alpha}.$$

If T is a tree in $\mathbb{T}_\Delta$, then $\mathrm{graph}(T)$, where T is viewed as a closed λ-term over Σ_Δ, is isomorphic to $\mathrm{gr}(T)$. Also, for every string $a_1 \ldots a_n$, $\mathrm{graph}(/a_1 \ldots a_n/)$ is the same as $\mathrm{sgr}(a_n \ldots a_1)$, the string graph of the reversal of $a_1 \ldots a_n$.[4]

Lemma 1. *Let M be a closed linear λ-term. If $M \twoheadrightarrow_\beta M'$, then $\mathrm{graph}(M)$ and $\mathrm{graph}(M')$ are isomorphic.*

Proof. Let $(d_1, \ldots, d_m) = \overrightarrow{\mathrm{Con}}(M)$. Since M is linear, M reduces to M' by non-erasing non-duplicating β-reduction. This implies that for some permutation ρ of $\{1, \ldots, m\}$, $\overrightarrow{\mathrm{Con}}(M') = (d_{\rho(1)}, \ldots, d_{\rho(m)})$ and $\widehat{M}[y_1, \ldots, y_m] \twoheadrightarrow_\beta \widehat{M'}[y_{\rho(1)}, \ldots, y_{\rho(m)}]$ by non-erasing non-duplicating β-reduction. By the Subject Reduction and Subject Expansion Theorems (see Hindley 1997), $\widehat{M}[y_1, \ldots, y_m]$ and $\widehat{M'}[y_{\rho(1)}, \ldots, y_{\rho(m)}]$ share the same principal typing. It follows that $\mathrm{graph}(M)$ and $\mathrm{graph}(M')$ are isomorphic. $\square$

Let $\mathscr{G} = (\Sigma, \Sigma', \mathscr{L}, s)$ be a second-order ACG with $\Sigma = (A, C, \tau)$ and $\Sigma' = (A', C', \tau')$. We assume that for every c, $\mathscr{L}(c)$ is in η-long form relative to $\mathscr{L}(\tau(c))$, so that every $M \in \mathcal{O}(\mathscr{G})$ is in η-long β-normal form relative to $\mathscr{L}(s)$.

We associate with $\mathscr{G}$ a certain hyperedge replacement grammar $\mathrm{hr}(\mathscr{G})$, where each abstract constant c of $\mathscr{G}$ corresponds to a production π_c of $\mathrm{hr}(\mathscr{G})$. If $\tau(c) = p_1 \to \cdots \to p_n \to p_0$, then we define

$$(1) \qquad\qquad M_c = \mathscr{L}(c)p_1 \ldots p_n.$$

Note that M_c is a closed λ-term over $\Sigma'' = (C' \cup A, A', \tau'')$, where $\tau''(c) = \tau'(c)$ for $c \in C'$ and $\tau''(p) = \mathscr{L}(p)$ for $p \in A$. Let π_c be the production

$$p_0 \to \mathrm{graph}(M_c).$$

The hyperedge replacement grammar associated with $\mathscr{G}$ is $\mathrm{hr}(\mathscr{G}) = (A, C', P, s)$, where

$$P = \{\, \pi_c \mid c \in C \,\}.$$

---

[3] A principal typing of a λ-term M is a typing of M from which all other typings of M can be obtained by type substitution.

[4] As in de Groote and Pogodalla 2004 and Kanazawa 2006, $/a_1 \ldots a_n/$ is the λ-term $\lambda z.a_1(\ldots (a_n z) \ldots)$ representing $a_1 \ldots a_n$.

Note that the mapping $c \mapsto \pi_c$ induces an isomorphism from Σ to $\Sigma_{\mathrm{hr}(\mathscr{G})}$. Thus, the closed λ-terms of atomic type over Σ may be identified with the derivation trees of $\mathrm{hr}(\mathscr{G})$. In particular, the elements of the abstract language of $\mathscr{G}$ are identical to the complete derivation trees of $\mathrm{hr}(\mathscr{G})$.

Lemma 2. *Let $\mathscr{G} = (\Sigma, \Sigma', \mathscr{L}, \tau)$ be a second-order ACG. For every closed λ-term T over Σ of atomic type, $\mathrm{yield}_{\mathrm{hr}(\mathscr{G})}(T) = \mathrm{graph}(\mathscr{L}(T))$.*

Proof. Induction on T. Suppose that $T = cT_1 \ldots T_n$, $\tau(c) = p_1 \to \cdots \to p_n \to p$, and $\mathrm{yield}_{\mathrm{hr}(\mathscr{G})}(T_i) = \mathrm{graph}(\mathscr{L}(T_i))$ for $i = 1, \ldots, n$. Let

$$\overrightarrow{\mathrm{Con}}(\mathscr{L}(c)) = (d_1, \ldots, d_m),$$
$$\overrightarrow{\mathrm{Con}}(\mathscr{L}(T_i)) = (f_{i,1}, \ldots, f_{i,m_i}) \quad \text{for } i = 1, \ldots, n.$$

Let

$$\Gamma_0 \vdash \widehat{\mathscr{L}(c)}[y_1, \ldots, y_m] : \alpha_1 \to \cdots \to \alpha_n \to \alpha_0$$

be a principal typing of $\widehat{\mathscr{L}(c)}[y_1, \ldots, y_m]$. Then

$$\Gamma_0, x_1 : \alpha_1, \ldots, x_n : \alpha_n \vdash \widehat{M_c}[y_1, \ldots, y_m, x_1, \ldots, x_n] : \alpha_0$$

is a principal typing of $\widehat{M_c}[y_1, \ldots, y_m, x_1, \ldots, x_n]$ and $\mathrm{nont}(\mathrm{graph}(M_c)) = (x_1, \ldots, x_n)$. Let

$$\Gamma_i \vdash \widehat{\mathscr{L}(T_i)}[z_{i,1}, \ldots, z_{i,m_i}] : \delta_i$$

be a principal typing of $\widehat{\mathscr{L}(T_i)}[z_{i,1}, \ldots, z_{i,m_i}]$. Clearly,

$$(\Gamma_0, \Gamma_1, \ldots, \Gamma_m)\sigma \vdash$$
$$\widehat{\mathscr{L}(c)}[y_1, \ldots, y_m]\widehat{\mathscr{L}(T_1)}[z_{1,1}, \ldots, z_{1,m_1}] \ldots \widehat{\mathscr{L}(T_n)}[z_{n,1}, \ldots, z_{n,m_n}] : \alpha_0\sigma$$

is a principal typing of $\widehat{\mathscr{L}(T)}[y_1, \ldots, y_m, z_{1,1}, \ldots, z_{n,m_n}]$, where σ is a most general unifier of the equations

$$\alpha_i = \delta_i, \quad i = 1, \ldots, n.$$

From this, it easily follows that

$$\mathrm{graph}(\mathscr{L}(T)) = \mathrm{graph}(M_c)[x_1 := \mathrm{graph}(\mathscr{L}(T_1)), \ldots, x_n := \mathrm{graph}(\mathscr{L}(T_n))].$$

By induction hypothesis, we have

$$\begin{aligned}
\mathrm{yield}_{\mathrm{hr}(\mathscr{G})}(T) &= \mathrm{graph}(M_c)[x_1 := \mathrm{yield}_{\mathrm{hr}(\mathscr{G})}(T_1), \ldots, x_n := \mathrm{yield}_{\mathrm{hr}(\mathscr{G})}(T_n)] \\
&= \mathrm{graph}(M_c)[x_1 := \mathrm{graph}(\mathscr{L}(T_1)), \ldots, x_n := \mathrm{graph}(\mathscr{L}(T_n))] \\
&= \mathrm{graph}(\mathscr{L}(T)).
\end{aligned}$$

$\square$

Theorem 3. *For every second-order ACG $\mathscr{G}$, we have*

$$L(\mathrm{hr}(\mathscr{G})) = \{\, \mathrm{graph}(M) \mid M \in \mathcal{O}(\mathscr{G}) \,\}.$$

Proof. By Lemma 2, $L(\mathrm{hr}(\mathscr{G})) = \{\, \mathrm{graph}(\mathscr{L}(T)) \mid T \in \mathcal{A}(\mathscr{G}) \,\}$. If $M = |\mathscr{L}(T)|_\beta$, then by Lemma 1, $\mathrm{graph}(M)$ is isomorphic to $\mathrm{graph}(\mathscr{L}(T))$.   □

Corollary 4. *The class of tree languages generated by second-order ACGs is included in TR(HR). The class of string languages generated by second-order ACGs is included in STR(HR).*

4 From Hyperedge Replacement Grammars to Second-Order ACGs

It is known that $STR(HR)$ exactly equals the class of multiple context-free languages (or, equivalently, LCFRS languages). This follows from the results by Engelfriet and Heyker (1991) and by Weir (1992), but an outline of a more direct proof is given by Engelfriet (1997). Since de Groote and Pogodalla (2004) show that the LCFRSs can be encoded by second-order ACGs in $\mathbf{G}(2,4)$, we have

Theorem 5. *The class of string languages generated by second-order ACGs in $\mathbf{G}(2,4)$ includes $STR(HR)$.*

Corollary 6. *The class of string languages generated by second-order ACGs equals $STR(HR)$.*

Corollary 7 (Salvati). *All string languages generated by second-order ACGs are generated by ACGs in $\mathbf{G}(2,4)$.*

The equivalence in string generating power between second-order ACGs and LCFRSs, and consequently Corollary 7, were first obtained by Salvati (2007), using Weir's (1992) result. The proof provided here avoids detour through deterministic tree-walking transducers, using a simple and direct translation from second-order ACGs to hyperedge replacement grammars.[5]

As for tree generating power, we can use the tree generating normal form for hyperedge replacement grammars due to Engelfriet and Maneth (2000).

A *simple* tree $T \in \mathbb{T}_\Delta(Y_k)$, where each $y_i \in Y_k$ occurs exactly once, is represented by a *tree graph* of type $k+1$, namely, $\mathrm{gr}(T) = \mathrm{graph}(\lambda y_1 \ldots y_k.T)$ (viewing T as a linear λ-term with $\mathrm{FV}(T) = Y_k$).

---

[5] The point here is that LCFRSs, second-order ACGs, and hyperedge replacement grammars are all similar formalisms, and the translations from LCFRSs to second-order ACGs and then to hyperedge replacement grammars are very straightforward. The only hard work is in the direction from hyperedge replacement grammars to LCFRSs, which uses a transformation of string generating hyperedge replacement grammars due to Habel (1992).

A *forest* (i.e., sequence of trees) $T_1, \ldots, T_k$ is represented by the *forest graph* $\mathrm{gr}(T_1, \ldots, T_k) = \mathrm{gr}(T_1) \oplus \cdots \oplus \mathrm{gr}(T_k)$, where $\oplus$ denotes disjoint union, concatenating the sequences of external nodes. A *linked ranked alphabet* is a ranked alphabet Δ together with a mapping $f \in \Delta^{(k)} \mapsto \mathrm{link}(f) = (r_1, \ldots, r_n)$ such that $\sum_{i=1}^{n} r_i = k$. If H is a hypergraph over a linked ranked alphabet Δ, $\mathrm{cut}(H)$ is the result of replacing every hyperedge e with label f, where $\mathrm{link}(f) = (r_1, \ldots, r_n)$, by distinct new hyperedges $e_1, \ldots, e_n$ such that e_j is incident with the $(\sum_{i=1}^{j-1} r_i) + 1$-th up to the $\sum_{i=1}^{j}$-th node of e and has any label with rank r_j. A hypergraph H over a linked alphabet Δ is a *linked forest* of type $(r_1, \ldots, r_k)$ if $\mathrm{cut}(H) = H_1 \oplus \cdots \oplus H_k$ for some tree graphs $H_1, \ldots, H_k$ of type $r_1, \ldots, r_k$, respectively. A hyperedge replacement grammar $G = (N, \Delta, S, P)$ is in *tree generating normal form* if (i) $N \cup \Delta$ is a linked ranked alphabet such that $S \in N^{(1)}$ and for every $f \in \Delta^{(k)}$, $\mathrm{link}(f) = (k)$, and (ii) for every production $B \to H$ in P, H is a linked forest over $N \cup \Delta$ of type $\mathrm{link}(B)$.

The grammar in Example 1 is a hyperedge replacement grammar in tree generating normal form, where $\mathrm{link}(A) = (2, 2)$.

Theorem 8 (Engelfriet and Maneth). *For every tree generating hyperedge replacement grammar G, there is a hyperedge replacement grammar G' in tree generating normal form such that $L(G') = L(G)$.*

Let $G = (N, \Delta, P, S)$ be a hyperedge replacement grammar in tree generating normal form. Let H be a linked forest over $N \cup \Delta$ of type $(r_1, \ldots, r_k)$ with $\mathrm{cut}(H) = H_1 \oplus \cdots \oplus H_k$. Let $\mathrm{nont}(H) = (e_1, \ldots, e_n)$ and let $e_{i,1}, \ldots, e_{i,l_i}$ be the hyperedges of $H_1 \oplus \cdots \oplus H_k$ that come from e_i. Assume that the labels of $e_{i,j}$ in $H_1 \oplus \cdots \oplus H_k$ are distinct variables $z_{i,j}$, and let Z be the ranked alphabet consisting of all the $z_{i,j}$. For $i = 1, \ldots, k$, let $T_i \in \mathbb{T}_{\mathrm{dec}(\Delta \cup Z)}(Y_{r_i-1})$ be the simple tree such that $\mathrm{gr}(T_i) = H_i$. Define

$$
\begin{aligned}
M_H = \lambda x_1 x_2 \ldots x_n w. \\
\quad x_1\big(\lambda z_{1,1} \ldots z_{1,l_1}. \\
\quad x_2\big(\lambda z_{2,1} \ldots z_{2,l_2}. \\
\quad \vdots \\
\quad x_n\big(\lambda z_{n,1} \ldots z_{n,l_n}.w(\lambda y_1 \ldots y_{r_1-1}.T_1) \ldots (\lambda y_1 \ldots y_{r_k-1}.T_k)\big) \ldots \big)\big).
\end{aligned}
$$

M_H is a closed linear λ-term over Σ_Δ.

We now describe how to represent G by a second-order ACG $\mathrm{acg}(G) = (\Sigma, \Sigma_{\mathrm{dec}(\Delta)}, \mathscr{L}, s)$. The abstract vocabulary of $\mathrm{acg}(G)$ is $\Sigma = (N \cup \{s\}, P \cup \{d\}, \tau)$, where $(N, P, \tau \restriction P)$ is Σ_G, the second-order signature associated with G, and $\tau(d) = S \to s$. The lexicon $\mathscr{L}$ is defined as follows. First, $\mathscr{L}(s) = o$. For a nonterminal $B \in N$ with $\mathrm{link}(B) = (r_1, \ldots, r_k)$, we let

$$
\mathscr{L}(B) = ((o^{r_1-1} \to o) \to \cdots \to (o^{r_k-1} \to o) \to o) \to o.
$$

For a production $\pi = B \to H$ in P with $\mathrm{nont}(H) = (e_1, \ldots, e_n)$, we let

$$
\mathscr{L}(\pi) = M_H.
$$

Finally, we let $\mathscr{L}(d) = \lambda x.x(\lambda y.y)$. We can see that $\mathscr{L}$ is indeed a lexicon: $\vdash_{\Sigma_{\mathrm{dec}(\Delta)}} \mathscr{L}(c) : \mathscr{L}(\tau(c))$ for every $c \in P \cup \{d\}$. Note that $\mathrm{acg}(G) \in \mathbf{G}(2, 4)$.

Clearly, the derivation trees of G can be identified with the closed λ-terms over Σ of type $B \in N$.

Example 2. Let G be the hyperedge replacement grammar in Example 1. Then $\mathrm{acg}(G) = (\Sigma, \Sigma_\Delta, \mathscr{L}, s)$, where $\Delta = \Delta^{(0)} \cup \Delta^{(1)} \cup \Delta^{(2)} = \{e\} \cup \{a, b\} \cup \{f\}$ and

$$\Sigma = (\{S, A, s\}, \{\pi_1, \pi_2, \pi_3, \pi_4, d\}, \tau),$$

$$\tau(\pi_1) = A \to S, \quad \tau(\pi_2) = S, \quad \tau(\pi_3) = A \to A, \quad \tau(\pi_4) = A, \quad \tau(d) = S \to s,$$

$$\mathscr{L}(S) = (o \to o) \to o, \quad \mathscr{L}(A) = ((o \to o) \to (o \to o) \to o) \to o, \quad \mathscr{L}(s) = o,$$

$$\mathscr{L}(\pi_1) = \lambda x w.x(\lambda z_1 z_2.w(f(z_1 e)(z_2 e))),$$

$$\mathscr{L}(\pi_2) = \lambda w.w(fee),$$

$$\mathscr{L}(\pi_3) = \lambda x w.x(\lambda z_1 z_2.w(\lambda y.a(z_1(by)))(\lambda y.a(z_2(by)))),$$

$$\mathscr{L}(\pi_4) = \lambda w.w(\lambda y.a(by))(\lambda y.a(by)),$$

$$\mathscr{L}(d) = \lambda x.x(\lambda y.y).$$

Note that if H is a hypergraph, $H \oplus \mathrm{gr}(y_1)$ is obtained from H by adding one new node and counting it as external nodes twice: $H \oplus \mathrm{gr}(y_1) = (V_H \cup \{v\}, E_H, \mathrm{nod}_H, \mathrm{lab}_H, \mathrm{ext}_H{}^\frown(v, v))$.

Lemma 9. *Let G be a hyperedge replacement grammar in tree generating normal form and $\mathscr{L}$ be the lexicon of $\mathrm{acg}(G)$. If T is a derivation tree of G, then $\mathrm{graph}(\mathscr{L}(T)) = \mathrm{yield}_G(T) \oplus \mathrm{gr}(y_1)$.*

Proof. We prove the lemma by induction on T. Let $T = \pi T_1 \dots T_n$, where $\pi = B \to H$, $\mathrm{nont}(H) = (e_1, \dots, e_n)$, and $\mathrm{lab}(e_i) = B_i$. It is easy to see that $\mathrm{graph}(M_\pi)$ as defined in (1) in Sect. 3 is isomorphic to the graph $(V_H \cup \{v_1, \dots, v_{n+1}\}, E_H, \mathrm{nod}, \mathrm{lab}_H, \mathrm{ext})$ over $\mathrm{inc}(\mathrm{inc}(N)) \cup \Delta$, where

$$\mathrm{nod}(e) = \mathrm{nod}_H(e) \quad \text{if } \mathrm{lab}_H(e) \in \Delta,$$

$$\mathrm{nod}(e_i) = \mathrm{nod}_H(e_i){}^\frown(v_{i+1}, v_i),$$

$$\mathrm{ext} = \mathrm{ext}_H{}^\frown(v_{n+1}, v_1).$$

Then

$$\mathrm{graph}(\mathscr{L}(T)) = \mathrm{graph}(M_\pi)[e_1 := \mathrm{graph}(\mathscr{L}(T_1)), \dots, e_n := \mathrm{graph}(\mathscr{L}(T_n))]$$
$$\text{as in the proof of Lemma 2,}$$
$$= \mathrm{graph}(M_\pi)$$
$$[e_1 := \mathrm{yield}_G(T_1) \oplus \mathrm{gr}(y_1), \dots, e_n := \mathrm{yield}_G(T_n) \oplus \mathrm{gr}(y_1)]$$
$$\text{by induction hypothesis,}$$
$$= H[e_1 := \mathrm{yield}_G(T_1), \dots, e_n := \mathrm{yield}_G(T_n)] \oplus \mathrm{gr}(y_1)$$
$$\text{by the above characterization of } \mathrm{graph}(M_\pi),$$
$$= \mathrm{yield}_G(T) \oplus \mathrm{gr}(y_1).$$

$\square$

Theorem 10. *Let G be a hyperedge replacement grammar in tree generating normal form. Then $L(G) = \{\, \mathrm{graph}(M) \mid M \in \mathcal{O}(\mathrm{acg}(G)) \,\}$.*

Proof. The abstract language $\mathcal{A}(\mathrm{acg}(G))$ of $\mathrm{acg}(G)$ equals

$$\{\, dT \mid T \text{ is a complete derivation tree of } G \,\}\ ,$$

so it suffices to show that for every complete derivation tree T of G, we have $\mathrm{yield}_G(T) = \mathrm{graph}(|\mathscr{L}(dT)|_\beta)$. This easily follows from Lemmas 1 and 9.   □

Corollary 11. *The class of tree languages generated by second-order ACGs in $\mathbf{G}(2,4)$ includes $TR(HR)$.*

Corollary 12. *The class of tree languages generated by second-order ACGs equals $TR(HR)$.*

Corollary 13. *All tree languages generated by second-order ACGs are generated by ACGs in $\mathbf{G}(2,4)$.*

5   Conclusion

Second-order abstract categorial grammars and hyperedge replacement grammars generalize string and tree grammars with "context-free" derivations in two directions, by employing data structures that encompass strings and trees as special cases, namely linear λ-terms and hypergraphs. The present work shows two things. Since linear λ-terms can be represented by hypergraphs but not all hypergraphs can be represented by linear λ-terms, second-order ACGs are less general than hyperedge replacement grammars. However, when the generated language consists of strings or trees, both formalisms are equivalent in generating power.

Past work in formal grammar theory has shown that the class of string languages generated by second-order ACGs and hyperedge replacement grammars is extremely robust, with a large number of formalisms with equivalent power, not all of them grammars with "context-free" derivations. The class of tree languages generated by second-order ACGs and hyperedge replacement grammars may be thought of as a similarly "natural" class in the realm of tree languages. It would be interesting to find more formalisms with equivalent tree generating power.

References

1. Engelfriet, Joost. 1997. Context-free graph grammars. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages, Volume 3: Beyond Words*, pages 125–213. Berlin: Springer.
2. Engelfriet, Joost and Linda Heyker. 1989. The string generating power of context-free hypergraph grammars. *Journal of Computer and System Sciences* **43**, 328–360.

3. Engelfriet, Joost and Sebastian Maneth. 2000. Tree languages generated by context-free graph grammars. In H. Ehrig et al., editors, *Graph Transformation*, Lecture Notes in Computer Science 1764, pages 15–29. Berlin: Springer.

4. Engelfriet, J. and E. M. Schmidt. 1977. IO and OI, part I. *The Journal of Computer and System Sciences* **15**, 328–353.

5. de Groote, Philippe and Sylvain Pogodalla. 2004. On the expressive power of Abstract Categorial Grammars: Representing context-free formalisms. *Journal of Logic, Language and Information* **13**, 421–438.

6. Habel, Annegret. 1992. *Hyperedge Replacement: Grammars and Languages*. Berlin: Springer.

7. Hindley, J. Roger. 1997. *Basic Simple Type Theory*. Cambridge: Cambridge University Press.

8. Kanazawa, Makoto. 2006. Abstract families of abstract categorial languages. In G. Mints and R. de Queiroz, editors, Proceedings of the 13th Workshop on Logic, Language, Information and Computation (WoLLIC 2006). *Electronic Notes in Theoretical Computer Science* **165**, 65–80.

9. Rounds, William C. 1970. Mappings and grammars on trees. *Mathematical Systems Theory* **4**, 257–287.

10. Salvati, Sylvain. 2007. Encoding second order string ACG with deterministic tree walking transducers. In Shuly Wintner, editor, *Proceedings of FG 2006: The 11th conference on Formal Grammar*, pages 143–156. FG Online Proceedings. Stanford, CA: CSLI Publications.

11. Seki, Hiroyuki, Takashi Matsumura, Mamoru Fujii, and Tadao Kasami. 1991. On multiple context-free grammars. *Theoretical Computer Science* **88**, 191–229.

12. Sørensen, Morten Heine and Paweł Urzyczyn. 2006. *Lectures on the Curry-Howard Isomorphism*. Amsterdam: Elsevier.

13. Weir, David J. 1988. *Characterizing Mildly Context-Sensitive Grammar Formalisms*. Ph.D. dissertation. University of Pennsylvania.

14. Weir, David. 1992. Linear context-free rewriting systems and deterministic tree-walking transducers. In *Proceedings of the 30th Annual Meeting of the Association for Computational Linguistics*, pages 136–143.

On the Membership Problem for Non-linear Abstract Categorial Grammars

Sylvain Salvati [*]

INRIA-futurs, LaBRI, Université de Bordeaux
salvati@labri.fr

1 Introduction

Abstract Categorial Grammars have been introduced in [1] as a simple tool to model the interface between syntax and semantics. In those grammars, there is a distinction between the abstract level in which deep structures are represented as proofs in linear logic and the object level which accounts for the surface structures. The *lexicon* is formalized here as a homomorphism which performs the translation between those two levels. This lexicon is bound to be described by means of higher order linear λ-terms (each bound variable occurs exactly once). In this paper we are concerned with the consequences on the membership problem when the linearity constraint of lexicons is dropped. Extending ACGs in that direction is natural since linearity has many shortcomings especially when one is concerned with semantics. But also, many structures can be encoded in the simply typed λ-calculus, such as tuples, enumerated types, conditionals, lists or even relational databases [2]. All those structures can serve both in the modelisation of the syntax and of the semantics of natural languages. We show here that in the particular case of second order ACGs, the membership problem remains decidable while non-linear lexicons are used. This result is of particular importance since it means that, in general, generating texts from meaning representations is decidable when the semantics is Montague-like [3].

Our approach is similar to the one generally used in formal language theory that consists in showing the closure of the class of languages under investigation by intersection with regular sets. ACGs do not define tree or string languages but languages of λ-terms. Thus the main question we face is related to the definition of a suitable notion of regularity for the simply typed λ-calculus. The tools involved to cope with that problem are based on intersection types [4]. Even though we won't discuss the issue of regularity of the λ-calculus here, all that we do here is closely related to such a notion, and the proofs we get are quite similar to the usual syntactic proofs of closure of Context Free Languages under intersection with regular sets. They also resemble to the proofs of [5] where it is showed that Abstract Categorial Languages (languages defined by ACGs) are closed under the inverse of a relabeling.

The outline of the paper is as follows, in section 2 we introduce the necessary technical notions. In section 3, we define the main tool on which our results are based, *Higher Order Intersection Signatures*. We give the main theorems concerning this tool, in particular we present a *coherence theorem* which proves that for any M, the set $\{N | N =_{\beta\eta} M\}$ can be characterized by typing properties expressed on Higher Order Intersection Signatures. But space limitation does not allow us to give their proofs. Finally section 4 shows that the class of languages of λ-terms defined by second order non-linear ACGs is closed under typing judgements over Higher Order Intersection Signature. We also outline the reason why this kind of closure property does not hold in general for non-linear ACGs. Nevertheless, for second order non-linear ACGs, this property, together with the coherence theorem, reduces the membership problem to the emptiness problem which finally entails the expected decidability result.

2   Linear and non-linear ACGs

A *higher order signature* Σ is a triple $(\mathcal{A}, \mathcal{C}, \tau)$ such that $\mathcal{A}$ is a finite set of atomic types, $\mathcal{C}$ is a finite set of constants, and τ is a function from $\mathcal{C}$ to $\mathcal{T}_\mathcal{A}$, the set of *simple types* (or *types*). $\mathcal{T}_\mathcal{A}$ is the smallest set containing $\mathcal{A}$ and verifying the property that whenever it contains α_1 and α_2 then it contains $(\alpha_1 \to \alpha_2)$. We will use Greek lowercase letters, α, β, γ, ... to denote types, when necessary we will also use indices and superscripts. We will write $\alpha_1 \to \cdots \to \alpha_n \to \beta$ for the type $(\alpha_1 \to (\cdots \to (\alpha_n \to \beta) \cdots))$. Moreover, higher order signatures will be denoted by Σ (sometimes with subscripts), Σ_1, Σ_2, ... and, unless stated otherwise, we will assume that Σ is the triple $(\mathcal{A}, \mathcal{C}, \tau)$, Σ_{sub} is the triple $(\mathcal{A}_{sub}, \mathcal{C}_{sub}, \tau_{sub})$ with *sub* being either a subscript or an index. In general, we will use the roman lowercase letters a, b, c, d, e to denote constants. The order $ord(\alpha)$ of a type α is defined to be 1 if that type is atomic and $max(ord(\alpha_1) + 1, ord(\alpha_2))$ if $\alpha = \alpha_1 \to \alpha_2$. The notion of order is extended to higher order signatures, and $ord(\Sigma) = max\{c \in \mathcal{C} | ord(\tau(c))\}$.

We then assume that we are given an infinite and countable set $\mathcal{V}$ of λ-variables; in general, x, y and z (possibly with indices) will range over λ-variables. The λ-calculus we will use is typed *à la Church* so that terms have a unique type. In particular λ-variables (or simply variables when it is unambiguous) explicitly convey their types. Given a finite set of atomic types $\mathcal{A}$, the association of λ-variables with types is simply obtained by taking $\mathcal{V} \times \mathcal{T}_\mathcal{A}$. Given $\alpha \in \mathcal{T}_\mathcal{A}$, we write $\mathcal{V}^\alpha$ for the set $\mathcal{V} \times \{\alpha\}$, the set of λ-variables of type α. In general the elements of $\mathcal{V}^\alpha$ will be written x^α, y^α and z^α instead of (x, α), (y, α) or (z, α), we may also sometimes omit the type annotation when it is irrelevant.

The set, Λ_Σ, of λ-terms built on Σ, is the union of the sets of the family $(\Lambda_\Sigma^\alpha)_{\alpha \in \mathcal{T}_\mathcal{A}}$ which is defined as the smallest family verifying:

1. if $x \in \mathcal{V}$ and $\alpha \in \mathcal{T}_\mathcal{A}$, then $x^\alpha \in \Lambda_\Sigma^\alpha$,
2. if $c \in \mathcal{C}$ then $c \in \Lambda_\Sigma^{\tau(c)}$,
3. if $M_1 \in \Lambda_\Sigma^{\alpha \to \beta}$, $M_2 \in \Lambda_\Sigma^\alpha$ and if $x^{\gamma_1} \in FV(M_1)$ and $x^{\gamma_2} \in FV(M_2)$ implies that $\gamma_1 = \gamma_2$, then $(M_1 M_2) \in \Lambda_\Sigma^\beta$,

4. if $M \in \Lambda_{\Sigma}^{\beta}$ and if whenever $x^{\gamma} \in FV(M)$ we have $\gamma = \alpha$, then $\lambda x^{\alpha}.M \in \Lambda_{\Sigma}^{\alpha \rightarrow \beta}$.

The side conditions on the types associated to variables is responsible for the fact that free occurrences of a variable in a term all have the same type. The notation $FV(M)$ stands for the set of free variables of M and is defined as usual. We will use the letters M, N, P, Q, possibly with some indices, to denote λ-terms. Given $M_1 \in \Lambda_{\Sigma}^{\alpha_n}$, ..., $M_n \in \Lambda_{\Sigma}^{\alpha_n}$, we define $M[x_1^{\alpha_1} := M_1; \ldots; x_n^{\alpha_n} := M_n]$ to be the result of the simultaneous capture-avoiding substitution of $x_1^{\alpha_1}$ by M_1, ... and of $x_n^{\alpha_n}$ by M_n. We write $M \rightarrow_{\beta\eta} M'$ to denote that M $\beta\eta$-contracts to M', $M \xrightarrow{*}_{\beta\eta} M'$ denotes the fact that M is $\beta\eta$-reducible to M' and $M =_{\beta\eta} M'$ stands for M is $\beta\eta$-convertible to M'; we assume that the reader is familiar with these notions and that of normal form; we otherwise refer him/her to [6]. In general we will write $M_0 M_1 \ldots M_n$ instead of $(\ldots (M_1 M_2) \ldots M_n)$ and $\lambda x_1^{\alpha_1} \ldots x_n^{\alpha_n}.M$ instead of $\lambda x_1^{\alpha_1}. \ldots .\lambda x_n^{\alpha_n}.M$.

A *homomorphism between two higher order signatures* Σ_1 *and* Σ_2 is a pair (g, h) such that g maps $\mathcal{T}_{\mathcal{A}_1}$ to $\mathcal{T}_{\mathcal{A}_2}$, h maps Λ_{Σ_1} to Λ_{Σ_2} and the following holds:

1. $g(\alpha \rightarrow \beta) = g(\alpha) \rightarrow g(\beta)$
2. $h(x^{\alpha}) = x^{g(\alpha)}$
3. for $c \in \mathcal{C}_1$, $h(c)$ is a closed element (*i.e.* $FV(h(c)) = \varnothing$) of $\Lambda_{\Sigma_1}^{\tau_1(c)}$
4. $h(M_1 M_2) = h(M_1)h(M_2)$
5. $h(\lambda x^{\alpha}.M) = \lambda x^{g(\alpha)}.h(M)$.

It is easy to establish that whenever $M \in \Lambda_{\Sigma_1}^{\alpha}$ then $h(M) \in \Lambda_{\Sigma_2}^{g(\alpha)}$. In general, if $\mathcal{H} = (g, h)$ is a homomorphism, we will write $\mathcal{H}(\alpha)$ for $g(\alpha)$ and $\mathcal{H}(M)$ for $h(M)$.

Non-linear[1] Abstract Categorial Grammars are 4-tuples $\mathcal{G} = (\Sigma_1, \Sigma_2, \mathcal{L}, S)$ where:

1. Σ_1 is a higher order signature, *the abstract vocabulary,*
2. Σ_2 is a higher order signature, *the object vocabulary,*
3. $\mathcal{L}$ is a homomorphism, *the lexicon* and,
4. S is an element of $\mathcal{A}_1$, *the distinguished type.*

Note that contrary to the usual definition we do not require that Σ_1 or Σ_2 are linear signature or the lexicon to be linear. The order of a non-linear ACG is the order of its abstract vocabulary. A non-linear ACG $\mathcal{G} = (\Sigma_1, \Sigma_2, \mathcal{L}, S)$ defines two languages:

- *The abstract language:* $\mathcal{A}(\mathcal{G}) = \{M \in \Lambda_{\Sigma_1}^{S} | M \text{ is closed}\}$,
- *The object language:* $\mathcal{O}(\mathcal{G}) = \{M \in \Lambda_{\Sigma_2}^{\mathcal{L}(S)} | \exists N \in \mathcal{A}(\mathcal{G}).\mathcal{L}(N) =_{\beta\eta} M\}$

---

[1] as opposed to the usual ACGs.

3   Higher order intersection signatures

In this section we define *higher order intersection signatures* (HOIS). This new
kind of signatures is used as a second layer of typing over simply typed λ-calculus.
The reader must be careful not to get confused between the simple type a λ-
term may have and the intersection types higher order intersection signatures
may give to it. HOIS adapt intersection types [4], usually dedicated to the un-
typed λ-calculus, to the simply typed λ-calculus. The traditional notation for the
intersection types is $(\alpha \cap \beta)$, but it is provable that the $\cap$ connector enjoys prop-
erties like idempotence, associativity, commutativity and has a neutral element
ω.[2] Since, for technical reasons, we want to use types modulo these equivalence
relations, we will use a different notation for intersection types. The intersection
of several types will be denoted by the set containing those types. Note that we
will write $\mathcal{P}(A)$ the powerset of a set A.

Given a higher order signature Σ, a finite set I and a function ρ from I to $\mathcal{A}$,
we define the family of intersection types $\cap_\rho = (\cap_\rho^\alpha)_{\alpha \in \mathcal{T}_\mathcal{A}}$ as the smallest family
verifying:

1. $\cap_\rho^\alpha = \rho^{-1}(\alpha)$ if $\alpha \in A$
2. $\cap_\rho^{\alpha \to \beta} = \mathcal{P}(\cap_\rho^\alpha) \times \{\alpha\} \times \cap_\rho^\beta$

Intuitively the intersection type $(S, \alpha, p) \in \cap_\rho^{\alpha \to \beta}$ would be written in the
usual syntax as $p_1 \cap \ldots \cap p_n \to p$ when $S \neq \varnothing$ and $S = \{p_1; \ldots; p_n\}$ or $\omega_\alpha \to p$[3]
when $S = \varnothing$. The fact that we use $\mathcal{P}(\cap_\rho^\alpha) \times \{\alpha\} \times \cap_\rho^\beta$ instead of $\mathcal{P}(\cap_\rho^\alpha) \times \cap_\rho^\beta$ in
the definition $\cap_\rho^{\alpha \to \beta}$ has the consequence that if p belongs to a member of the
family $\cap_\rho$ then it does not belong to any other member of $\cap_\rho$ (*i.e.* there is a
unique α such that $p \in \cap_\rho^\alpha$). Thus an intersection type p that belong to $\cap_\rho^\alpha$ is,
as one may expect, only used to type the terms of Λ_Σ^α.

Note that given $\alpha \in \mathcal{A}$, we have $|\cap_\rho^\alpha| = \mathcal{O}(|I|)$ and given $\alpha \to \beta \in \mathcal{T}_\mathcal{A}$ we
have $|\cap_\rho^{\alpha \to \beta}| = 2^{|\cap_\rho^\alpha|} \times |\cap_\rho^\beta|$. Thus $\cap_\rho^\alpha$ is finite for every $\alpha \in \mathcal{T}_\mathcal{A}$.

Then, a *higher order intersection signature* Π is a 4-tuple (Σ, I, ρ, f) where:

1. Σ is a higher order signature,
2. I is a finite set of atomic types.
3. ρ is a mapping from I to $\mathcal{A}$,
4. f associates to any $c \in C$ a subset of $\cap_\rho^{\tau(c)}$.

Such a higher order intersection signature is also called an *intersection sig-
nature over* Σ. In general we shall write $\cap_\Pi^\alpha$ instead of $\cap_\rho^\alpha$ and $\cap_\Pi$ instead of $\cap_\rho$
for a HOIS $\Pi = (\Sigma, I, \rho, f)$. A *type basis* or a *type environment* is a sequence
$x_1^{\alpha_1} : S_1, \ldots, x_n^{\alpha_n} : S_n$ where for all i, $S_i \subseteq \cap_\Pi^{\alpha_i}$ and the variables $x_i^{\alpha_i}$ are pairwise
distinct.

---

[2] ω stands for the universal type, *i.e.* every term can be typed by ω, and thus ω
   represents the set of all λ-terms.

[3] intuitively ω_α stands for the simply typed version of ω and it is the universal inter-
   section type of the terms that belong to Λ_Σ^α.

Given an intersection signature $\Pi = (\Sigma, I, \rho, f)$, we define derivations which establish judgements of the form: $\Gamma \vdash_\Pi M : p$ where Γ is a type environment, $M \in \Lambda_\Sigma^\alpha$ and $p \in \cap_\Pi^\alpha$. The judgments are obtained with the following deduction rules:

$$\frac{p \in S}{\Gamma, x^\alpha : S \vdash_\Pi x^\alpha : p} \text{ Axiom} \qquad \frac{p \in f(c)}{\Gamma \vdash_\Pi c : p} \text{ Const} \qquad \frac{\Gamma, x^\alpha : S \vdash_\Pi M : p}{\Gamma \vdash_\Pi \lambda x^\alpha.M : (S, \alpha, p)} \text{ Abstraction}$$

$$\frac{\Gamma \vdash_\Pi M : (S, \alpha, p) \quad N \in \Lambda_\Sigma^\alpha \quad \forall q \in S.\ \Gamma \vdash_\Pi N : q}{\Gamma \vdash_\Pi MN : p} \text{ App}$$

The properties of terms which are typed in an intersection signature are quite similar to the properties one usually obtains with intersection types. The most important one is that judgement are invariant modulo β-convertibility.

Theorem 1. *If $\Gamma \vdash_\Pi M : p$ is derivable and $M \xrightarrow{*}_\beta N$ then $\Gamma \vdash N : p$ is also derivable.*

Theorem 2. *If $M \in \Lambda_\Sigma$, $M \xrightarrow{*}_{\beta\eta} N$ and $\Gamma \vdash_\Pi N : p$ is derivable then $\Gamma \vdash_\Pi M : p$ is also derivable.*

Note that this last theorem only holds when $M \in \Lambda_\Sigma$, that is when M is simply typed.

Actually, derivability is not preserved under η-contraction. In order to obtain it we need define the following subtyping relation on intersection types:

$$\frac{\iota \in \rho^{-1}(\alpha)}{\iota \sqsubseteq^\alpha \iota} \qquad \frac{T \subseteq \cap_\Pi^\alpha \quad \forall p \in S.\exists q \in T.q \sqsubseteq^\alpha p}{T \sqsubseteq^{\mathcal{P}_\alpha} S} \qquad \frac{S \sqsubseteq^{\mathcal{P}_\alpha} T \quad q \sqsubseteq^\beta p}{(T, \alpha, q) \sqsubseteq^{\alpha \to \beta} (S, \alpha, p)}$$

We may add the following typing rule: $\dfrac{\Gamma \vdash_\Pi M : p \quad p \sqsubseteq^\alpha q}{\Gamma \vdash_\Pi M : q}$. When a derivation uses this subtyping rule, we write the judgement $\Gamma \vdash_\Pi^\sqsubseteq M : p$. Actually on terms in long form any derivation established by $\vdash_\Pi^\sqsubseteq$ can also be derived by $\vdash_\Pi$, as stipulated by the following theorem.

Theorem 3. *If M is in long form, $\Gamma \vdash_\Pi M : p$ is derivable and $p \sqsubseteq^\alpha q$ then $\Gamma \vdash_\Pi M : q$ is derivable.*

So the judgments of the derivation system $\vdash_\Pi^\sqsubseteq$ are closed under $\beta\eta$-convertibility when the subjects are elements of Λ_Σ, but derivability in $\vdash_\Pi^\sqsubseteq$ is equivalent to derivability in $\vdash_\Pi$ on long forms. This gives us a nice tool to avoid the subsumption rule in proofs.

HOIS may express very strong properties about the terms. In particular the following theorem shows that they can characterize $\beta\eta$-convertible classes of λ-terms.

Theorem 4. *(Coherence) Given $M \in \Lambda_\Sigma$ there is a HOIS Π over Σ, a basis Γ and $p \in \cap_\Pi$ such that for every $N \in \Lambda_\Sigma$, $\Gamma \vdash_\Pi^\sqsubseteq N : p$ if and only if $M =_{\beta\eta} N$.*

For example, $M = \lambda f^{o \to o} x^o . \underbrace{f^{o \to o}(\ldots (f^{o \to o} x^o) \ldots)}_{n \times}$, the church numeral n, is the only λ-term, modulo $\beta\eta$-convertibility, that can be typed with (written in an intuitive notation) $([0] \to [1] \cap \ldots \cap [n-1] \to [n]) \to [0] \to [n]$.

This expressivity has actually a cost since HOIS can also encode λ-definability which is undecidable [7]. This shows that it is in general undecidable to know whether there is a term M such that $\Gamma \vdash_\Pi M : p$ is derivable.

4 Decidability of the membership problem for second order ACGs

To obtain the decidability result we aim at, we start by showing a closure property of the languages of λ-terms defined by second order non-linear ACGs. We indeed prove that the class of languages defined by second order non-linear ACGs is closed under intersection with the set of terms typable on a higher order intersection signature with all the types of a given set P. Then the use of the coherence theorem reduces the membership problem to the problem of the emptiness of the language of a second order ACG, which is trivially decidable (it amounts to checking whether a local tree language is empty or not).

Given Π a higher order intersection signature, $T_1 \subseteq \cap_\Pi^\alpha$ and $T_2 \subseteq \cap_\Pi^\beta$ we define $T_1 \to_\alpha T_2$ to be the set $\{(T_1, \alpha, p) | p \in T_2\}$. When taking $T_1 \subseteq \cap_\Pi^\alpha$ and $T_2 \subseteq \cap_\Pi^\beta$, it is obvious that $T_1 \to_\alpha T_2 \subseteq \cap_\Pi^{\alpha \to \beta}$. Furthermore, given, $T \subseteq \cap_\Pi^\alpha$ we will write $\Gamma \vdash_\Pi^\sqsubseteq M : T$ (resp. $\Gamma \vdash_\Pi M : T$) to denote the fact that for all $p \in T$, $\Gamma \vdash_\Pi^\sqsubseteq M : p$ (resp. $\Gamma \vdash_\Pi M : p$) is derivable.

Theorem 5. *Given a second order non-linear ACG $\mathcal{G} = (\Sigma_0, \Sigma_1, \mathcal{L}, S)$, an intersection signature $\Pi = (\Sigma_1, I, \rho, f)$ over and $P \subseteq \cap_\Pi^{\mathcal{L}(S)}$, there is a second order non-linear ACG $\mathcal{G}'$ such that:*

$$\mathcal{O}(\mathcal{G}') = \{M \mid M \in \mathcal{O}(\mathcal{G}) \land \forall p \in P. \ \vdash_\Pi^\sqsubseteq M : p\}$$

Proof. We define $\mathcal{G}' = (\Sigma_0', \Sigma_1, \mathcal{L}', \langle S, P \rangle)$ following way:

1. $A_0' = \{\langle \alpha, Q \rangle | \alpha \in A_0 \land Q \subseteq \cap_\Pi^{\mathcal{L}(\alpha)}\}$,
2. $C_0' = \{\langle c, \langle S_1, \ldots, S_n \rangle, S_0 \rangle | \tau_0(c) = \alpha_1 \to \cdots \to \alpha_n \to \alpha_0 \land$
$\forall i \in [0, n]. \ S_i \subseteq \cap_\Pi^{\mathcal{L}(\alpha_i)} \land$
$\vdash_\Pi^\sqsubseteq \mathcal{L}(c) : S_1 \to_{\mathcal{L}(\alpha_1)} \cdots \to_{\mathcal{L}(\alpha_{n-1})} S_n \to_{\mathcal{L}(\alpha_n)} S_0\}$,
3. $\tau_0'(\langle c, \langle S_1, \ldots, S_n \rangle, S_0 \rangle) = \langle \alpha_1, S_1 \rangle \to \cdots \to \langle \alpha_n, S_n \rangle \to \langle \alpha_0, S_0 \rangle$ when $\tau_0(c) = \alpha_1 \to \cdots \to \alpha_n \to \alpha_0$,
4. $\mathcal{L}'(\langle \alpha, Q \rangle) = \mathcal{L}(\alpha)$,
5. $\mathcal{L}'(\langle c, \langle S_1, \ldots, S_n \rangle, S_0 \rangle) = \mathcal{L}(c)$.

We claim that given $M \in \Lambda_{\Sigma_1}^{\mathcal{L}(\alpha)}$ and $Q \subseteq \cap_\Pi^{\mathcal{L}(\alpha)}$, the two following properties are equivalent:

1. there is a closed term $N \in \Lambda_{\Sigma_0}^{\alpha}$ such that $\mathcal{L}(N) =_{\beta\eta} M$, and $\vdash_{\overline{\Pi}}^{\sqsubseteq} M : Q$ is derivable,
2. there is $N' \in \Lambda_{\Sigma_0'}^{\langle \alpha, Q \rangle}$ such that $\mathcal{L}(N') =_{\beta\eta} M$.

The first direction of the equivalence can be proved by induction on the structure of N and by using simple properties of the derivations of $\vdash_{\overline{\Pi}}^{\sqsubseteq} \mathcal{L}(N) : Q$ (we can do that since typing judgements are closed under $\beta\eta$-convertibility, $\mathcal{L}(N) =_{\beta\eta} M$, and $\vdash_{\overline{\Pi}}^{\sqsubseteq} M : Q$). The second direction of the equivalence is an easy induction on the structure of N'.

Note that the construction of $\mathcal{G}'$ is effective.

Theorem 6. *Given $\mathcal{G} = (\Sigma_1, \Sigma_2, \mathcal{L}, S)$ a second order non-linear ACG, and M, it is decidable whether $M \in \mathcal{O}(\mathcal{G})$.*

Proof. The coherence theorem gives the existence of Π and p such that for all N, the derivability of $\vdash_{\overline{\Pi}}^{\sqsubseteq} N : p$ implies that $M =_{\beta\eta} N$. The previous theorem allows us to effectively build an ACG $\mathcal{G}'$ such that $\mathcal{O}(\mathcal{G}') = \{N | N \in \mathcal{O}(\mathcal{G}) \wedge \vdash_{\overline{\Pi}}^{\sqsubseteq} N : p\}$. Thus $\mathcal{O}(\mathcal{G}')$ is either empty if $M \notin \mathcal{O}(\mathcal{G})$ or it contains, modulo $\beta\eta$-conversion, one element which is no other that M if $M \in \mathcal{O}(\mathcal{G})$. Thus checking the emptiness of $\mathcal{G}'$ amounts to check $M \notin \mathcal{O}(\mathcal{G})$ which shows that our decidability result.

If we take an arbitrary non-linear ACG, then the closure theorem does not hold. Indeed, the emptiness of an arbitrary ACG is decidable since minimal logic is decidable. But we saw that that intersection types could code for λ-definability. It is easy to devise a higher order ACG such that the set of terms of its object language that are typable with a certain set of intersection types are precisely the terms which λ-define some point in a full model. Thus with [7] we get that the emptiness of this language is undecidable so that it cannot be the language of a higher order ACG whose emptiness is decidable. By having constants in the abstract signature that code for deletion and duplication, it is easy to show that even if the abstract language is taken as linear, then one can define sets of λ-terms that λ-define some point in a full model by intersecting the language with the property of being typable with a set of intersection types. Then the reasoning we have held to show that non-linear ACGs were not closed under intersection with these kind of properties may not apply since we do not know whether the emptiness of ACGs with a linear language is in general decidable or not. But we conjecture that the languages are also not closed under these intersection properties. We also conjecture that in general the membership problem is undecidable for those grammars.

References

1. de Groote, P.: Towards abstract categorial grammars. In for Computational Linguistic, A., ed.: Proceedings 39th Annual Meeting and 10th Conference of the European Chapter, Morgan Kaufmann Publishers (2001) 148–155

2. Hillebrand, G.G.: Finite Model Theory in the Simply Typed Lambda Calculus. PhD thesis, Department of Computer Science, Brown University, Providence, Rhode Island 02912 (May 1994)
3. Montague, R.: Formal Philosophy: Selected Papers of Richard Montague. Yale University Press, New Haven, CT (1974)
4. Dezani-Ciancaglini, M., Giovannetti, E., de' Liguoro, U.: Intersection Types, Lambda-models and Böhm Trees. In: MSJ-Memoir Vol. 2 "Theories of Types and Proofs". Volume 2. Mathematical Society of Japan (1998) 45–97
5. Kanazawa, M.: Abstract families of abstract categorial languages. Electr. Notes Theor. Comput. Sci. **165** (2006) 65–80
6. Barendregt, H.P.: The Lambda Calculus: Its Syntax and Semantics. Volume 103. Studies in Logic and the Foundations of Mathematics, North-Holland Amsterdam (1984) revised edition.
7. Loader, R.: The undecidability of λ-definability. In Anderson, C.A., Zeleny, M., eds.: Logic, Meaning and Computation: Essays in memory of Alonzo Church. Kluwer (2001) 331–342

Non-Associative Categorial Grammars and Abstract Categorial Grammars

Christian Retoré, Sylvain Salvati

INRIA-futurs, LaBRI, Université de Bordeaux
{retore, salvati}@labri.fr

Abstract. We first show here that the parse structures (the normal natural deduction trees) of a non-associative Lambek grammar can be encoded in second order signatures and then decoded into lambda-terms representing the derivations by means of a linear homomorphism. This result allows one to represent within the unifying framework of Abstract Categorial Grammars the usual interface between syntax and semantics that advocates in favor of categorial formalisms.

1 Introduction

Abstract Categorial Grammars (ACGs) [1] have been defined as a simplification of the usual categorial formalisms. The spirit of ACGs is to minimize the number of primitives that are involved in their definition. The λ-calculus is used as a means to represent all the necessary objects such as proof derivations, trees and strings; while homomorphisms implement constraints such as dominance in trees or word order in strings. The expressivity of ACGs has already been explored in [2], but one may wonder whether one can *represent faithfully* any categorial formalism. By *represent faithfully* we mean that not only the string language is recognized, but also that the proof structures that allow the elegant and well-known interface between syntax and semantics of categorial formalisms can be represented. Another interest of such a representation is that it allows one to import into ACGs the analyses of grammatical phenomena implemented in categorial grammars. We investigate this question by showing how to encode the usual architecture of the non-associative Lambek calculus [3] in Abstract Categorial Grammars. The coding is presented in two steps. Due to space limitations, we will concentrate here on the more technical first step and only outline the second one. This first step consists in representing the natural deduction trees corresponding to grammatical analyses in an NL grammar as terms built on a second order signature and to map those terms to the λ-terms that represent those derivations by means of a linear homomorphism. This suffices to represent the grammatical analyses of an NL-grammar into an ACG. The second step which consists in building within ACGs the interface between syntax and semantics of NL-grammars is only outlined. It consists in mapping the λ-terms obtained form the first step to the analyzed sentence and to some meaning representation.

The paper is organised as follows. We first define the simply typed λ-calculus and Abstract Categorial Grammars in section 2. In section 3, we present the non-associative Lambek calculus and NL-grammars. And finally section 4 describes the embedding we propose.

2 λ-calculus and Abstract Categorial Grammars

Higher order signatures, denoted by Σ or Σ_{sub} (*sub* being either an index or a subscript), are triples $(\mathcal{A}, \mathcal{C}, \tau)$ (unless stated otherwise, we assume that we have $\Sigma = (\mathcal{A}, \mathcal{C}, \tau)$ and $\Sigma_{sub} = (\mathcal{A}_{sub}, \mathcal{C}_{sub}, \tau_{sub})$) where $\mathcal{A}$ is a finite set of *atomic types*, $\mathcal{C}$ is a finite set of *constants* and τ is a *typing function*, *i.e.* a function from $\mathcal{C}$ to $\mathcal{T}_\mathcal{A}$ where $\mathcal{T}_\mathcal{A}$ is the set of *types* built from $\mathcal{A}$ and by using the binary infix operator $\to$. We assume that $\to$ associates to the right and therefore that $\alpha_1 \to \cdots \alpha_n \to \beta$ stands for type $(\alpha_1 \to \cdots (\alpha_n \to \beta) \cdots)$. The set, Λ_Σ, of λ-terms built on the higher order signature Σ is the union of the members of the family $(\Lambda_\Sigma^\alpha)_{\alpha \in \mathcal{T}_\mathcal{A}}$ which is the smallest family satisfying:

1. $x^\alpha \in \Lambda_\Sigma^\alpha$ (x^α is a λ-*variable*),
2. $c \in \Lambda_\Sigma^{\tau(c)}$,
3. $M_1 \in \Lambda_\Sigma^{\beta \to \alpha}$ and $M_2 \in \Lambda_\Sigma^\beta$ imply that $(M_1 M_2) \in \Lambda_\Sigma^\alpha$, and
4. $\lambda x^\beta . M \in \Lambda_\Sigma^{\beta \to \alpha}$ whenever $M \in \Lambda_\Sigma^\beta$.

As usual we write $\lambda x_1^{\alpha_1} \ldots x_n^{\alpha_n} . M$ for $\lambda x_1^{\alpha_1} . \ldots . \lambda x_n^{\alpha_n} . M$ and $M_0 M_1 \ldots M_n$ for $(\ldots (M_0 M_1) \ldots)$. We take for granted that the notion of free variables ($FV(M)$ denotes the set of free variables of M), α-conversion, β-conversion, normal form, long form, head of a term are known (see [4] and [5]).

A term M is said to be *linear* if $M = x^\alpha$; if $M = c$; if $M = M_1 M_2$, M_1 and M_2 are linear, and $FV(M_1) \cap FV(M_2) = \varnothing$; or if $M = \lambda x^\beta . M'$, M' is linear and $x^\beta \in FV(M')$.

A *homomorphism* between the signatures Σ_1 and Σ_2 is a pair (g, h) such that g maps $\mathcal{T}_{\mathcal{A}_1}$ to $\mathcal{T}_{\mathcal{A}_2}$, h maps Λ_{Σ_1} to Λ_{Σ_2} and verify the following properties:

1. $g(\alpha \to \beta) = g(\alpha) \to g(\beta)$,
2. $h(x^\alpha) = x^{g(\alpha)}$,
3. $h(c)$ is a closed term (*i.e.* $FV(h(c)) = \varnothing$) of $\Lambda_{\Sigma_2}^{g(\tau(c))}$,
4. $h(M_1 M_2) = h(M_1) h(M_2)$ and
5. $h(\lambda x^\beta . M) = \lambda x^{g(\beta)} . h(M)$.

A homomorphism is said *linear* if it associates closed linear terms to constants. We write $\mathcal{H}(\alpha)$ and $\mathcal{H}(M)$ respectively instead of $g(\alpha)$ and of $h(M)$ for a given homomorphism $\mathcal{H} = (g, h)$. Note that if $\mathcal{H}$ is a homomorphism from Σ_1 to Σ_2 and $M \in \Lambda_{\Sigma_1}^\alpha$ then $\mathcal{H}(M) \in \Lambda_{\Sigma_2}^{\mathcal{H}(\alpha)}$, note furthermore that if $\mathcal{H}$ and M are both linear then so is $\mathcal{H}(M)$.

An *Abstract Categorial Grammar* [1] (ACG) is a 4-tuple $(\Sigma_1, \Sigma_2, \mathcal{L}, S)$ where Σ_1 is the *abstract vocabulary*, Σ_2 is the *object vocabulary*, $\mathcal{L}$ is linear homomorphism, the *lexicon*, and S is an element of $\mathcal{A}_1$, the *distinguished type*.

A *non-linear Abstract Categorial Grammar* is an ACG whose lexicon may be an arbitrary homomorphism. An ACG $\mathcal{G} = (\Sigma_1, \Sigma_2, \mathcal{L}, S)$ (*resp.* a non-linear ACG) defines two languages: the *abstract language*: $\mathcal{A}(\mathcal{G}) = \{M \in \Lambda^S_{\Sigma_1} | M$ is closed and linear$\}$, the *object language*: $\mathcal{O}(\mathcal{G}) = \{M | \exists N \in \mathcal{A}(\mathcal{G}) \wedge M$ is the long normal form of $\mathcal{L}(N)\}$.

Note that given a homomorphism $\mathcal{L}'$ from Σ_2 to Σ_3, and an ACG $(\Sigma_1, \Sigma_2, \mathcal{L}, S)$, then $(\Sigma_1, \Sigma_3, \mathcal{L}' \circ \mathcal{L}, S)$ is an ACG when $\mathcal{L}'$ is linear and otherwise it is a non-linear ACG.

3   The non-associative Lambek calculus

In this paper we deal with the non-associative Lambek calculus without product known as NL [3]. Given a finite set Cat, called the *basic set of categories*, the set of categories built on Cat, NL_{Cat}, is the smallest set containing Cat and having the property that if $A, B \in NL_{Cat}$ then $\backslash(B, A)$ and $/(B, A)$ are in NL_{Cat}.[1] Categories will be represented by the roman uppercase letters A, B, C, D, E and F (possibly with some indices). A *hypothesis base*, or simply a *base*, is a binary tree whose leaves are elements of NL_{Cat}; any element of NL_{Cat} can be considered as a base, and given two bases Γ and Δ, we write (Γ, Δ) the new base obtained from them. Given a base Γ, we write $\overline{\Gamma}$ the list of the leaves of Γ taken from left to right.

The non-associative Lambek calculus derives judgements of the form $\Gamma \vdash A$ where Γ is a base and A belongs to NL_{Cat}. These judgements are obtained with the following rules:

$$\frac{}{A \vdash A} \, Ax. \qquad \frac{(\Gamma, B) \vdash A}{\Gamma \vdash /(B, A)} \, / \, I \qquad \frac{(B, \Gamma) \vdash A}{\Gamma \vdash \backslash(B, A)} \, \backslash \, I$$

$$\frac{\Gamma \vdash \backslash(B, A) \quad \Delta \vdash B}{(\Delta, \Gamma) \vdash A} \, / \, E \qquad \frac{\Gamma \vdash /(B, A) \quad \Delta \vdash B}{(\Gamma, \Delta) \vdash A} \, \backslash \, E$$

If we define the functions $f_\backslash$ and $f_/$ as binary operators over contexts such that $f_\backslash(\Gamma, \Delta) = (\Delta, \Gamma)$ and $f_/(\Gamma, \Delta) = (\Gamma, \Delta)$ we then may write the introduction and the elimination rules as rules parametrised by the operator $op \in \{\backslash; /\}$:

$$\frac{f_{op}(\Gamma, B) \vdash A}{\Gamma \vdash op(B, A)} \, op \, I \qquad \frac{\Gamma \vdash op(B, A) \quad \Delta \vdash B}{f_{op}(\Gamma, \Delta) \vdash A} \, op \, E$$

This remark will simplify the notation of derivations in the following of the paper. We will also write $\widetilde{op}$ for $\backslash$ if $op = /$ and for $/$ when $op = \backslash$.

A non-associative Lambek grammar (NL-grammar) is a 4-tuple $G_{NL} = (W, Cat, \chi, S)$ where W is a set of *words*, Cat is a set of *atomic categories*, χ is

---

[1] In the literature $\backslash(B, A)$ is rather written as $B\backslash A$ and $/(B, A)$, as A/B. We adopt these non-conventional notations so as to facilitate the encoding of NL-grammars in ACGs.

a function from W to finite subsets of NL_{Cat}, and $S \in Cat$. In order to account for grammaticality, we now allow a new kind of hypothesis in hypothesis bases. These new hypotheses are pairs $\langle w, A \rangle$ such that $A \in \chi(w)$. They represent the use of a *lexical entry* of the NL-grammar in a derivation. A base is said to be *lexical* if all its leaves are such pairs. We also add the following rule:

$$\frac{A \in \chi(w)}{\langle w, A \rangle \vdash A} \, Lex$$

The only distinction between this new kind of hypothesis and the usual ones is that they cannot be discharged by using the rules $\backslash \, I$ and $/ \, I$.

A sentence $w_1 \ldots w_n$ of W^* is said to be *accepted* by G_{NL} if there is a lexical base Γ such that $\overline{\Gamma} = [\langle w_1, A_1 \rangle; \ldots; \langle w_n, A_n \rangle]$ and $\Gamma \vdash S$ is derivable. Such a derivation is called a *grammatical analysis* of $w_1 \ldots w_n$. The language defined by G_{NL} is the set of sentences in W^* that it accepts.

The Curry-Howard correspondence associates λ-terms to derivations. This correspondence allows one to interpret the notion of reduction defined for natural deduction systems as β-reduction. The following system shows how to transform a derivation in NL into a linear λ-term built on the signature $\Sigma_{G_{NL}}$ with $\mathcal{A}_{G_{NL}} = Cat$, $\mathcal{C}_{G_{NL}} = \{\langle w, A \rangle | w \in W \wedge A \in \chi(w)\}$ and $\tau_{G_{NL}}(\langle w, A \rangle) = A^\dagger$ where $op(A_1, A_2)^\dagger = A_1^\dagger \rightarrow A_2^\dagger$.

$$\frac{A \in \chi(w)}{\langle w, A \rangle \vdash \langle w, A \rangle : A} \qquad \frac{}{x^{A^\dagger} : A \vdash x^{A^\dagger} : A} \, Ax.$$

$$\frac{f_{op}(\Gamma, x^{B^\dagger} : B) \vdash M : A}{\Gamma \vdash \lambda x^{B^\dagger}.M : op(B, A)} \, op \, I \quad \frac{\Gamma \vdash M_1 : op(B, A) \; \Delta \vdash M_2 : B}{f_{op}(\Gamma, \Delta) \vdash (M_1 M_2) : A} \, op \, E$$

This correspondence allows us to talk about derivations in normal form, derivations in long forms, and also about the head of a derivation as induced from the λ-calculus. In particular the derivations that are in normal form enjoy the so-called subformula property; for grammatical derivation, this property implies that the formulae used in the grammatical analyses of G_{NL} can only be S or some subformula of $A \in \bigcup_{w \in W} \chi(w)$. We adopt the notation $\mathcal{F}_{G_{NL}}$ for this set of formulae for a NL-grammar G_{NL}.

One can devise a lexicon $\mathcal{Y}$ which maps every atomic category to the type $* \rightarrow *$ so that every term M denoting a grammatical analyses of $w_1 \ldots w_n$ is mapped by $\mathcal{Y}$ to the term $\lambda x^*.w_1(\ldots (w_n \, x^*) \ldots)$ which represents the sentence $w_1 \ldots w_n$. Furthermore, the meaning representation of the sentence, for Montague-like semantics, can be obtained from those λ-terms, by means of a non-linear lexicon $\mathcal{L}_{Sem}$.

4   Coding NL-grammars into ACGs

In this section we define an ACG $\mathcal{G}_{G_{NL}} = (\Sigma_{Der \, G_{NL}}, \Sigma_{G_{NL}}, \mathcal{L}_{G_{NL}}, [\bullet \vdash S])$ ($\Sigma_{G_{NL}}$ is as defined in the previous section) so that $\mathcal{O}(\mathcal{G}_{G_{NL}})$ contains exactly

the λ-terms in long normal form that are associated to the grammatical analyses of some given NL-grammar, $G_{NL} = (W, Cat, \chi, S)$. $\Sigma_{Der\,G_{NL}}$ is a second order signature (*i.e.* every constants have types of the form $\alpha_1 \to \cdots \to \alpha_n \to \alpha_0$ where for all $i \in [0, n]$, α_i is an atomic type) whose terms encode derivations in NL. The role of the lexicon is to decode these terms into λ-terms representing the derivation.

In doing so, we try to reveal intrinsic properties of the grammatical analyses of NL-grammars. By definition, these analyses are derivations of the form $\Gamma \vdash S$ where Γ is a lexical base. We therefore investigate what may be the shape of a long normal derivation of $\Gamma \vdash A$ when Γ is lexical and $A \in \mathcal{F}_{S_{NL}}$ (recall that $\mathcal{F}_{S_{NL}}$ is the set of formulae that can occur in the grammatical analyses of G_{NL}). Such a derivation is described by figure 1.

$$
\begin{array}{c}
\dfrac{\dfrac{\vdots}{f_{op_1}(\Gamma, A_1) \vdash D_2 = op_2(C_2, D_3)} \quad \dfrac{\vdots}{A_2 \vdash C_2}}{f_{op_2}(f_{op_1}(\Gamma, A_1), A_2) \vdash D_3} \; op_2\ E
\end{array}
$$

$$
\dfrac{\dfrac{\dfrac{\vdots}{\Gamma_{n-1} \vdash op_n(C_n, B)} \quad \dfrac{\vdots}{A_n \vdash C_n}}{f_{op_n}(\Gamma_{n-1}, A_n) \vdash B} \; op_n\ E}{\quad} \; op_n\ I
$$

$$
\dfrac{\vdots}{f_{op_1}(\Gamma, A_1) \vdash B_2} \; op_2\ I
$$

$$
\dfrac{}{\Gamma \vdash op_1(A_1, B_2) = B_1 = A} \; op_1\ I
$$

The notations are as follows: $A = op_1(A_1, \ldots op_n(A_n, B) \ldots)$, $B_n = D_n = op_n(A_n, B)$, $B_k = op_k(A_k, B_{k+1})$, $D_k = op_k(C_k, D_{k+1})$, $\Gamma_1 = \Gamma$, and $\Gamma_{k+1} = f_{op_k}(\Gamma_k, A_k)$

Fig. 1. Shape of the long normal derivation of $\Gamma \vdash A$

We note that, since the whole derivation is in normal form, the derivation of $f_{op_1}(\Gamma, A_1) \vdash D_2$ must end with a sequence of rules of the form $op\ E$. There are two possibilities, either the head is a lexical hypothesis and we have that it has the following shape, with the convention that $F_p = D_1$ and $F_i = op'_i(E_i, F_{i+1})$, $\Delta_1 = \langle w, F_1 \rangle$, $\Delta_{i+1} = f_{op'_i}(\Delta_i, \Theta_i)$ and the constraint that $\Gamma = \Delta_p$:

$$\cfrac{\cfrac{\cfrac{B_1 \in \chi(w)}{\langle w, F_1\rangle \vdash F_1 = op'_1(E_1, F_2)}\ Lex \qquad \cfrac{\vdots}{\Theta_1 \vdash E_1}}{f_{op'_1}(\langle w, F_1\rangle, \Theta_1) = \Delta_2 \vdash F_2}\ op'_1\ E}{\cfrac{\vdots}{\varGamma \vdash D_1 = op_1(C_1, D_2)}\ op'_p\ E \qquad \cfrac{\vdots}{A_1 \vdash C_1}}{f_{op_1}(\varGamma, A_1) \vdash D_2}\ op_1\ E$$

Note that the bases Θ_i must be lexical since $\varGamma$ is lexical.

In case the head of the derivation of $f_{op_1}(\varGamma, A_1) \vdash D_2$ is the hypothesis A_1, we have that this derivation has the following shape:

$$\cfrac{\cfrac{}{A_1 \vdash \overline{op_1}(C_1, D_2)}\ Ax. \qquad \cfrac{\vdots}{\varGamma \vdash C_1}}{f_{\overline{op_1}}(A_1, \varGamma) = f_{op_1}(\varGamma, A_1) \vdash D_2}\ \overline{op_1}\ E$$

Note that the two possible derivations in long normal form of $\varGamma \vdash A$ where $\varGamma$ is a lexical base we have depicted, require only to be able to construct two kinds of derivations: either derivations of sequents of the same form as $\varGamma \vdash A$ or derivations of sequents like $C \vdash A$. Thus, to fulfill our objective of encoding grammatical analyses within a second order signature, we also need to analyse the shape of long normal derivations of sequents of the form $C \vdash A$. Again the general shape of such a derivation is given by figure 1, but in that case we have $\varGamma = C$. Here, there are two possibilities for the derivation of $f_{op_1}(C, A_1) \vdash D_2$: either the head of the derivation is the hypothesis C or it is the hypothesis A_1. If the head is C then we get the following derivation for $f_{op_1}(C, A_1) \vdash D_2$:

$$\cfrac{\cfrac{}{C \vdash C = D_1 = op_1(C_1, D_2)}\ Ax. \qquad \cfrac{\vdots}{A_1 \vdash C_1}}{f_{op_1}(C, A_1) \vdash D_2}$$

When A_1 is the head, the derivation of $f_{op_1}(C, A_1) \vdash D_2$ is:

$$\cfrac{\cfrac{}{A_1 \vdash A_1 = \overline{op_1}(C_1, D_2)}\ Ax. \qquad \cfrac{\vdots}{C \vdash C_1}}{f_{\overline{op_1}}(A_1, C) = f_{op_1}(C, A_1) \vdash D_2}$$

We see that the derivations in long normal form of sequents like $C \vdash A$ can be built by using other derivations of sequents of the same form. We now have obtained a sufficiently precise account of all the cases that are necessary to encode derivations as terms over a second order signature.

We now define a second order signature $\Sigma_{Der\,G_{NL}}$ which encodes the grammatical analyses of an NL-grammar $G_{NL} = (W, Cat, \chi, S)$. The set of types of

$\Sigma_{Der\,G_{NL}}$ is given by $\mathcal{A}_{G_{NL}} = \{[\bullet \vdash A] | A \in \mathcal{F}_{G_{NL}}\} \cup \{[B \vdash A] | A, B \in \mathcal{F}_{G_{NL}}\}$. The types of the form $[\bullet \vdash A]$ are the type of the terms encoding the derivations whose conclusion is of the form $\Gamma \vdash A$ with Γ being a lexical base; a type of the form $[B \vdash A]$ is the type of the encodings of the derivations whose conclusion is $B \vdash A$. The set of constants of $\Sigma_{Der\,G_{NL}}$ is given by $\mathcal{C}_{G_{NL}} = C_{Const\,1} \cup C_{Const\,2} \cup C_{Var\,1} \cup C_{Var\,2}$. The constants of $C_{Const\,1}$ and of $C_{Const\,2}$ will serve the construction of derivations whose conclusions are of the form $\Gamma \vdash A$ with Γ being lexical, they respectively correspond to the cases where the head is a lexical hypothesis and where the head is a hypothesis; the constants of $C_{Var\,1}$ and of $C_{Var\,2}$ code for the two cases of derivations of the form $B \vdash A$, respectively the case where B is the head of the derivation and the case where it is not. Along with the definition of those sets we define $\tau_{G_{NL}}$ but also $\mathcal{L}_{G_{NL}}$ (we let $\mathcal{L}_{G_{NL}}([\bullet \vdash A]) = A^\dagger$ and $\mathcal{L}_{G_{NL}}([C \vdash A]) = C^\dagger \to A^\dagger$) a lexicon that *decodes* the terms built with those constants into linear λ-terms that represent the encoded derivations:

$$C_{const\,1} = \{\langle c, F, A \rangle | \quad F \in \chi(c) \wedge A \in \mathcal{F}_{G_{NL}}$$
$$\wedge\, F = op'_1(E_1, \dots op'_p(E_p, op_1(C_1, \dots op_n(C_n, B)\dots))\dots)$$
$$\wedge\, A = op_1(A_1, \dots op_n(A_n, B)\dots)$$
$$\wedge\, B \in Cat\}$$

In this case we have $\tau_{G_{NL}}(\langle c, F, A \rangle) = \beta_1 \to \cdots \beta_p \to \alpha_1 \to \cdots \to \alpha_n \to [\bullet \vdash A]$ with $\beta_i = [\bullet \vdash E_i]$ and $\alpha_j = [A_j \vdash C_j]$. Furthermore:

$$\mathcal{L}_{G_{NL}}(\langle c, F, A \rangle) = \lambda x_1 \dots x_p y_1 \dots y_n z_1 \dots z_n . \langle c, F \rangle x_1 \dots x_p (y_1\, z_1) \dots y_n\, z_n)$$

$$C_{Const\,2} = \{\langle \bullet, A \rangle | \quad A \in \mathcal{F}_{G_{NL}}$$
$$\wedge\, A = op_1(A_1, \dots op_n(A_n, C)\dots)$$
$$\wedge\, A_1 = op'_1(C_1, \dots op'_n(C_n, C)\dots)$$
$$\wedge\, op'_1 = \overline{op_1} \wedge 1 < i \le n \Rightarrow op'_i = op_i$$
$$\wedge\, C \in Cat\}$$

Here $\tau_{M_{NL}}(\langle \bullet, A \rangle)$ is $\alpha_1 \to \cdots \to \alpha_n \to [\bullet \vdash A]$ where $\alpha_1 = [\bullet \vdash C_1]$ and $\alpha_i = [A_i \vdash C_i]$ when $1 < i \le n$; and:

$$\mathcal{L}_{G_{NL}}(\langle \bullet, A \rangle) = \lambda x_1 \dots x_n z_1 \dots z_n . z_1\, x_1 (x_2\, z_2) \dots (x_n\, z_n)$$

$$C_{Var\,1} = \{\langle C, A \rangle_1 | \quad A \in \mathcal{F}_{G_{NL}} \wedge B \in \mathcal{F}_{G_{NL}}$$
$$\wedge\, A = op_1(A_1, \dots op_n(A_n, B)\dots)$$
$$\wedge\, C = op_1(C_1, \dots op_n(C_n, B)\dots)$$
$$\wedge\, B \in Cat\}$$

For that set we let $\tau_{G_{NL}}(\langle C, A \rangle_1) = \alpha_1 \to \cdots \to \alpha_n \to [C \vdash A]$ with $\alpha_i = [A_i \vdash C_i]$, and also:

$$\mathcal{L}_{G_{NL}}(\langle C, A \rangle_1) = \lambda x_1 \dots x_n z_0 \dots z_n . z_0 (x_1\, z_1) \dots (x_n\, z_n)$$

$$C_{Var\,2} = \{\langle C, A\rangle_2 | \quad A \in \mathcal{F}_{G_{NL}} \wedge B \in \mathcal{F}_{G_{NL}}$$
$$\wedge\, A = op_1(A_1, \ldots op_n(A_n, B)\ldots)$$
$$\wedge\, A_1 = op'_1(C_1, \ldots op'_n(C_n, B)\ldots)$$
$$\wedge\, op'_1 = \overline{op_1} \wedge 1 < i \le n \Rightarrow op'_i = op_i$$
$$\wedge\, B \in Cat\}$$

Finally we let $\tau_{G_{NL}}(\langle C, A\rangle_2) = \alpha_1 \to \cdots \to \alpha_n \to [C \vdash A]$ such that $\alpha_1 = [C \vdash C_1]$ and for $1 < i \le n$, $\alpha_i = [A_i \vdash C_i]$; and

$$\mathcal{L}_{G_{NL}}(\langle C, A\rangle_2) = \lambda x_1 \ldots x_n z_0 \ldots z_n.z_1(x_1\,z_0)(x_1\,z_2)\ldots(x_n\,z_n)$$

It is easy to check that whenever that the set of long normal forms of elements of $\mathcal{L}_{G_{NG}}(\Lambda^{[\bullet \vdash A]}_{\Sigma_{Der\,G_{NL}}})$ (resp. $\{Mx^{C^\dagger}|M' \in \mathcal{L}_{G_{NG}}(\Lambda^{[C \vdash A]}_{\Sigma_{Der\,G_{NL}}})\}$) is exactly the set of λ-terms representing the long normal forms of the derivations of sequents like $\Gamma \vdash A$ where Γ is a lexical base (resp. $C \vdash A$)). Note that the size of $\Sigma_{Der\,G_{NL}}$ and the size of $\mathcal{L}_{G_{NL}}$ are polynomial with respect to the size of G_{NL}.

Then according to the following picture:
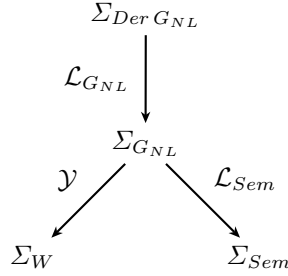
we may define into ACGs, using non-linear ACGs for the semantics, the interface between syntax and semantics that is usually defined for NL.

References

1. de Groote, P.: Towards abstract categorial grammars. In for Computational Linguistic, A., ed.: Proceedings 39th Annual Meeting and 10th Conference of the European Chapter, Morgan Kaufmann Publishers (2001) 148–155
2. de Groote, P., Pogodalla, S.: On the expressive power of abstract categorial grammars: Representing context-free formalisms. Journal of Logic, Language and Information **13**(4) (2004) 421–438
3. Lambek, J.: On the calculus of syntactic types. In Jakobson, R., ed.: Studies of Language and its Mathematical Aspects, Proceedings of the 12th Symposium of Applied Mathematics. (1961) 166–178
4. Barendregt, H.P.: The Lambda Calculus: Its Syntax and Semantics. Volume 103. Studies in Logic and the Foundations of Mathematics, North-Holland Amsterdam (1984) revised edition.
5. Huet, G.: Résolution d'équations dans des langages d'ordre 1,2,...,ω. Thèse de doctorat es sciences mathématiques, Université Paris VII (1976)

Nonlocal Dependencies via Variable Contexts

Carl Pollard[1]

The Ohio State University, Columbus, OH 43210, USA,
`pollard@ling.ohio-state.edu`

1   Introduction

This paper is an interim report on my efforts since around the turn of the millenium to develop a grammar framework that combines the advantages of derivational/proof-theoretic approaches such as Principles and Parameters (P&P) and Categorial Grammar (CG) with those of constraint-based/model-theoretic approaches such as Head-Drive Phrase Structure Grammar (HPSG) and Lexical-Functional Grammar (LFG), and the Simpler Syntax (SS) of Culicover and Jackendoff. I have done most of this work under the rubric of Higher Order Grammar (HOG), but this name is misleading since it ascribes too much importance to the choice of the the underlying type theory in terms of which grammars are expressed. At the time I embarked on this project, I was attracted by the familiar feel and conceptual clarity of Lambek and Scott's higher-order categorical logic and used that as a point of departure. But in reality, the choice of type-theoretic platform is not so important; suitably elaborated versions of linear logic, Martin-Löf type theory, the calculus of constructions, etc. would do just as well, as long as certain type-theoretic resources are available. Inter alia, these include (1) a multiplicity of implicative type-constructors, each with it own characteristic 'flavor' of resource-sensitivity (or lack thereof); (2) a mechanism for separation-style subtyping; (3) and limited facility for defining 'kinds' (special sets of types) that provide the ranges of parameters for (schematic) polymorphic typing.

From the linguistic point of view (rather than the logical one), some of the important desiderata for a grammar framework are the following. (1) It should be easy to express linguistic analyses, including ones originally developed within current widely used frameworks (both derivational and constraint-based). (2) The underlying math and logic should be as standard and mainstream as possible, rather than be based on exotic, arcane, idiosyncratic, or home-brew technology. (3) The semantics should be readily understandable to anyone who knows Montague semantics. (4) The notation should make the gist of a grammatical analysis readily comprehensible to linguists working in a wide variety of frameworks, without requiring technical knowledge of the underlying math and logic. (5) The amount of math and logic background needed for real mastery should not exceed what a linguistics grad student has time to learn in a quarter or semester.

I will use the name 'Convergent Grammar' (CVG) for the framework I will describe here. I chose this name because the framework can be seen as a com-

ing together of ideas of widely varying provenances, be they transformational, phrase-structural, or categorial. Correspondingly, the notation is intended to look like a notational variant of X-grammar to an X-grammarian, for as many values of X as possible.

Seven basic ideas of Convergent Grammar are as follows. **First, CVG is relational rather than syntactocentric.** More specifically, just as in HPSG, LFG, or SS, we think of linguistic expressions as **tuples**, i.e. entities with 'simultaneous' or 'parallel' components, rather than as having different 'levels' of structure in a 'sequential'or 'cascaded' relationship. This differentiates CVG not only from the traditional transformational architecture, but also from the variety of Curry-inspired type-theoretic frameworks (ACG, Lambda-Grammar, GF, HOG, LGL, etc.) that posit homomorphisms from tectogrammar/abstract syntax to other levels (usually phenogrammar and semantics).

Second, CVG (like the Curry-inspired frameworks, but unlike more established forms of CG such as CCG and TLG) *does* have a tectogrammatical component. But its connections to phenogrammar and semantics are relational rather than functional. This means that, unlike most kinds of CG, it is not necessary for every meaning difference (for example, which NP antecedes a pronoun, or where a quantifier or in-situ wh-phrase takes its scope) to have a syntactic reflex. Likewise, not every semantic type difference (say, the difference between an individual and an individual quantifier) has to be mirrored by a corresponding syntactic type difference; i.e. *John* and *every boy* can both be simply NPs. Analogously, it is posssible to have a phenogrammatical difference without a corresponding tectogrammatical difference, e.g. optional clitic climbing in Czech (Hana 2007). In this paper we will ignore phenogrammar and notate linguistic expressions as tecto/meaning pairs; but since most of the examples are from English, which has a fairly simple pheno-tecto interface, usually the corresponding pheno-terms can be read off of the tecto-terms by replacing each tecto-constant with its lexical phonology and erasing the punctuation.

Third, CVG makes a sharp distinction between valence (or local) dependencies (such as SUBJ (subject), SPR (specifier), and COMP (complement)), and nonlocal dependencies, such as SLASH (for extraction), REL (for relative pronouns), and QUE (for 'moved' interrogative wh-expressions). This distinction is inspired by HPSG. But unlike HPSG, CVG is based on type theory rather than feature logic, and so these various kinds of dependencies are treated as different flavors of implication rather than as attributes in feature structures. More specifically, the local vs. nonlocal distinction corresponds to the difference between implications *with* an elimination rule (modus ponens, aka merge) but no introduction rule, and ones *without* an introduction rule. The latter, which are the main focus of this paper, in turn are divided into those, such as SLASH and the INSITU implication (for *in situ* interrogative expressions) which have an introduction rule (hypothetical proof or move), and

those such as PRON (for nonrelexive pronouns) or NAME (for proper NPs) which have *neither* introduction *nor* elimination rules.[1]

Fourth, some flavors of variables (undischarged hypotheses) are stored together with a characteristic piece of semantic information. For example, variables corresponding to names, pronouns, and wh-expressions are stored together with a proposition with that variable free that serves as a sortal restriction, whereas a variable corresponding to an unscoped quantifier is stored together with the quantifier, which will take its scope when the hypothesis is discharged.[2]

Fifth, whereas most of the nonlocal implications store only a semantic variable, a few of them store a tecto/semantics ordered pair of variables. This corresponds to the difference between covert and overt movement. Implications of the latter type include SLASH (extraction), RNR (right node raising), and ASSOC (for the associate phrase in phrasal comparative constructions). What these three kinds of nonlocal dependencies have in common is the need to keep a record of the syntactic category of the expression (trace of extraction or right-node raising, or the comparative associate) that triggered the hypothesis, until the point in the derivation where the hypothesis is withdrawn.

Sixth, different flavors of nonlocal implication are subject to different structural rules. For example, in English, the SLASH implication has contraction (thereby licensing parasitic gaps) but not permutation (thereby blocking crossed dependencies); whereas the QSTOR implication, for unscoped quantifiers, has permutation (since quantifiers can scope out of their surface order) but not contraction (since, e.g. *every dog bit every dog* does not have a reading that paraphrases *every dog bit itself*). Some structural rules provide a channel of communication between two different implications. For example, a PRON (pronoun) hypothesis can contract into another hypothesis (its **antecedent**, which need not be another pronoun); intutitively, we think of this kind of contraction asymmetrically: the antecedent hypothesis absorbs the pronominal one. By comparison, a hypothesis corresponding to an unscoped quantifier (QSTOR), a name (NAME), or a wh-expression (QUE, REL, INSITU, or PSEU[3]) cannot contract into another hypothesis; in GB parlance, they are 'R-expressions' and therefore cannot be 'referentially dependent". Of course the idea that anaphora is contraction is not original; but making the syntax-semantics interface relational instead of functional enables us to limit anaphoric contraction (as opposed to parasitic-gap contraction) to the semantic side (where the only implication, the familiar in-

---

[1] The idea that phrase-structural SLASH-binding is analogous to hypothetical proof goes back a long way; I believe I first heard it from Bob Carpenter in the mid-1980's. And Chomsky's (1981) characterization of (wh-movement) traces as 'syntactic variables' and dislocated wh-expressions as 'binding operators' points, however vaguely, in the same direction.

[2] Technically, these extra pieces of semantic information should perhaps be handled by replacing the implications in question by ternary type constructors with an additional contravariant argument.

[3] For the wh-expressions in pseudoclefts.

tuitionistic one, allows contraction to begin with), without dragging the syntax (tectostructure) into it.[4]

And seventh, some hypotheses are never withdrawn. These are **parameters** (in Prawitz's sense), free variables that have to be 'anchored' in the utterance context. For example, a NAME hypothesis cannot be withdrawn, but is stored together with a sortal restriction that constrains what contextually salient individuals it can be anchored to (ones with the right name). And PRON hypotheses need not be withdrawn (in which case they are usually called deictic). In fact, there is no rule of hypothetical proof for pronominal variables; they become parameters unless they contract into nonpronominal variables (corresponding to unscoped quantifiers or wh-expressions, or to unbound traces) which themselves are withdrawn when the coirresponding operators take their scope.

2   Formal Aspects of CVG

From a formal point of view, we are going to treat linguistic expressions not simply as tuples consisting of a tecto-term and a semantic term, but rather as typing judgments about such tuples. More precisely, they will be proofs (possibly with undischarged hypotheses) in a system of natural deduction with multiple implications. The style of natural deduction we adopt employs **variable contexts** to track undischarged hypotheses, but also records, for each hypothesis, what 'flavor' of nonlocal dependency it corresponds to. In other words, the variable context is **partitioned** into different flavors of hypotheses. To explain how this works, we start with a brief overview of a simpler system using this style of natural deduction but with only one flavor of implication, namely a familiar typed lambda calculus (TLC) thought of as a Curry-Howard proof term calculus for ordinary implicative intutionistic propositional logic.

2.1   TLC with Variable Contexts

The set of TLC terms is usually defined as follows:

(1) **TLC Terms**

    a. (Hypotheses) If x is a variable of type A, then $\vdash x : A$;

    b. (Axioms) if c is a constant of type A, then $\vdash c : A$;

    c. (Modus Ponens) if $\vdash f : A \supset B$ and $\vdash a : A$, then $\vdash f(a) : B$; and

    d. (Hypothetical Proof) if x is a variable of type A and $\vdash b : B$, then $\vdash \lambda_x b : A \supset B$.

    NB.: Each type has its own set of variables (*Church typing*).

---

[4] In place of the slogan "anaphora is contraction", we suggest "binding isn't; movement is", i.e. extraction is (lambda-)binding (hypothetical proof), whereas 'binding' in the sense of antecedent-pronoun 'coindexing' is *not* binding, but rather contraction.

We introduce (variable) contexts in order to make various aspects of hypothesis management explicit.

(2) **Contexts**

    a. A **context** is a string of variable/type pairs, written to the left of the turnstile in a typing judgment.

    b. The contexts keep track of the undischarged hypotheses.

    c. Contexts are *strings* (not just sets or lists) because we track

       i. the *order* of hypotheses

      ii. *multiple occurrences* of the same hypothesis.

    d. We make explicit the *structural rules* that allow contexts to be restructured.

    e. Instead of typed variables, we use a fixed stock of general-purpose variables and let the contexts track what types are assigned to the variables in a given proof (*Curry typing*).

    f. Each variable occurs at most once in a context.

    g. We use capital Greek letters as metavariables over contexts.

Reformulated with contexts, our presentation of TLC looks like this:

(3) **Implicative TLC Reformulated Using Contexts**

    a. (Hypotheses) $x : A \vdash x : A$;

    b. (Axioms) $\vdash c : A$ (c a constant of type A);

    c. (Modus Ponens) if $\Gamma \vdash f : A \supset B$ and $\Delta \vdash a : A$, then $\Gamma, \Delta \vdash f(a) : B$;

    d. (Hypothetical Proof) if $x : A, \Gamma \vdash b : B$, then $\Gamma \vdash \lambda_x b : A \supset B$.

    e. (Weakening) if $\Gamma \vdash b : B$, then $x : A, \Gamma \vdash b : B$;

    f. (Permutation) if $x : A, y : B, \Gamma \vdash c : C$, then $y : B, x : A, \Gamma \vdash c : C$;

    g. (Contraction) if $x : A, y : A, \Gamma \vdash b : B$, then $x : A, \Gamma \vdash b[y/x] : B$.

Of course we don't really have to make the structural rules explicit; we can make Permutation and Contraction implicit by using sets of hypothees instead of strings, and make weakening implicit by allowing the discharge of nonexistent hypotheses. But this kind of presentation generalizes to our multi-implicative setting, where each nonlocal flavor of implication is subject to differerent structural rules.

2.2   Local Dependendencies

We now begin to introduce CVG, ignoring nonlocal dependencies for the moment. As in the Curry-inspired frameworks, semantics and tectostructure each have their own type logics and corresponding Curry-Howard proof term calculi. For semantics, this is just a positive intuitionistic TLC (like the implicative one above, but with the T (unit type) and $\wedge$ (conjunction) type constructors and

the usual term constructors for (nullary and binary) cartesian products that accompany them, viz. pairing, projections, and the logical constant[5] $*:$ T). To do real semantics, we then extend this TLC to a classical higher-order logic with separation subtyping and basic types Ind (individual concepts), Prop (propositions), Ent (entities), and Bool (truth values), appropriate for hyperintensional semantics. (See Pollard in press for details.)

We will need an abundance of constants for word meanings, such as the following:

(4) **Some Semantic Constants**

$\vdash$ Fido' : Ind

$\vdash$ Felix' : Ind

$\vdash$ Mary' : Ind

$\vdash$ rain' : Prop

$\vdash$ bark' : Ind $\supset$ Prop

$\vdash$ bite' : (Ind $\wedge$ Ind) $\supset$ Prop

$\vdash$ give' : (Ind $\wedge$ Ind $\wedge$ Ind) $\supset$ Prop

$\vdash$ believe' : (Ind $\wedge$ Prop) $\supset$ Prop

$\vdash$ bother' : (Prop $\wedge$ Ind) $\supset$ Prop

Turning to syntax (tectogrammar),and ignoring some fine points such as tense and agreement, we start with some basic tectogrammatical types (syntactic categories): Fin (finite clause), Inf (infinitive clause), Top (topicalized clause), $\bar{\text{S}}$ (*that*-S), It (dummy it), There (dummy there), Nom (nominative noun phrase), and Acc (accusative noun phrase).[6]

Next we need some tectogrammatical type constructors, starting with implications for each 'valence feature'.[7]

These include $\multimap_{\text{SUBJ}}$ (subject), $\multimap_{\text{SPR}}$ (specifier[8]), $\multimap_{\text{COMP}}$ (complement), and $\multimap_{\text{MKD}}$ (marked, the grammatical relation a sentence bears to a complementizer

---

[5] We use this for the meanings of semantically vacuous expressions, such as dummy pronouns.

[6] We could have started with more general syntactic categories such as NP and S and used separation subtyping to define the more fine-grained types; e.g. introduce constants nom,acc : NP $\supset$ Bool, so that Nom is an abbreviation for NP_{nom} (the subtype of NP whose characteristic function is nom). In short: so-called 'head features' are just characteristic functions.

[7] To save space, we will often abbreviate supercripts and subscripts as follows: SU (subject), SP (specifier)m C (complements), and M (marked).

[8] As in HPSG, here specifiers include, inter alia, determiners in NPs and degree specifiers of positive gradable adjectives, but not subjects or extractees as in GB.

that marks it). In addition we need a noncommutative linear conjunction (fusion) $\circ$, which is mainly used to allow verbs (as in HPSG) to select multiple complements simultaneously (i.e. without currying).

We can then introduce tectogrammatical constants such as the following:

(5) **Some Tectogrammatical Constants**

$\vdash \mathsf{it_{dum}} : \mathrm{It}$

$\vdash \mathsf{there_{dum}} : \mathrm{There}$

$\vdash \mathsf{that} : \mathrm{Fin} \multimap_\mathrm{M} \bar{\mathrm{S}}$

$\vdash \mathsf{Fido_n}, \mathsf{Felix_n}, \mathsf{Mary_n} : \mathrm{Nom}$

$\vdash \mathsf{Fido_a}, \mathsf{Felix_a}, \mathsf{Mary_a} : \mathrm{Acc}$

$\vdash \mathsf{rained} : \mathrm{It} \multimap_\mathrm{SU} \mathrm{Fin}$

$\vdash \mathsf{barked} : \mathrm{Nom} \multimap_\mathrm{SU} \mathrm{Fin}$

$\vdash \mathsf{bit} : \mathrm{Acc} \multimap_\mathrm{C} (\mathrm{Nom} \multimap_\mathrm{SU} \mathrm{Fin})$

$\vdash \mathsf{gave} : (\mathrm{Acc} \circ \mathrm{Acc}) \multimap_\mathrm{C} (\mathrm{Nom} \multimap_\mathrm{SU} \mathrm{Fin})$

$\vdash \mathsf{believed_1} : \mathrm{Fin} \multimap_\mathrm{C} (\mathrm{Nom} \multimap_\mathrm{SU} \mathrm{Fin})$

$\vdash \mathsf{believed_2} : \bar{\mathrm{S}} \multimap_\mathrm{C} (\mathrm{Nom} \multimap_\mathrm{SU} \mathrm{Fin})$

$\vdash \mathsf{bothered_1} : \mathrm{Acc} \multimap_\mathrm{C} (\bar{\mathrm{S}} \multimap_\mathrm{SU} \mathrm{Fin})$

$\vdash \mathsf{bothered_2} : (\mathrm{Acc} \circ \bar{\mathrm{S}}) \multimap_\mathrm{C} (\mathrm{It} \multimap_\mathrm{SU} \mathrm{Fin})$

Next, we need a simple tectogrammatical proof theory analogous to the semantic TLC, in order to build up complex syntactic terms. This provides an introduction term constructor for fusion, and elimination (merge) term constructors for each of the "valence features" (local flavors of implication:

(6) **Some Tectogrammatical Rules for Local Dependencies**

Fusion: If $\vdash a : A$ and $\vdash b : B$
then $\vdash a \cdot b : A \circ B$

Subject Merge: If $\vdash a : A$ and $\vdash f : A \multimap_\mathrm{SU} B$
then $\vdash (^\mathrm{SU}\ a\ f) : B$

Specifier Merge: If $\vdash a : A$ and $\vdash f : A \multimap_\mathrm{SP} B$
then $\vdash (^\mathrm{SP}\ a\ f) : B$

Complement Merge: If $\vdash f : A \multimap_\mathrm{C} B$ and $\vdash a : A$
then $\vdash (f\ a\ ^\mathrm{C}) : B$

Marked Merge: If $\vdash f : A \multimap_\mathrm{M} B$ and $\vdash a : A$
then $\vdash (f\ a\ ^\mathrm{M}) : B$

Note that the term constructors corresponding to valence features are just different flavors of eval (aka apply). In terms built with these constructors, whether the 'argument' is written to the left or the right serves as a mnemonic of how the corresponding pheno-entities (which we are ignoring) get linearized in English: subjects and specifiers to the left of the functor, complements and markees on the right. These rules enable us to construct complex tecto-terms (Curry-Howard proof terms for syntactic derivations) like the following:

(7) **A Complex Tectogrammatical Term**

$$({}^{\text{SU}} \, \mathsf{it_{dum}} \, (\mathsf{bothered}_2 \, (\mathsf{Mary_a} \cdot (\mathsf{that} \, ({}^{\text{SU}} \, \mathsf{Fido_n} \, \mathsf{barked}) \, {}^{\text{M}})) \, {}^{\text{C}}))$$

Except for the superscripts and the dot between the two fused complements, terms like these play much the same role here that traditional labelled bracketings play in transformational grammar (TG): they obviate the need to fill pages with arboreal representations (labelled order trees) of derivations. This is a not inconsiderable advantage of writing logical grammars in natural-deduction style; analogous to the fact that ordinary TLC obviates the need to display trees for proofs in intuitionistic logic.

There are some differences between our tecto-terms and TG labelled bracketings, though. For one thing, there is no need to label the left parentheses with syntactic categories, since these can be inferred from the categories of the constants. Of course this is the same reason that subterms of TLC terms don't have to be labelled with their types!

A second, and much more important, difference is that Chomskyan labelled bracketings, or their arboreal variants (phrase markers) are taken by mainstream generative grammarians to be the syntactic objects themselves. But our tecto-terms are not the tectogrammatical objects themselves, just as TLC terms are not the same thing as the functions that they denote in an interpretation. Instead, to get the actual tectogrammatical object, we have to see what the tecto-term denotes in an (algebraic or categorical) model of the grammar. As long as there are no binding operations or term equivalences (as there are in TLC), this does not really matter, since each term can be taken to denote itself in the canonical (term) model. But once we introduce variables and binding operators for nonlocal dependencies, and analogs of the usual TLC term-equivalences, it will matter very much, because it means that we can no longer think of movement as mechanically picking up a subtree of a tree and reattaching it someplace else. Instead, we really have to think of an extraction trace as a variable which is bound by some flavor (e.g. SLASH) of lambda operator. In other words, there is no sense in which anything was ever actually in the trace position and then got pulled out, just as there is no sense in which the lambda term $\lambda_x \mathsf{bite'}(\mathsf{Fido'}, x)$ was somehow derived from $\mathsf{bite'}(\mathsf{Fido'}, \lambda)$. This is the essential difference between CVG and TG.[9]

---

[9] In fact, I think the standard transformational view of movement as a destructive operation on trees is fundamentally incoherent, though I won't attempt to justify that assertion in the little time and space available to me here.

Now we have typed tecto-terms and typed semantic-terms, but we have not related them to each other yet. The usual categorial approach would be to assign semantic terms to syntactic constants, and then extend to a 'structure-preserving' total function in the familiar way. But thinking relationally rather than syntactocentrically, we instead recursively define a set of tecto-semantics pairs[10], the **signs**. In order to do that, we first have to add a (schematically) polymorphic basic type $\Sigma_{A,B}$, parametrized by a tecto-type A and a semantic type B; for each choice of A and B, $\Sigma_{A,B}$ will be a subtype of $A \wedge B$.[11]

The base of the recursion—the lexicon—consists of lexical entries like the following:[12]

(8) **Some CVG Lexical Entries**

$\vdash \mathsf{it_{dum}}, * : \Sigma_{\mathrm{It,T}}$

$\vdash \mathsf{there_{dum}} : \Sigma_{\mathrm{There,T}}$

$\vdash \mathsf{that}, \lambda_p p : \Sigma_{\mathrm{fin} \multimap_{\mathrm{M}} \bar{\mathrm{S}}, \mathrm{Prop} \supset Prop}$

$\vdash \mathsf{Fido_n}, \mathsf{Fido'} : \Sigma_{\mathrm{Nom,Ind}}$

$\vdash \mathsf{rained}, \mathsf{rain'} : \Sigma_{\mathrm{It} \multimap_{\mathrm{su}} \mathrm{Fin}, \mathrm{T} \supset \mathrm{Prop}}$

$\vdash \mathsf{barked}, \mathsf{bark'} : \Sigma_{\mathrm{Nom} \multimap_{\mathrm{su}} \mathrm{Fin}, \mathrm{Ind} \supset \mathrm{Prop}}$

$\vdash \mathsf{bit}, \lambda_y \lambda_x \mathsf{bite'}(x,y) : \Sigma_{\mathrm{Acc} \multimap_{\mathrm{C}} (\mathrm{Nom} \multimap_{\mathrm{su}} \mathrm{Fin}), (\mathrm{Ind} \wedge \mathrm{Ind}) \supset \mathrm{Prop}}$

$\vdash \mathsf{gave}, \lambda_{y,z} \lambda_x \mathsf{give'}(x,y,z):$
$\Sigma_{(\mathrm{Acc} \circ \mathrm{Acc}) \multimap_{\mathrm{C}} (\mathrm{Nom} \multimap_{\mathrm{su}} \mathrm{Fin}), (\mathrm{Ind} \wedge \mathrm{Ind}) \supset (\mathrm{Ind} \supset \mathrm{Prop})}$

$\vdash \mathsf{believed_1}, \lambda_p \lambda_x \mathsf{believe'}(x,p):$
$\Sigma_{\mathrm{Fin} \multimap_{\mathrm{C}} (\mathrm{Nom} \multimap_{\mathrm{su}} \mathrm{Fin}), (\mathrm{Prop} \wedge \mathrm{Ind}) \supset \mathrm{Prop}}$

$\vdash \mathsf{believed_2}, \lambda_p \lambda_x \mathsf{believe'}(x,p):$
$\Sigma_{\bar{\mathrm{S}} \multimap_{\mathrm{C}} (\mathrm{Nom} \multimap_{\mathrm{su}} \mathrm{Fin}), (\mathrm{Prop} \wedge \mathrm{Ind}) \supset \mathrm{Prop}}$

$\vdash \mathsf{bothered_1}, \lambda_x \lambda_p \mathsf{bother'}(p,x):$
$\Sigma_{\mathrm{Acc} \multimap_{\mathrm{C}} (\bar{\mathrm{S}} \multimap_{\mathrm{su}} \mathrm{Fin}), \mathrm{Ind} \supset (\mathrm{Prop} \supset \mathrm{Prop})}$

$\vdash \mathsf{bothered_2} : \lambda_{x,p} \lambda_t \mathsf{bother'}(p,x):$
$\Sigma_{(\mathrm{Acc} \circ \bar{\mathrm{S}}) \multimap_{\mathrm{C}} (\mathrm{It} \multimap_{\mathrm{su}} \mathrm{Fin}), (\mathrm{Ind} \wedge \mathrm{Prop}) \supset (\mathrm{T} \supset \mathrm{Prop})}$

It remains to add grammar rules to license nonlexical signs. Here we show just Fusion and Subject Merge; the formulation of the other Merge rules is similar.

(9) **Some Grammar Rules for Local Dependencies**

---

[10] Actually they would be triples if we had not ignored phenogrammar.

[11] Technically, this means we introduce a parametrized family of constants $\mathsf{sign}_{A,B} : (A \wedge B) \supset \mathrm{Bool}$, and then write $\Sigma_{A,B}$ for the subtype of $A \wedge B$ whose characteristic function is $\mathsf{sign}_{A,B}$.

[12] To avoid clutter, I omit the flanking parentheses for the tuples.

Fusion: If $\vdash a, c : \Sigma_{A,C}$ and $\vdash b, d : \Sigma_{B,D}$
then $\vdash a \cdot b, (c, d) : \Sigma_{A \circ B, C \wedge D}$

Subject Merge: If $\vdash a, c : \Sigma_{A,C}$ and $\vdash f, v : \Sigma_{A \multimap_{\text{su}} B, C \supset D}$
then $\vdash (^{\text{su}}\, a\, f), v(c) : \Sigma_{B,D}$

2.3   Extraction

Because of space limitations, we will illustrate the CVG approach to nonlocal dependencies using just one flavor of nonlocal implication, the SLASH (SL) flavor $\multimap_{\text{SL}}$, which handles extraction (roughly, the range of phenomena discussed by Chomsky (1977) under the rubric of wh-movement or by Levine and Hukari (2006) under the rubric of unbounded dependency constructions).

To get started, we need to allow SLASH-flavored hypotheses (aka traces) and also to amend the grammar rules we already have to propagate hypotheses 'upward', analogously to SLASH-inheritance in PSG:

(10) **Grammar Rules with Variable Contexts**

Trace: SL : $t, x \vdash t, x : \Sigma_{A,B}$ $(B \neq \text{T})$

Fusion: If SL : $\Gamma \vdash a, c : \Sigma_{A,C}$ and SL : $\Delta \vdash b, d : \Sigma_{B,D}$
then SL : $\Gamma; \Delta \vdash a \cdot b, (c, d) : \Sigma_{A \circ B, C \wedge D}$

Subject Merge: If SL : $\Gamma \vdash a, c : \Sigma_{A,C}$
and SL : $\Delta \vdash f, v : \Sigma_{A \multimap_{\text{su}} B, C \supset D}$
then SL: $\Gamma; \Delta \vdash (^{\text{su}}\, a\, f), v(c) : \Sigma_{B,D}$

The Trace rule simply hypothesizes, in the SLASH way, a sign with syntactic category A and semantic type B; the restriction $B \neq \text{T}$ is to prevent extraction of semantically vacuous constituents (such as dummy pronouns). The syntactic variable t is like a Chomsky-style trace, but it does not have to be replaced by a 'logical variable' x 'at LF'; instead it begins life paired with x. To put it another way, CVG can be thought of as a kind of TG in which 'Syntax' and 'LF' are derived in parallel rather than by cascading the former into the latter. Note that the field label SLASH to the left of the turnstyle tells which partition of the variable environment the hypothesis belongs to. (But since it is the only nonlocal flavor we will consider, from now on we will omit it; that is, all hypotheses in this paper are implicitly SLASH-hypotheses.)

Note that unless we say something, there is nothing to prevent any premise of any grammar rule from being a trace; that is, arbitrary constitiuents can be extracted. Of course this cannot be permitted; for example, in English extraction seems to be limited to subjects, complements, and right-hand modifiers of verbal projections. Thus we must stipulate, for each natural language, which premises of which rules can be instantiated as traces. Such stipulations play the same role as GB's ECP or HPSG's Trace Principle.

Note also that the variable contexts of two-premise rules need not be the same, that is these constructors are multiplicative relative to $\multimap_{\text{SL}}$. But our rule schema for coordination (omitted here) would require the SLASH fields of the conjuncts to be the same, to capture the ATB (across-the-board) condition on extraction from coordinate structures:

(11)   a. BAGELS$_i$, Kim likes $\mathbf{t}_i$ and Sandy hates $\mathbf{t}_i$.

    b. *BAGELS$_i$, Kim likes muffins and Sandy hates $\mathbf{t}_i$.

    c. *BAGELS$_i$, Kim likes $\mathbf{t}_i$ and Sandy hates muffins.

Now we can hypothesize extractions and remember we did so, but we still have no way to discharge hypotheses (movement) or collapse them (parasitic gaps). For movement, we posit two rules (for the finite and infinitive landing sites), and for parasitic gaps, we posit SLASH-flavored contraction. Additionally, we posit a Topicalization Rule distinct from Finite Move (and we would need other analogous rules for relative clauses, consituent questions, pseudo-clefts, etc.):

(12) **Additional Rules for** SLASH

    Finite Move: If $t, x : A, B; \Gamma \vdash s, p : \Sigma_{\text{Fin,Prop}}$
    then $\Gamma \vdash \lambda_t^{\text{SL}} s, \lambda_x p : \Sigma_{A \multimap_{\text{SL}} \text{Fin}, B \supset \text{Prop}}$

    Infinitive Move: If $t, x : A, B; \Gamma \vdash f, v : \Sigma_{C \multimap_{\text{SU}} \text{Inf}, D \supset \text{Prop}}$
    then $\Gamma \vdash \lambda_t^{\text{SL}} f, \lambda_x v : \Sigma_{A \multimap_{\text{SL}} (C \multimap_{\text{SU}} \text{Inf}), B \supset (D \supset \text{Prop})}$

    Contraction: If $\Gamma; t, x : A, B; t', y : A, B; \Delta \vdash c, d : \Sigma_{C,D}$
    then $\Gamma; t, x : A, B; \Delta \vdash c[t'/t], d[y/x] : \Sigma_{C,D}$

    Topicalization: If $\Gamma \vdash a, b : \Sigma_{A,B}$
    and $\Delta \vdash c, d : \Sigma_{A \multimap_{\text{SL}} \text{Fin}, B \supset \text{Prop}}$
    then $\Gamma, \Delta \vdash \tau(a, c), d(b) : \Sigma_{\text{Top,Prop}}$

3   Analyses

3.1   Topicalization

The rules above license proofs such as the following:

(13)   a. Felix, Fido bit.

    b. $\vdash \tau(\mathsf{Felix}_a, \lambda_t^{\text{SL}}(^{\text{SU}}\,\mathsf{Fido}_n\,(\mathsf{bit}\,t\,^{\text{C}}))), \mathsf{bite'}(\mathsf{Fido'}, \mathsf{Felix'}) : \Sigma_{\text{Top,Prop}}$

It is instructive to note the ways that this analysis is TG-like, and those in which it is not. The trace is literally a syntactic variable; it is literally bound in the syntax by a phonologically null operator (λ^{SL}) that scopes over the finite sentence; and the topicalized constituent is then adjoined to the resulting abstract. On the other hand, there is no null complementizer; there is no sense in which the empty operator occupies a Specifier position; there is no sense in which the syntax is transformed into the LF; and there is no need to further massage the LF into a real logical formula (it is already a propositional term of higher-order logic).

3.2   *Tough*-Movement

To analyze *tough*-movement, we require two more basic syntactic types S_{adj} (for adjectival small clauses) and Bse (for base-form (i.e. uninflected) clauses), and the following three lexical entries (assuming for simplicity that the copula takes only adjectival complements):

(14) **More Lexical Entries**

$\vdash \mathsf{is}, \lambda_v v : \Sigma_{(\mathrm{Nom}\multimap_{\mathrm{SU}}S_{adj})\multimap_{\mathrm{C}}(\mathrm{Nom}\multimap_{\mathrm{SU}}\mathrm{Fin}),(\mathrm{Ind}\supset\mathrm{Prop})\supset(\mathrm{Ind}\supset\mathrm{Prop})}$

$\vdash \mathsf{easy}, \lambda_r \lambda_x \mathsf{easy}'(r(x)) :$
$\Sigma_{(\mathrm{Acc}\multimap_{\mathrm{SL}}(\mathrm{Nom}\multimap_{\mathrm{SU}}\mathrm{Inf}))\multimap_{\mathrm{C}}(\mathrm{Nom}\multimap_{\mathrm{SU}}S_{adj}),(\mathrm{Ind}\supset(\mathrm{Ind}\supset\mathrm{Prop}))\supset(\mathrm{Ind}\supset\mathrm{Prop})}$

$\vdash \mathsf{to}, \lambda_v v : \Sigma_{(\mathrm{Nom}\multimap_{\mathrm{SU}}\mathrm{Bse})\multimap_{\mathrm{C}}(\mathrm{Nom}\multimap_{\mathrm{SU}}\mathrm{Inf}),(\mathrm{Ind}\supset\mathrm{Prop})\supset(\mathrm{Ind}\supset\mathrm{Prop})}$

With these additions, the grammar now licenses proofs such as the following:

(15)   a. Felix is easy to bite.
     b. $\vdash (^{\mathrm{SU}}\mathsf{Felix_n}\ (\mathsf{is}\ (\mathsf{easy}\ \lambda_t^{\mathrm{SL}}(\mathsf{to}\ (\mathsf{bite}\ t\ ^{\mathrm{C}})\ ^{\mathrm{C}})\ ^{\mathrm{C}})\ ^{\mathrm{C}})),$
       $\mathsf{easy}'(\lambda_z \mathsf{bite}'(z, \mathsf{Felix}')) : \Sigma_{\mathrm{Fin,Prop}}$

Again, this analysis is strikingly simliar to a standard TG analysis: the object of the infinitive is a trace bound in the syntax by an empty operator that scopes over the infinitive complement.[13] On the other hand, the syntactic analysis does not have to be transformed into an LF; instead it is already paired with a term whose interpretation is the proposition that the property of biting Felix itself has the property of being easy.

3.3   Extraction and Structural Rules

In so-called parasitic gap constructions, two traces appear to be bound by the same operator, seemingly violating GB's Bijection Principle (that there must be a one-to-one correspondence between traces and binders):

(16)   a. This book is hard impossible to start **t** without finishing **t**.
     b. These are the reports we filed **t** without reading **t**.
     c. Which rebel leader did rivals of **t** assassinate **t**?

We analyze these as cases where Contraction applies 'lower' in the derivation than Move. Of course this means that $\multimap_{\mathrm{SL}}$ is not linear. On the other hand, there are no structural rules of Weakening or Permutation for $\multimap_{\mathrm{SL}}$. The absence of Weakening prohibits 'movement from nowhere':

(17) **The Prohibition on 'Movement from Nowhere'**
     a. *Bagels, Kim likes muffins.

---

[13] The Infinitive Move rule is also used to analyze purpose adjuncts, infinitive relatives, infinitive constituent questions, *too/enough*-constructions, etc.

    b. *Who did you see Kim?

    c. *The student who I talked to Sandy just arrived.

    d. *What this country needs a good five-cent cigar is a good 20-cent draft beer.

    e. *It's Kim that Sandy likes Dana.

    f. *John is easy to please Mary.

And the absence of Permutation accounts for the Prohibition on Crossed Dependencies:

(18) **The Prohibition on Crossed Dependencies**

    a.   i. This violin$_j$, even the most difficult sonatas are easy λ_{t_i} [to play $\mathbf{t}_i$ on $\mathbf{t}_j$].

       ii. *This sonata$_i$, even the most exquisitely crafted violins are difficult λ_{t_j} [to play $\mathbf{t}_i$ on $\mathbf{t}_j$].

    b.   i. Which problems$_i$ don't you know who$_j$ [to talk to $\mathbf{t}_j$ about $\mathbf{t}_i$]?

       ii. *Which people$_j$ don't you know what$_i$ [to talk to $\mathbf{t}_j$ about $\mathbf{t}_i$]?

Of course, nothing *requires* two traces in the same sentence (or even the same verb phrase) to contract into each other: 'violins-and-sonatas'-type examples are analyzed unproblematically, as long as the traces are bound in the right order;

(19) a. My violin, your sonata is easy to play on.

    b. $\vdash \tau((^{\text{SP}}$ my violin$), \lambda_t^{\text{SL}}(^{\text{SU}}$ $(^{\text{SP}}$ your sonata$)$
(is (easy $\lambda_{t'}^{\text{SL}}$ (to (play $(t' \circ ($on t $^{\text{C}}))$ $^{\text{C}})$ $^{\text{C}})$ $^{\text{C}})$ $^{\text{C}})))$,
easy'(play-on'(your'(sonata'), my'(violin'))) : $\Sigma_{\text{Fin,Prop}}$

4   Conclusion

We analyze signs as tecto/semantics tuples, and grammars as natural deduction systems with partitioned variable contexts. Local (valence) dependencies are (in English, noncommutative) linear implications, and merges are their elimination rules. Nonlocal dependencies are implications without elimination rules. Extraction phenomena are analyzed in terms of a particular nonlocal implication, $\multimap_{\text{SL}}$, and movements are its introduction rules. The basic properties of movement (parasitic gaps, prohibition on movement from nowhere, prohibition on crossed dependencies) are consequences of the facts that $\multimap_{\text{SL}}$ is subject to Contraction, but not to Weakening or Permutation. The ATB condition on extraction from coordinate structures arises from the fact that the coordination constructor is additive with respect to $\multimap_{\text{SL}}$.

    As far as syntax (tectogrammar) is concerned, the resulting theory, which originated as an attempt to reconcile CG with HPSG, ends up looking a good deal like a rational reconstruction of GB. However, there is no sense in which LF is derived from syntax, and there is no need to further tweak LFs to produce real meanings. Instead, syntax and semantics are constructed in parallel, just as in Montague Grammar. This is why I call the framework Convergent Grammar.

References

1. Anoun, H. and A. Lecomte. 2006. Linear grammars with labels. *Formal Grammar* 2006.
2. Chomsky, N. 1977. On wh-movement. In P. Culicover, T. Wasow, and A. Akmajian, eds., *Formal Syntax*, 71-132. Academic Press.
3. Chomsky, N. 1981. *Lectures on Government and Binding*. Dordecht: Foris.
4. De Groote, P. 2001. Toward abstract categorial grammars. EACL 10:148-155.
5. Hana, J. 2007. *Czech Clitics in Higher Order Grammar*. Ph.D. dissertation, The Ohio State University.
6. Lambek, J. and P. Scott. 1986. *Introduction to Higher-Order Categorical Logic*. Cambridge University Press.
7. Levine, R. and T. Hukari. 2006. *The Unity of Unbounded Dependency Constructions*. Stanford:CSLI.
8. Mitchell, J. and P. Scott. 1989. Typed lambda models and cartesian closed categories. *Contemporary Mathematics* 92:301-316.
9. Muskens, R. 2003. Languages, lambdas, and logic. In G.-J. Kruiff and R. Oehrle, eds., *Resource Sensitivity in Binding and Anaphora*, 23-54. Kluwer.
10. Prawitz, D. 1965. *Natural Deduction: A Proof-Theoretical Study*. Stockholm: Almqvist and Wiksell.
11. Pollard, C. 2004. Type-logical HPSG. *Formal Grammar* 2004.
12. Pollard, C. In press. Hyperintensions. *Journal of Logic annd Computation*.
13. Ranta, A. 2002. Grammatical framework. *Journal of Functional Programming* 14:145-189.

Typed Lambda Language of Acyclic Recursion and Scope Underspecification

Roussanka Loukanova

Computational Linguistics
Uppsala University

Abstract. In this paper, I will introduce the typed lambda-calculus of acyclic recursion developed by (Moschovakis [9]) from the perspective of applications to computational semantics of natural language. The emphasis of the paper is on representation of quantifier scope underspecification.

1 Introduction

Moschovakis introduced the logical calculus of the formal language L_{ar}^λ of acyclic recursion by the intention for mathematical modelling of the computational aspects of the meanings of the natural language expressions and utterances. For any term A of the language L_{ar}^λ, the concept of *denotation* of A, $\mathsf{den}(A)$, is a proper generalization of Montague's notion of intension. I.e., $\mathsf{den}(A)$ is a function which assigns to each state (consisting of world, time, space and other context components) a designated interpretation of A in that index. The language L_{ar}^λ has two kinds of variables, which together with types and expressions for states, and the acyclic recursion, add significantly more expressive power to the language of typed λ-calculus from algorithmic perspective. The notion of *referential intension* of a term A, $\mathsf{int}(A)$, models the abstract concept of meaning as an algorithm which computes the denotation $\mathsf{den}(A)$ of A. The logic of L_{ar}^λ is a proper extension of Montague's IL since there are recursion terms of L_{ar}^λ that are not synonymous with any *explicit terms* of L_{ar}^λ: A term is explicit if the constant *where* does not occur in it, i.e., it has been constructed without using any step with acyclic recursion, and thus, is an IL term. Respectively, there are different NL sentences that translate into the same λ-term of Montague's IL, and, thus, are represented as synonymous in Montague's IL; but, the language L_{ar}^λ differentiates these sentences by rendering them into different terms of L_{ar}^λ which are not referentially synonymous. Any two referentially synonymous terms of L_{ar}^λ are associated with identical algorithms for computing their denotational meanings.

The first part of this paper introduces the syntax of the language of L_{ar}^λ, its denotational and computational semantics, and the concepts of referential intension and referential synonymy of the terms of L_{ar}^λ. The logic of L_{ar}^λ is summarized by giving the rules of the reduction calculus for computing the canonical forms of the terms and some major results, such as the Referential

Synonymy Theorem. Since the language L_{ar}^{λ} of acyclic recursion is developed for rendering NL expressions into terms of L_{ar}^{λ}, which model their referential meanings, a natural question arises: what terms are rendering NL sentences with two or more quantifier scope readings? Any such sentence may have more than one denotational interpretation, and thus, different meanings depending on context and other factors such as speakers' intentions. The last, major part of the paper considers the potential power of the language L_{ar}^{λ} for representing quantifier scope ambiguities in NL.

I elaborate on terms with multiple quantifiers the scopes of which are represented as underspecified. Such terms represent abstract mathematical algorithms for computing denotations of varying scopes. I show with examples how such terms can be specified to particular scoping of quantifiers. Some of the examples and their representations in terms of L_{ar}^{λ} have been inspired by work on "minimal recursion semantics" (MRS) by Copestake et al. ([1]).

2   The Typed λ-language of Acyclic Recursion

Gallin ([2]) introduced the two-sorted type theoretic system TY_2 in which Montague's intensional logic IL (see Montague [3]–[5], [6], [7], [8]) can be interpreted. Moschovakis' typed λ-calculus L_{ar}^{λ} of acyclic recursion is a proper extension of TY_2. In what follows, I will define the syntax, semantics and calculus of L_{ar}^{λ} by following Moschovakis ([9]).

2.1   Syntax of L_{ar}^{λ}

The set *Types* of types of L_{ar}^{λ} is the smallest set defined recursively as follows:

1. $e, t, s \in \textit{Types}$
2. If $\tau_1, \tau_2 \in \textit{Types}$, then $(\tau_1 \rightarrow \tau_2) \in \textit{Types}$.

The above definition can be given, as in Moschovakis ([9]), in the following way, which is commonly used in computer science:

$$(\textit{Types}) \qquad\qquad \tau :\equiv e \mid t \mid s \mid (\tau_1 \rightarrow \tau_2)$$

The intuition behind types is that they classify the kinds of objects denoted by well-formed expressions (called terms) of L_{ar}^{λ} depending on their types. For each type τ, there is a set $\mathbb{T}_\tau$ of objects of type τ; and L_{ar}^{λ} has a set of terms associated with τ which can denote only objects in $\mathbb{T}_\tau$. The association of a type τ with a term A is denoted by $A : \tau$. The type e is associated with primitive objects (entities called individuals) of the domain, as well as with the terms of L_{ar}^{λ} denoting individuals. The type s is for states consisting of various context information such as a possible world (a situation), a time moment, a space location, and a speaker; t is the type of the truth values. The type $(\tau_1 \rightarrow \tau_2)$ is associated with unary functions that take as arguments objects of type τ_1 and have as values objects of type τ_2.

The Vocabulary of L_{ar}^{λ} consists of constants of all types. For each type $\tau \in$ *Types*, L_{ar}^{λ} has a finite set K_{τ} the elements of which are called *constants* of type τ. For each type $\tau \in$ *Types*, L_{ar}^{λ} has two (nonempty) denumerable sets, *PureVar*$_{\tau}$ and *RecVar*$_{\tau}$, of variables of type τ. All these sets of constants and variables are pairwise disjoint, i.e., they do not have common elements:

$(Constants\ of\ type\ \tau)$ $\qquad\qquad\qquad\qquad K_{\tau} = \{\mathsf{c}_0^{\tau}, \ldots, \mathsf{c}_{k_{\tau}}^{\tau}\}$

$(Constants)$ $\qquad\qquad\qquad\qquad\qquad K = \bigcup_{\tau \in Types} K_{\tau}$

$(Pure\ Variables\ of\ Type\ \tau)$ $\qquad PureVars_{\tau} = \{v_0^{\tau}, v_1^{\tau}, \ldots\}$

$(Pure\ Variables)$ $\qquad\qquad\qquad PureVars = \bigcup_{\tau \in Types} PureVars_{\tau}$

$(Recursion\ Variables\ of\ type\ \tau)$ $\qquad RecVars_{\tau} = \{p_0^{\tau}, p_1^{\tau}, \ldots\}$

$(Recursion\ Variables)$ $\qquad\qquad RecVars = \bigcup_{\tau \in Types} RecVars_{\tau}$

$$PureVars \neq RecVars \neq Vars \neq K$$

Pure variables are used for quantification, while recursion variables are used for "saving" or "storing" results of intermediate calculations during complex recursive computations starting from the basic facts. This is why, the recursion variables are called also *locations*. Some practical abbreviations are:

(1) $\qquad\qquad \widetilde{\mathsf{t}} :\equiv (\mathsf{s} \to \mathsf{t})$ the type of propositions

(2) $\qquad\qquad \widetilde{\mathsf{e}} :\equiv (\mathsf{s} \to \mathsf{e})$ the type of individual concepts

(3) $\qquad\qquad \widetilde{\tau} :\equiv (\mathsf{s} \to \tau)$ where $\tau \in$ *Types*

(4) $\quad \tau_1 \times \tau_2 \to \tau \equiv (\tau_1 \to (\tau_2 \to \tau))$ where $\tau_1, \tau_2, \tau \in$ *Types*

(5) $\qquad A(B, C) \equiv A(B)(C)$ where $A : (\tau_1 \to (\tau_2 \to \tau))$, $B : \tau_1$ and $C : \tau_2$

Table 1 gives typical natural language syntactic categories with examples of renderings (translations) of lexical items into the language L_{ar}^{λ} and their types. An important feature of the language L_{ar}^{λ} is that it has expressions, such as variables and constants, of type s for states. Intuitively, while natural language does not have lexical items for states, including expressions for states as components of the logical representations of meanings explicates the dependence of interpretations on the context, e.g., on the states. For a detailed discussion of the importance of including expressions for states in a formal language for representing meanings, and the impact on its logic theory, see Muskens ([10]). **The Terms** of $L_{ar}^{\lambda}(K)$ are defined by recursion starting with constants K and variables. Typical notations throughout the paper are

(6) $\qquad A : \tau \iff A$ is a term of $L_{ar}^{\lambda}(K)$ of type $\tau \iff A \in Terms_{\tau}$

The set of terms of $L_{ar}^{\lambda}(K)$ is $Terms = \bigcup_{\tau \in Types} Terms_{\tau}$, where for every $\tau \in$ *Types*, the set $Terms_{\tau}$ of terms of type τ, and the free and bound occur-

Abbr NL Category		L_{ar}^{λ} Constants	L_{ar}^{λ} Type
	Pure objects	$0, 1, 2, \ldots$	e
NP	Proper names	$john, mary, \ldots$	$\widetilde{\mathsf{e}}$
IV	Intransitive Verbs	$run, smile, \ldots$	$\widetilde{\mathsf{e}} \to \widetilde{\mathsf{t}}$
CN	Common Nouns	$man, woman, dog, \ldots$	$\widetilde{\mathsf{e}} \to \widetilde{\mathsf{t}}$
TV	Transitive Verbs	$like, love, \ldots$	
NP	Quantified Noun Phrases		$(\widetilde{\mathsf{e}} \to \widetilde{\mathsf{t}}) \to \widetilde{\mathsf{t}}$
DET	Determiners	$every, some, a, \ldots$	$(\widetilde{\mathsf{e}} \to \widetilde{\mathsf{t}}) \times (\widetilde{\mathsf{e}} \to \widetilde{\mathsf{t}}) \to \widetilde{\mathsf{t}}$

Table 1. Examples of L_{ar}^{λ} constants and types rendering NL expressions (words and phrases)

rences of variables and constants in them, are defined recursively as follows:

(Const)
If $\mathsf{c} \in K_\tau$, then $\mathsf{c} : \tau$
(i.e., every constant of type τ is also a term of type τ);
the occurrence of c in the term c is free.

(Vars)
If $x \in PureVars_\tau \cup RecVars_\tau$, then, as a term, $x : \tau$
(i.e., every variable of type τ is also a term of type τ);
the occurrence of x in the term x is free.

(Appl)
If $A : (\sigma \to \tau)$ and $B : \sigma$, then $A(B) : \tau$;
all free occurrences of variables (constants) in A and B are also free in $A(B)$;
all bound occurrences of variables in A and B are also bound in $A(B)$.

(λ-abstr)
If $x \in PureVars_\sigma$ (i.e., $x : \sigma$) and $A : \tau$, then $\lambda x(A) : (\sigma \to \tau)$;
all occurrences of x in $\lambda x(A)$ are bound;
all bound occurrences of variables in A are also bound in $\lambda x(A)$;
all other free occurrences of variables (constants) in A, except
the occurrences of x, are also free in $\lambda x(A)$.

(ARec)
For any $n \geq 0$, if $A_i : \sigma_i$ $(0 \leq i \leq n)$, $p_i \in RecVars_{\sigma_i}$, $p_i \neq p_j$ $(1 \leq i, j \leq n)$
and $\{p_1 := A_1, \ldots, p_n := A_n\}$ is an acyclic system,
then A_0 where $\{p_1 := A_1, \ldots, p_n := A_n\} : \sigma_0$;
all occurrences of $p_1, \ldots, p_n$ in A_0 where $\{p_1 := A_1, \ldots, p_n := A_n\}$ are bound;

all other free occurrences of variables (constants) in $A_0, \ldots, A_n$ are also free in A_0 where $\{p_1 := A_1, \ldots, p_n := A_n\}$.

Acyclic Recursion Let $n \geq 0$, and for all i, $1 \leq i \leq n$, $\sigma_i \in Types$, $A_i : \sigma_i$, $p_i \in RecVars_{\sigma_i}$, and $p_i \neq p_j$ $(1 \leq i, j \leq n)$. The system of assignments $\{p_1 := A_1, \ldots, p_n := A_n\}$ is *acyclic* if there is a rank assignment function $\mathsf{rank} : \{p_1, \ldots, p_n\} \longrightarrow \mathbb{N}$ such that

(8)   for all $i, j \in \{1, \ldots, n\}$, if p_j occurs free in A_i, then $\mathsf{rank}(p_j) < \mathsf{rank}(p_i)$.

An L_{ar}^λ term A is called: *explicit* if the operator where does not occur in it; *recursive* if it is of the form A_0 where$\{p_1 := A_1, ..., p_n := A_n\}$ $(n \geq 1)$; an IL λ-calculus term if it is explicit and no recursion variable occurs in it; and *closed* if it has no free occurrences of variables (pure or recursive).

2.2   Denotational Semantics of L_{ar}^λ

An L_{ar}^λ *frame* is a set $\mathbb{T} = \{\mathbb{T}_\sigma \mid \sigma \in Types\}$, where $\mathbb{T}_\sigma$ are sets such that

(S1) $\mathbb{T}_e \neq \varnothing$ is a nonempty set of objects called *individuals* (entities), and $\{0, 1, er\} \subseteq \mathbb{T}_e$; $\mathbb{T}_t = \mathbb{T}_e$ is the set of the truth values; $\mathbb{T}_s \neq \varnothing$ is a nonempty set of states;

(S2) For all types $\tau_1, \tau_2 \in Types$,

$$\mathbb{T}_{(\tau_1 \rightarrow \tau_2)} \subseteq \{p \mid p \colon \mathbb{T}_{\tau_1} \longrightarrow \mathbb{T}_{\tau_2}\},$$

i.e., $\mathbb{T}_{(\tau_1 \rightarrow \tau_2)}$ is a subset of the set of all unary functions with arguments of types τ_1 and values of type τ_2.

A *variable assignment* is a function $g \colon PureVars \cup RecVars \longrightarrow \mathbb{T}$, which assigns semantic objects to the variables by respecting their typing. I.e., for each type $\tau \in Type$ and each variable $x \in PureVars_\tau \cup RecVars_\tau$, $g(x) \in \mathbb{T}_\tau$. For any type $\tau \in Type$, assignment g, object $t \in \mathbb{T}_\tau$ and variable $x \in PureVars_\tau \cup RecVars_\tau$, the notation $g\{x := t\}$ is used for the assignment having the same values as g for all variables, except possibly for the variable x to which $g\{x := t\}$ assigns the object t; $g\{x := t\}$ is called the *update* of g by the assignment $\{x := t\}$. Formally, $g\{x := t\}$ is the function such that, for every type $\tau \in Type$, $x, y \in PureVars_\tau \cup RecVars_\tau$, and $t \in \mathbb{T}_\tau$:

$$g\{x := t\}(y) = \begin{cases} g(y) & \text{if } y \neq x, \\ t & \text{if } y = x. \end{cases}$$

An L_{ar}^λ *structure* is a tuple $\mathfrak{A} = \langle \mathbb{T}, \mathcal{I}, \mathsf{den} \rangle$, satisfying the following conditions (S1)–(S4):

(S1)–(S2) $\mathbb{T}$ is an L_{ar}^λ frame, i.e., $\mathbb{T}$ satisfies the conditions (S1)–(S2).
(S3) $\mathcal{I}$ is a function $\mathcal{I} \colon K \longrightarrow \mathbb{T}$ (called the *interpretation* function of $\mathfrak{A}$) such that for every constant $\mathsf{c} \in K_\tau$, $\mathcal{I}(\mathsf{c}) = c$ for some $c \in \mathbb{T}_\tau$.

(S4) Given the set G of all variable assignments $g\colon PureVars \cup RecVars \longrightarrow \mathbb{T}$, den is a function, called the *denotation function* of $\mathfrak{A}$,

$$\mathsf{den}\colon Terms \longrightarrow \{f\colon G \longrightarrow \mathbb{T}\}$$

defined by recursion on the structure of the terms:

(D1) $\mathsf{den}(x)(g) = g(x);\ \ \mathsf{den}(c)(g) = \mathcal{I}(c);$

(D2) $\mathsf{den}(A(B))(g) = \mathsf{den}(A)(g)(\mathsf{den}(B)(g));$

(D3) $\mathsf{den}(\lambda x(B))(g) = h$, where h is the function defined as follows: for every $t \in \mathbb{T}_\tau$, where τ is such that $x : \tau$ (i.e., τ is the type of x), $h(t) = \mathsf{den}(B)(g\{x := t\});$

(D4) $\mathsf{den}(A_0 \text{ where } \{p_1 := A_1, \ldots, p_n := A_n\})(g) =$
$$\mathsf{den}(A_0)(g\{p_1 := \overline{p}_1, \ldots, p_n := \overline{p}_n\}),$$

where $\overline{p}_i \in \mathbb{T}_{\tau_i}$, for τ_i such that $p_i : \tau_i$, and $i \in \{1, \ldots, n\}$, are defined by recursion on $\mathsf{rank}(p_i)$ (not by the order of the appearing listing in the system) so that:

$$\overline{p_i} = \mathsf{den}(A_i)(g\{p_{k_1} := \overline{p}_{k_1}, \ldots, p_{k_m} := \overline{p}_{k_m}\}),$$

where $p_{k_1}, \ldots, p_{k_m}$ are the recursion variables $p_j \in \{p_1, \ldots, p_n\}$ such that $\mathsf{rank}(p_j) < \mathsf{rank}(p_i)$, i.e., $p_{k_1}, \ldots, p_{k_m}$ are the variables with ranks lower than $\mathsf{rank}(p_i)$.

Denotational Synonymy is defined as denotational equality between terms in a given structure $\mathcal{A}$:

(9) $\models A = B \iff$ for every assignement g, $\mathsf{den}(A)(g) = \mathsf{den}(B)(g)$

A structure $\mathfrak{A}$, and its frame $\mathbb{T}$, are called *standard* if for all $\tau_1, \tau_2 \in Types$,

$$\mathbb{T}_{(\tau_1 \to \tau_2)} = \{f \mid f\colon \mathbb{T}_{\tau_1} \longrightarrow \mathbb{T}_{\tau_2}\}.$$

In a standard structure $\mathfrak{A}$, (S4) defines recursively a unique denotation function $\mathsf{den}\colon Terms \longrightarrow \{f\colon G \longrightarrow \mathbb{T}\}$, such that for each term $A \in Terms_\tau$, $\mathsf{den}(A)$ is a function from variable assignments to objects in $\mathbb{T}_\tau$ of the appropriate type:

(10a)   if $A : \tau$, then $\mathsf{den}(A)\colon G \longrightarrow \mathbb{T}_\tau$ and

(10b)   for every $g \in G$, $\mathsf{den}(A)(g) \in \mathbb{T}_\tau$

Some simple, but important terms, formed from variables of both kinds, do not have meanings because they denote "immediately". Terms of the form p, u, $p(v_1, \ldots, v_n)$, $\lambda u_1 \ldots \lambda u_n\, p$, $\lambda u_1 \ldots \lambda u_n\, p(v_1, \ldots, v_m)$, where p is any recursion variable, and $u, u_1, \ldots, u_n, v_1, \ldots, v_m$ are pure variables, are called *immediate terms*, and are used throughout this paper. For a detailed discussion and the formal definition of the immediate terms, see Moschovakis ([9]). Intuitively, one may think of them as directly accessible memory locations for storing functions of the respective type with all their values already computed. Thus, after values are stored in the immediate terms, they do need to be computed. The immediate terms do not have meanings, since we take the notion of meaning to be "algorithm for computing semantic objects".

3   Referential Intension and Referential Synonymy

3.1 L_{ar}^{λ} Reduction Calculus

Congruence Two terms $A, B \in \mathit{Terms}$ are congruent, denoted by $A \equiv_c B$, if one can be obtained from the other by alphabetic changes of bound variables and *permutation* re-ordering of the assignment equations within acyclic recursion constructs. For a formal definition of the permutation re-ordering, see p. 6, Moschovakis ([9]). The rules of the L_{ar}^{λ} reduction calculus define a reduction relation between terms, $\Rightarrow \subseteq \mathit{Terms} \times \mathit{Terms}$. Intuitively, the reduction relation guarantees that:

> $A \Rightarrow B$ if and only if A and B are congruent, i.e., $A \equiv_c B$ (in some cases of simple terms A and B), *or* A and B have the same meaning and B represents that meaning more simply.

The following abbreviation simplifies the statement of some reduction rules:

$$\overrightarrow{r} := \overrightarrow{X} \equiv \{r_1 := X_1, \ldots, r_n := X_n\}$$

Reduction Rules:

(cong)	If $A \equiv_c B$, then $A \Rightarrow B$
(trans)	If $A \Rightarrow B$ and $B \Rightarrow C$, then $A \Rightarrow C$
(rep1)	If $A \Rightarrow A'$ and $B \Rightarrow B'$, then $A(B) \Rightarrow A'(B')$
(rep2)	If $A \Rightarrow B$, then $\lambda u\,(A) \Rightarrow \lambda u\,(B)$
(rep3)	If $A_i \Rightarrow B_i$, for $i = 0, \ldots, n$,

$$\text{then}\quad A_0 \text{ where } \{p_1 := A_1, \ldots, p_n := A_n\} \Rightarrow$$
$$B_0 \text{ where } \{p_1 := B_1, \ldots, p_n := B_n\}$$

(head)   **The head rule**

$$\big(A_0 \text{ where } \{\overrightarrow{p} := \overrightarrow{A}\}\big) \text{ where } \{\overrightarrow{q} := \overrightarrow{B}\}$$
$$\Rightarrow\ A_0 \text{ where } \{\overrightarrow{p} := \overrightarrow{A},\ \overrightarrow{q} := \overrightarrow{B}\}$$

where no p_i occurs free in any B_j for $i = 1, \ldots, n,\ j = 1, \ldots, m$

(B-S)   **The Bekič-Scott rule**

$$A_0 \text{ where } \{p := \big(B_0 \text{ where } \{\overrightarrow{q} := \overrightarrow{B}\}\big), \overrightarrow{p} := \overrightarrow{A}\}$$
$$\Rightarrow A_0 \text{ where } \{p := B_0, \overrightarrow{q} := \overrightarrow{B},\ \overrightarrow{p} := \overrightarrow{A}\}$$

where no q_i occurs free in any A_j for $i = 1, \ldots, n,\ j = 1, \ldots, m$

(recap)   **The recursion-application rule**

$$\left(A_0 \text{ where } \{\overrightarrow{p} := \overrightarrow{A}\}\right)(B) \Rightarrow A_0(B) \text{ where } \{\overrightarrow{p} := \overrightarrow{A}\}$$

where no p_i occurs free in B for $i = 1, \ldots, n$

(ap)   **The application rule**

$A(B) \Rightarrow A(p) \text{ where } \{p := B\}$ where B is proper, p is a new location

(λ-rule)   **The λ-rule**

$\lambda u \, (A_0 \text{ where } \{p_1 := A_1, \ldots, p_n := A_n\})$

$\qquad \Rightarrow \lambda u \, A_0' \text{ where } \{p_1' := \lambda u \, A_1', \ldots, p_n' := \lambda u \, A_n'\}$,   where

for all $i = 1, \ldots, m$, p_i' is a fresh location and A_i' is the substitution

$A_i' :\equiv A_i \{p_1 :\equiv p_1'(u), \ldots, p_n :\equiv p_n'(u)\}$.

Theorem §3.11. (see p. 25, Moschovakis [9]) If $A \Rightarrow B$, then $\models A = B$, i.e., A and B have identical denotations. (The proof is by the definition of the reduction relation.)

A term A is *irreducible* if the only reduction rules that can be applied to it are renaming of bound variables (pure or recursion) and reordering of assignments in the where-construct:

(12)   A term A is *irreducible* $\iff$ for every B, if $A \Rightarrow B$, then $A \equiv_c B$.

Canonical Form Theorem For each term A, there is a unique irreducible term denoted by $\mathsf{cf}(A)$, such that:

1. $\mathsf{cf}(A) \equiv A$ or $\mathsf{cf}(A) \equiv A_0 \text{ where } \{p_1 := A_1, \ldots, p_n := A_n\}$
2. $A \Rightarrow \mathsf{cf}(A)$
3. $\mathsf{cf}(A)$ is the unique up to congruence irreducible term to which A can be reduced, i.e., if $A \Rightarrow B$ and B is irreducible, then $B \equiv_c \mathsf{cf}(A)$.

3.2   Meanings as Abstract Algorithms

The language of acyclic recursion is naturally associated with twofold semantic information: (1) denotations of language expressions, and (2) referential intensions which are algorithms for computing denotations. Each well-formed expression A has referential intension which "computes" the denotation $\mathsf{den}(A)$ of A in given contexts modeled by interleaving semantic structures, interpretations of the vocabulary and variable assignments:

$$\underbrace{\text{Syntax} \Longrightarrow \text{Referential Intensions (Algorithms)} \Longrightarrow \text{Denotations}}_{\text{Semantics}}$$

The system of reduction rules that define the reduction relation provides a transparent way, i.e., an effective procedure to determine the denotations of expressions and their synonymy. The meaning of a term A, called the referential intension of A and denoted by $\mathsf{Int}(A)$, is the abstract algorithm[1] which computes the

---

[1] For a formal definition of $\mathsf{Int}(A)$, see Moschovakis ([9]).

denotation of A, $\mathsf{den}(A)$, where the definition of the denotation function $\mathsf{den}(A)$ is given by the clause (D4).

The canonical form $\mathsf{cf}(A)$ of A represents the algorithm $\mathsf{Int}(A)$ for calculating the meaning of A (if it exists) and, thus, of $\mathsf{cf}(A)$), in the simplest way: Recursively, the algorithm computes the basic facts and "stores" them in locations represented by recursion variables; Then, incrementally, the more complex facts, which depend on simpler ones, are computed, their values are also stored in locations, etc. until computing the denotation of the entire term.

Intuitively, two proper terms A and B are *referentially synonymous*, written $A \approx B$, if they model identical abstract algorithms, i.e., if they have identical referential intensions. Formally:

Referential Synonymy Theorem Two terms A, B are *referentially synonymous* $A \approx B$ iff A and B reduce to canonical forms with denotationally equivalent parts, i.e.

$$A \Rightarrow_{cf} A_0 \text{ where } \{p_1 := A_1, \ldots, p_n := A_n\},$$

$$B \Rightarrow_{cf} B_0 \text{ where } \{p_1 := B_1, \ldots, p_n := B_n\},$$

for some $n \geq 0$ and some terms, $A_0, A_1, \ldots, A_n, B_0, B_1, \ldots, B_n$, of appropriate types, such that:

$$\models A_i = B_i \ (i = 0, 1, \ldots, n).$$

3.3   Some Examples

In the rest of the paper, I will consider examples[2] of L_{ar}^{λ} of terms and reductions into forms that are simpler from computational point. The canonical form (modulo congruence) of a meaningful term is the simplest way to express the algorithm computing its meaning. In the considered examples, the types of the variables and constants will be taken appropriately depending on the terms in which they are used.

(13a)   Every cat sleeps.

(13b) $\xrightarrow{\text{render}} every(\lambda x \, cat(x), \lambda x \, sleep(x))$

(13c) $\Rightarrow every(p_1, p_2) \text{ where } \{p_1 := \lambda x \, cat(x), \ p_2 := \lambda x \, sleep(x))\}$

(13d) $\approx every(p_1, p_2) \text{ where } \{p_1 := cat, \ p_2 := sleep\}$

The following examples demonstrate rendering of sentences with anaphoric NPs. In them, m, m_1, and m_2 are recursion variables denoting individual concepts: $m : \widetilde{\mathsf{e}}$, $m_1 : \widetilde{\mathsf{e}}$, $m_2 : \widetilde{\mathsf{e}}$. I.e., the individuals denoted by m, m_1, and m_2 depend on the state. Even assuming that the three occurrences of the proper name Mary the NL sentences (14) and (15) designate the same person, in any

---

[2] The relation render is translation of natural language expressions into L_{ar}^{λ}, which is not the subject of this paper.

circumstances, these sentences do not have the same meanings. This is captured properly by the calculus of L_{ar}^λ as the following renderings show:

(14)   Mary likes herself.

$$\xrightarrow{\text{render}} \lambda x\, like(x,x)(mary) \approx \lambda x\, like(x,x)(m) \text{ where } \{m := mary\}$$

Despite that the denotations of m_1 and m_2 in the terms rendering (15) are equal, the terms in (15) do not have the same meanings as those in (14), i.e., they are not referentially synonymous.

(15)   Mary likes Mary.

$$\xrightarrow{\text{render}} like(mary, mary) \equiv \big(like(mary)\big)(mary)$$

$$\Rightarrow \big(like(mary)\big)(m_2) \text{ where } \{m_2 := mary\} \qquad\qquad\qquad \text{(ap)}$$

$$\Rightarrow \big(like(m_1) \text{ where } \{m_1 := mary\}\big)(m_2) \text{ where } \{m_2 := mary\} \quad \text{(ap, rep1,3)}$$

$$\Rightarrow \big(like(m_1)(m_2) \text{ where } \{m_1 := mary\}\big) \text{ where } \{m_2 := mary\} \quad \text{(recap, rep3)}$$

$$\Rightarrow like(m_1)(m_2) \text{ where } \{m_1 := mary,\, m_2 := mary\} \qquad\qquad \text{(head)}$$

$$\equiv like(m_1, m_2) \text{ where } \{m_1 := mary,\, m_2 := mary\} \qquad\qquad \text{(abbr)}$$

The rules for reduction, the Referential Synonymy Theorem of L_{ar}^λ and the rules for referential synonymy given in Table 8 on p. 30, Moschovakis ([9]) are the essential components of the calculus of referential synonymy, for representing meanings of NL expressions in a more adequate way from computational perspective.

Given that *like* is a constant, the following statements are valid in the calculus of referential synonymy:

(16a) $$like \approx \lambda x \big(\lambda y\, like(x,y)\big)$$

(16b) $$like(j,m) \approx \Big(\big(\lambda x \big(\lambda y\, like(x,y)\big)\big)(j)\Big)(m)$$

$$\equiv \big(\lambda x\, \lambda y\, like(x,y)\big)(j,m)$$

A very weak form of β-reduction is valid for the referential synonymy of L_{ar}^λ:

(β-reduction) $\big(\lambda u\, C\big)(v) \approx C\{u :\equiv v\}$, where

C is explicit and irreducible, u and v are pure variables,

and the substitution $C\{u :\equiv v\}$ is free.

This form of β-reduction does not justify referential synonymy of the terms in (17a)-(17c), if $m \in RecVar_{\widetilde{e}}$ is a recursion variable. Also, the canonical forms of the terms (17a)-(17e) are not congruent, thus:

(17a) $$\lambda y\big(\lambda x\, like(x,y)\big)(m) \not\approx \lambda x\, like(x,m)$$

(17b) $$\lambda x\, like(x,x)(mary) \not\approx like(mary, mary)$$

(17c) $$\lambda x\, like(x,x)(m) \not\approx like(m,m)$$

(17d) $\lambda x\,\lambda y\, like(x,y) \not\approx \lambda y\,\lambda x\, like(x,y)$

(17e) $like \not\approx \lambda y\,\big(\lambda x\, like(x,y)\big)$

(18a) is a simple example for a NL sentence with a quantifier NP and a possible rendering into referentially synonymous terms, where (18e) is its canonical form modulo congruence. The referential synonymy between the terms (18e) and (18f) follows from the rules of referential synonymy (see p. 30, Moschovakis [9]) and the synonymy $\lambda x\, cat(x) \approx cat$, which is valid if $cat : \widetilde{e}$ is a constant, i.e., if $cat \in K_{\widetilde{e}}$.

(18a)   Every cat likes Maja.

(18b) $\xrightarrow{\text{render}} \ldots \xrightarrow{\text{formalize}}$ (not in the subject of this paper)

(18c) $every(\lambda x\, cat(x), \lambda y\,\big(\lambda x\, like(x,y)\big)(maja))$

(18d) $\Rightarrow every(p_1, p_2)$ where
$$\{p_1 := \lambda x\, cat(x),\ p_2 := \lambda y\,\big(\lambda x\, like(x,y)\big)(maja)\}$$

(18e) $\Rightarrow every(p_1, p_2)$ where
$$\{p_1 := \lambda x\, cat(x),\ p_2 := \lambda y\,\big(\lambda x\, like(x,y)\big)(m),\ m := maja\}$$

(18f) $\approx every(p_1, p_2)$ where
$$\{p_1 := cat,\ p_2 := \lambda y\,\big(\lambda x\, like(x,y)\big)(m),\ m := maja\}$$

The sentence (18a) could be rendered into a different term that is not referentially synonymous to the terms in (18e) and (18f):

(19a)   Every cat likes Maja.

(19b) $\xrightarrow{\text{render}} \ldots \xrightarrow{\text{formalize}}$

(19c) $\Rightarrow every(p_1, p_2)$ where
$$\{p_1 := \lambda x\, cat(x),\ p_2 := \lambda x\, like(x, maja)\}$$

(19d) $\Rightarrow every(p_1, p_2)$ where
$$\{p_1 := \lambda x\, cat(x),\ p_2 := \lambda x\, like(x,m),\ m := maja\}$$

(19e) $\approx every(p_1, p_2)$ where
$$\{p_1 := cat,\ p_2 := \lambda x\, like(x,m),\ m := maja\}$$

Computationally and intuitively, the terms (18e) and (19d) mean different things.

4   Quantifier Scope Underspecification

In what follows, I will consider sentences like (20) which have occurrences of quantifier NPs posing multiple quantifier scope distributions. Such sentences (i.e., their syntactic analyses) would be rendered into terms of L_{ar}^{λ} representing their meanings with *quantifier scope underspecification*. Either of the several

different readings would be obtained in the availability of appropriate context information. The definition of the render relation (translation) is dependant on the syntactic theory of NL, and is not a subject of this paper. In what follows, I will consider the plausible L_{ar}^{λ} canonical terms representing the meanings of such renderings. I assume that the indexing of terms rendering NPs happens in the syntax-semantics interface representation, as for example in constraint based approaches, see Sag et al. ([11]). Let us consider the following NL sentence:

(20) $[[\text{Every man}]_{NP_i} \text{ loves } [\text{a woman}]_{NP_j}]_S$.

The semantic underspecification of the sentence (20) can be depicted by the following graph ("underspecified tree"):

(21) $\bullet \, [q := every(m, \lambda i\, p_1)]_i$ $\bullet \, [r := some(w, \lambda j\, p_2)]_j$ $\bullet \, p := love(i, j)$

$\bullet \, m := \overbrace{\lambda x\, man(x)}$ $\bullet \, p_1$ $\bullet \, w := \overbrace{\lambda y\, woman(y)}$ $\bullet \, p_2$

A candidate term for representing the underspecified meaning of the sentence (20) is the following one:

(22)

$\xrightarrow{\text{render}} h$ where $\{q := every(m, \lambda i\, p_1),$ (over-underspecified)

$\qquad\qquad r := some(w, \lambda j\, p_2),$

$\qquad\qquad m := \lambda x\, man(x), \; w := \lambda y\, woman(y),$

$\qquad\qquad p := love(i, j)\}$

where, the constants are as given in Table 1, and the variables are of the appropriate types.

We will call a term A of L_{ar}^{λ} *underspecified* if it has free occurrences of recursion variables (locations).

Note that (22) is an well-formed term of L_{ar}^{λ}, with free occurrences of the pure variables i and j, and free occurrences of the recursion variables h, p_1 and p_2. Thus, it is underspecified. The free variables h, p_1, p_2 can be specified, i.e. bound, by adding appropriate assignments in the scope of the where operator in the term (22). Binding can be done to result terms that do not have any free occurrences of recursion variables, and do not have any new constants or variables. Note that i and j have free occurrences in the above underspecified term (22). But, i and j are bound in the sub-terms $\lambda i\, p_1$ and $\lambda j\, p_2$, for which they do not carry the right indexing information for the arguments of $love(i, j)$ because $\models \lambda i\, p_1 = \lambda x\, p_1$ and $\models \lambda j\, p_2 = \lambda y\, p_2$, for any pure variables i, j, x and y. This so because $\mathsf{den}(\lambda i\, p_1)(g) = \mathsf{den}(\lambda x\, p_1)(g)$ and $\mathsf{den}(\lambda j\, p_2)(g) = \mathsf{den}(\lambda y\, p_2)(g)$ for any variable assignment g. Thus, $\lambda i\, p_1 \approx \lambda x\, p_1$ and $\lambda j\, p_2 \approx \lambda y\, p_2$.

The term (22) is over-underspecified (considered as a candidate for semantic representation of the sentence (20)), because (22) does not carry the information which is given by the grammatical structure of the sentence, in particular,

the indexing of the NP quantifiers. This indexing information is depicted by the underspecified graph (21) which specifies that the subscript index i of the quantifier $every(man)$ is the first argument of the predicate denoted by $love$, while the subscript j of $some(woman)$ fills the second argument slot of $love$. However, terms[3] representing the *de dicto* and *de re* readings of (20) can be obtained by appropriate assignments to the free locations h, p_1 and p_2: the *de dicto* by $h := q$, $p_1 := r$, $p_2 := p$; and the *de re* by $h := r$, $p_2 := q$, $p_1 := p$. The above issues about co-indexing the arguments of the verb with the quantifiers, and binding the free variables i and j, might be resolved by the syntactic grammar of NL by providing appropriate the render relation in the syntax-semantics interface between NL and L_{ar}^λ.

Another underspecified term which covers the underspecified meaning of (20) is the following:

(23) h where $\{q := \lambda j\, every(m, p_1(j)),$ (underspecified)

$\quad\quad\quad r := \lambda i\, some(w, p_2(i)),$

$\quad\quad\quad p := \lambda i'\, \lambda j'\, love(i', j'),\ m := \lambda x\, man(x),\ w := \lambda y\, woman(y)\}$

Terms representing the *de dicto* and *de re* readings can be obtained by binding the free locations h, p_1 and p_2 with the following specifying assignments in the scope of the where operator:

(24) h where $\{p_2 := \lambda x\, \lambda y\, love(x, y),\ p_1 := \lambda v\, r,\ h := q(u),$ (de dicto)

$\quad\quad\quad r := \lambda i\, some(w, p_2(i)),$ ("indexing")

$\quad\quad\quad q := \lambda j\, every(m, p_1(j)),$

$\quad\quad\quad p := \lambda i'\, \lambda j'\, love(i', j'),\ m := \lambda x\, man(x),\ w := \lambda y\, woman(y)\}$

(25) h where $\{p_1 := \lambda y\, \lambda x\, p(x, y),\ p_2 := \lambda v\, q,\ h := r(u),$ (de re)

$\quad\quad\quad q := \lambda j\, every(m, p_1(j)),$ ("indexing")

$\quad\quad\quad r := \lambda i\, some(w, p_2(i)),$

$\quad\quad\quad p := \lambda i'\, \lambda j'\, love(i', j'),\ m := \lambda x\, man(x),\ w := \lambda y\, woman(y)\}$

There are other specifications of the free locations h, p_1 and p_2 which give terms that do not correspond to any possible interpretation of the sentence (20). The term (23) represents closely what is common between the syntactic structures of the canonical forms of the typical explicit λ-terms representing the *de dicto* and *de re* readings of (20). For example, the term (23) specifies a location p for "storing" and providing "direct access" to the values of the entire function $\lambda i\, \lambda j\, love(i, j)$, while a location $l := love(i, j)$ would provide the value only of $\mathsf{den}(love(i, j))(g)$, i.e. the value for the particular $g(i)$ and $g(j)$. Also, the recursion variables q, r, p, m and w in the term (23) carry information for computing and "storing" all the values of the corresponding functions, including

---

[3] The term (22) carries close similarity to Montague's translation of the quantification operator $F_{10,n}$.

of the meanings of the quantifiers $every(man)$ and $some(woman)$ applied to "unspecified" properties $p_1(j)$ and $p_2(i)$, respectively.

The NL sentence[4] (26), like the previous example (20), has occurrences of quantifier NPs in the subject and complement positions. With it, we will see the typical structure of the canonical forms of all similar *de dicto* and *de re* renderings into terms of L_{ar}^λ, and also coordination of simple extensional (e.g., conjunctive) adjectives.

(26)   Every dog chases some white cat.

The terms (27) and (28) are the canonical forms (modulo congruence) obtained via reduction rules from the typical explicit terms rendering the *de dicto* and the *de re* readings of (26).

(27) h_0 where $\{h_0 := every(h_3, h_5),$ (de dicto)
$\qquad h_3 := \lambda x\, dog(x),\ h_5 := \lambda x\, some(h_7(x), h_4(x)),$
$\qquad h_7 := \lambda x\, \lambda y\, (p(y)\&q(y)),\ p := \lambda y\, white(y),\ q := \lambda y\, cat(y),$
$\qquad h_4 := \lambda x\, \lambda y\, chase(x, y)\}$

(28) h_0 where $\{h_0 := some(h_7, h_1),$ (de re)
$\qquad h_7 := \lambda y\, (p(y)\&q(y)),\ p := \lambda y\, white(y),\ q := \lambda y\, cat(y),$
$\qquad h_1 := \lambda y\, every(h_3(y), \lambda x\, h_4(x, y)),$
$\qquad h_3 := \lambda y\, \lambda x\, dog(x),\ h_4 := \lambda x\, \lambda y\, chase(x, y)\}$

Note that the terms assigned to h_7 in (27) and h_3 in (28) include additional λ-abstractions due to the λ-rule of reduction calculus. By the rules of the calculus of referential synonymy and the Referential Synonymy Theorem, the additional abstraction can be eliminated, because these sub-terms have the same denotations as terms without the extra-abstractions, and with which they are referentially synonymous. E.g., the terms (28) and (29) are referentially synonymous, i.e., they have the same meaning:

(29) h_0 where $\{h_0 := some(h_7, h_1),$ (de re)
$\qquad h_7 := \lambda y\, (p(y)\&q(y)),\ p := \lambda y\, white(y),\ q := \lambda y\, cat(y),$
$\qquad h_1 := \lambda y\, every(h_3', \lambda x\, h_4(x, y)),$
$\qquad h_3' := \lambda x\, dog(x),\ h_4 := \lambda x\, \lambda y\, chase(x, y)\}$

The underspecified meaning of the sentence (26) can be represented by the following graph:

(30)

$\bullet\ [h_1 : every(h_3, X)]_x \qquad\qquad \bullet\ [h_5 : some(h_7, Y)]_y$

$\bullet\ h_3 : \lambda x\, dog(x)\quad \bullet\ X \qquad \bullet\ h_7 : \lambda y\, (white(y)\&cat(y))\quad \bullet\ Y \qquad \bullet\ h_4 : chase(x, y)$

---

[4] This sentence is an example given in Copestake et al. ([1]).

An over-underspecified term with free recursion (location) variables, which covers the readings (27) and (28), and can be specified in other irrelevant to this sentence ways, is:

$$(31) \qquad h_0 \text{ where} \{ h_1 := \lambda y \, every(h_3, X), \qquad \text{(over-underspecified)}$$
$$h_5 := \lambda x \, some(h_7, Y),$$
$$h_3 := \lambda x \, dog(x), \; h_4 := \lambda x \, \lambda y \, chase(x, y),$$
$$h_7 := \lambda y \, (p(y) \& q(y)), \; p := \lambda y \, white(y), \; q := \lambda y \, cat(y) \}$$

The free recursion variables h_0, X and Y can be specified by various assignments, to result terms that have no free recursion variables and do not have any occurrences of new constants or variables. For example, the *de dicto* and *de re* terms can be obtained by adding the following assignments in the scope of the where operator of (31):

$$(32) \qquad h_0 := h_1(y), \qquad X := h_5, \qquad Y := \lambda y \, h_4(x, y), \qquad \text{for (27)}$$
$$(33) \qquad h_0 := h_5(x), \qquad X := \lambda x \, h_4(x, y), \qquad Y := h_1, \qquad \text{for (28).}$$

Similarly to (22), the term (31) is over-underspecified, but it caries information which (22) does not: The term (31) specifies a location h_4 for "storing" and providing "direct access" to the values of the entire function $\lambda x \, \lambda y \, chase(x, y)$. And it does not have the problem with pure free variables.

Terms similar to (31) are possible representations of sentences that combine quantifier scope underspecification with subject–complement underspecification. Each of the sentences (34), (35) and (36) in Bulgarian[5] (including other order combinations) has two possible subject–complement distributions of the NPs and corresponding co-indexing with the semantic argument roles of the verb. I.e., each of them can be translated into either of the English sentences (37a) and (37b). In addition to the index underspecification, each of these sentences is scope underspecified with respect to the NP quantifiers. The linear order of the NPs in the Bulgarian sentences determines the most common reading, but depending on the context, all combinations of interpretations are possible in these examples. The semantics and tense of the verb form in sentences with similar grammatical structure can eliminate some of the readings.

(34)	Vsyako	momche	shte tcelune	nyakoe	momiche
	Every	boy	will kiss	some	girl

(35)	Vsyako	momche	nyakoe	momiche	shte tcelune
	Every	boy	some	girl	will kiss

(36)	Nyakoe	momiche	vsyako	momche	shte tcelune

---

[5] The same phenomena exists in other languages. Similar examples were presented by my students in the Computational Semantics class, Autumn 2006 – Spring 2007.

Some	girl	every	boy	will kiss

(37a)   Every boy will kiss some girl.
(37b)   Some girl will kiss every boy.

5 The Conclusion: How to Define the *Render* Relation?

The referential intension $\mathsf{int}(A)$ of a term A of L_{ar}^λ is the abstract algorithm which computes its denotation $\mathsf{den}(A)$ depending on the variable assignment g. I.e., for every term A which has meaning[6], $\mathsf{int}(A)$ defines a function, denoted by $\overline{\mathsf{int}(A)}$, such that for every variable assignment g, $\overline{\mathsf{int}(A)}(g) = \mathsf{den}(A)(g)$. According to the Referential Synonymy Theorem, the abstract algorithm for computing the denotation $\mathsf{den}(A)$ is naturally represented by the canonical form of A, which is a term of the form: $\mathsf{cf}(A) = A_0$ where $\{p_1 := A_1, \ldots, p_n := A_n\}$, where $n \geq 0$, $p_1, \ldots, p_n$ are locations, and $A_0, \ldots, A_n$ are explicit and irreducible terms. $\mathsf{cf}(A)$ can be computed effectively by using the reduction rules. In this paper, I have formed the idea that NL sentences with scope ambiguities would be rendered into underspecified canonical forms, either directly by the render relation, or by applications of reduction rules to a rendering term.

Typically, for a given NL sentence that has multiple occurrences of quantifier expressions, there are alternative underspecified L_{ar}^λ terms, which are not referentially synonymous, and either of which can be used for obtaining all readings of that NL sentence. The syntactic theory of NL and the definition of the render relation should target rendering of NL expressions with scope ambiguities into most adequate underspecified L_{ar}^λ terms, so that (1) no information that is presented in the grammatical structure of the NL expression is lost (e.g., predicate argument indexing is represented), and (2) the underspecified term is computationally proper (e.g., only the weak form of β-reduction can be used, replacements of terms are free, etc.). Some of the alternative underspecified representations may carry more information than others from computational view.

The terms (27)–(28), the underspecified terms in the above examples and their *de dicto* and *de re* specifications are in canonical forms. Each variable assignment g, determines unique solution of the system of equations corresponding to the where parts of the canonical form of A. This implies that some of the plausible assignments g determine the scopes of the quantifiers in an underspecified term A which renders a sentence like (20) or (26). For example, for each given assignment g, the values $g(h_0)$, $g(X)$ and $g(Y)$, calculated by the term (31) extended by (32) (or by (33)), determine the denotations, and thus, the scopes of the quantifiers. The assignments specifying the free recursion variables in the extended equations define particular updates of the variable assignments g by the clause (D4) of the definition of the denotation function $\mathsf{den}(A)$. Intuitively, g can be considered as modelling context circumstances or speakers' references that decide scope ambiguities.

---

[6] Recall that immediate terms do not have meanings.

The language of acyclic recursion L_{ar}^λ with its calculus has very powerful computational features, e.g., effective calculation of the canonical forms and expressive adequateness. The clear mathematical properties of L_{ar}^λ are very promising for its applications to natural language semantics, which is work that is pending. The render relation appears as the best candidate for the syntax-semantics interface in the constraint-based lexicalized approaches to computational grammar. For example, Minimal Recursion Semantics in HPSG explicitly bears characteristics that can be formalized by the language of acyclic recursion.

References

1. Copestake, Ann, Dan Flickinger, Carl Pollard, and Ivan A. Sag. *Minimal Recursion Semantics: an Introduction.* Research on Language and Computation 3.4: (2006) 281–332.
2. Gallin, D. *Intensional and Higher-Order Modal Logic.* North-Holland. (1975)
3. Montague, R. *English as a formal language.* Linguaggi nella Società e nella Tecnica (Milan) (Bruno Visentini et al., editors), Edizioni di Comunità, (1970) 189-284. Reprinted in Montague (1974b)
4. Montague, R. *Pragmatics and intensional logic.* Synthèse, vol. 22, (1970) 68-94. Reprinted in Montague (1974b)
5. Montague, R. *Universal Grammar.* Theoria, vol. 36, (1970) 373-398. Reprinted in Montague (1974b)
6. Montague, R. *The Proper Treatment of Quantification in Ordinary English.* Approaches to Natural Language: Proceedings of the 1970 Stanford Workshop on Grammar and Semantics (J. Hintikka et al., editors), D. Reidel Publishing Co, Dordrecht, (1973) 221-224. Reprinted in Montague (1974b)
7. Montague, R. *Pragmatics and intensional logic.* Synthèse 22. (1974) 68-94. Reprinted in Thomason (1974) p 119-147.
8. Montague, R. *Formal philosophy.* Yale University Press. New Haven and London. Selected papers of Richard Montague, Edited by Richmond H. Thomason. (1974)
9. Moschovakis, Yiannis N. *A logical calculus of meaning and synonymy.* Linguistics and Philosophy, v. 29. (2006) pp. 27 – 89.
10. Muskens, Reinhard. *Meaning and Partiality.* CSLI Publications. (1995)
11. Sag, Ivan A., Thomas Wasow, and Emily M. Bender. *Syntactic Theory — A Formal Introduction.* 2nd Ed. Stanford: CSLI Publications. (2003)

A Montague-based Model of Generative Lexical Semantics

Bruno Mery, Christian Bassac, and Christian Retoré

SIGNES group, LaBRI, INRIA, ERSS
Université de Bordeaux
351 cours de la Libération
F-33405 Talence Cédex, France

Abstract. Computational semantics has long relied upon the Montague correspondance between syntax and semantics, which is not by itself well suited for the computing of some phenomena, such as logical polysemy, addressed by recent advances in lexical semantics. Our aim is to integrate the results of lexical semantics studies such as the Generative Lexicon Theory in a straightforward way with the existing computing of logical forms, in order to form a Montagovian framework for lexical semantics. In addition, we will outline a way to integrate other kinds of semantic information.

Computational semantics has long relied upon Montague's logical syntax to semantics system, or its variants. While the computing of logical forms from syntax is well known and has recently been formalized in type theory (e.g., by [Ranta, 2004]), the process is not fine-grained enough for a large part of language. Specifically, the Generative Lexicon Theory detailed in [Pustejovsky, 1995] and Transfers of Meanings introduced in [Nunberg, 1993] both present convincing data in favor of co-compositionality of meaning, the fact that the same lexical items often convey different yet related senses depending upon what other terms they are applied to. [Pustejovsky, 1995] also provides a complete description of the mechanisms of lexical semantics, sufficient to express the processes behind composition and co-composition.

Our goal is to present an efficient way to express those principles in a generic Montagovian computational semantics framework. In this paper, we first introduce the Generative Lexicon Theory, and explore the assumed link with Montague-style logical forms. We then detail our model and its variants, including how to extend it to other theories and phenomena, and finish with a discussion of the generality and possibilities of use of such a system.

1 The Generative Lexicon Theory

Pustejovsky's Generative Lexicon Theory (GL) gives a strongly-motivated model for many cases of logical polysemy, together with a rich structuration of the

meaning of concepts. It uses an inheritance-based hierarchy of types, each corresponding to a lexical concept. Each lexical entry (associated with each type) also includes:

- the number and types of arguments needed (for a predicate),
- the characterization of the event structure associated with the concept, if any, and
- the associated *qualia*, or modes of explanation of the concept: what its properties are (*formal*), what it is made of / part of (*constitutive*), what it can be used for (*telic*), what can cause it to come into being (*agentive*)... the idea being that a word can, under certain conditions, refer to any of its qualia (e.g., "ship" can be derived from "sail").

Some of this type of information might be *underspecified*, i.e. not defined in the lexicon, and filled in during the computation of the meaning of an utterance.

In addition, some lexical entries are of a so-called *complex* (or *dot*) type, expressing two or more *aspects* of different, hierarchically not comparable types (none is the subtype of the other). For instance, if one supposes that there are physical objects of type P and informational contents of type I, then the lexical item *book* would be of type $I \bullet P$.

2 Linking GL with Montague semantics

2.1 Compositional semantics

The compositional and co-compositional mechanisms introduced by GL presupposes that a very basic kind of structure is available, such as a syntactic tree. This structure indicates which terms are actually applied to which in an utterance, i.e., which are the arguments of which predicates.

The basic Montagovian notion of application and abstraction then applies, considering that the lexicon is responsible for the typing of the term (the argument structures defining types for the term and its arguments).

In certain situations where normal applications would result in a type clash, though, the other mechanisms of GL are employed, and it is understood that a certain number of *type coercion* operations are licensed :

- if a predicate needs an argument of a type α, and the actually selected argument is of type α', with α and α' compatible (i.e., one is a subtype of the other in the type hierarchy), then the application is valid by *type accommodation*;
- if a predicate needs an argument of a type α, and the actually selected argument is of type β, with α being part of a certain quale of β, then the application is valid by *exploitation* over said quale;
- if a predicate needs an argument of a type α, and the actually selected argument is of type $\alpha \bullet \beta$, then the application is valid by *$\bullet$-exploitation*.

2.2   Current formulations

Type-theoritical formalizations of these semantics, however, are far from trivial. The original theory, as well as more recent formalisms such as [Asher and Pustejovsky, 2005] or [Pustejovsky, 2006], fails to remain within a standard logical calculus.

This does not mean that these formulations are inadequate, but the fact is that the level of detail for some phenomena is such that it looks difficult to integrate literally all constraints expressed in any logic type system and keep it yet simple and sound. Therefore, those formalisms struggle with the necessity of expressing the results of long studies in lexical semantics and the need for actually useable logical rules.

2.3   Our approach

We do not want to model the entirety of GL in our typing system and logical rules. Our main concern being economy, we have thought that we might keep the information present in types as simple as possible, and add the necessary data carried by the lexicon on a term-to-term basis.

Thus, we take as a basis the already complete and sound Montague logic system, with simple types, and keep the same application and abstraction rules. The one exception is that, in addition of the classical logical term, each variable will be able to convey additional lexical information by providing additional, optional terms (involving additional logical constants)that might be used when type coercion is required. We shall detail this model in the next section. . .

3   A model with optional terms

3.1   Basic assumptions

Our model is based upon classical Montague semantics with simple types. It supposes a hierarchy similar to an ontology, such as described in [Pustejovsky, 2006], where $\top$ is the universal type with three main subtypes, *Entity*, *Event*, *Property*, the various subtypes of which form the complete type lattice.

We shall use the standard simply-typed λ-calculus with a slightly different use of the lexicon that enhances the application of a term to another in two different ways :

- *self-adaptation*: a lexical item might provide a number of optional terms that allow a type change when the classical application would yield a type error, and
- *selection-projection*: a predicate may select for an argument of an unspecified type and attempt *afterwards* to project a certain type upon it, using a distinct set of terms that the predicate and argument might both contribute to.

This system is derived from the idea, expressed in both [Nunberg, 1993] and [Pustejovsky, 1995], that each lexical entry can licence some type shifts or coercions. It does not change overmuch the general rules for Montague semantics – as the increase in expressive power is due to extra terms, rather than extra data in the typing system that would have to be captured by adequate rules.

The main point is that each lexical item contributes *at least* a term, that *must* be consumed (as per usual), plus a (finite) number of *optional* terms that might be used.

Each use of such a term might:

- change the type of the non-optional term, in order to solve some typing mismatch;
- change the associated structure of the term, for example by specifying a quale;
- change the term itself (number of arguments, etc.);
- enable future reference for the (newly created) concept in the discourse.

In the following, type accommodation is supposed, i.e., any given type shall share the property of its supertypes, including type-coercing terms.

3.2 Self-adaptation

Each instance x of some lexically-defined type τ might provide some additional terms. Operators $f_1 \ldots f_k$ are defined within τ, together with the main term (in GL terms, within the *argument structure*), and are of type $\tau \to \sigma_i$ for some $\sigma_i \neq \tau$. Thus, when needed, any $f_i(x)$ might be substituted to x.

For example, suppose that objects of type "computer", Ct, have an extensive *constitutive* quale, including at least an object of type "processing unit", CPU. Then we have an operator $f : Ct \to CPU$, which, when used on a computer x, would yield a pointer to its processor $f(x)$. Thus, supposing clock-related predicates apply only on objects of type CPU, we would have a derivation[1] :

Example 1. A 2-GHz computer
$\exists x, \lambda y^{CPU}.Clock(y) \ [x^{Ct}]$
$\exists x^{Ct}, \lambda y^{CPU}.Clock(y) \ [(f(x))^{CPU}]$

That mechanism suffices to licence most cases of qualia-exploitation (with an operator for every exploitation possible on the agent, telic or constituents), as well as some specific lexical rules such as *grinding*, where an object is subject to an irrevocable change, such as herb→food, which is modeled by a type-coercing operator, as in :

Example 2. Freshly prepared lemongrass
$\exists x, \lambda y^F.Fresh(y)[x^H]$
$\exists x^H, \lambda y^F.Fresh(y)[(f(x))^F]$

---

[1] In the following, arguments are enclosed in square brackets for the sake of clarity.

As every occurrence of the concerned object is changed, that use of type-coercing operator might be likened to *passing a variable by reference* in programming.

3.3   Projection after selection

Another group of optional terms might be provided, either by an argument or predicate, in order to modify the argument after its selection. Those operators $g_1 \ldots g_l$ are also lexically defined.

They might be used when the predicate is on the model of $\lambda x^\top . P(\Pi_\tau(x))$ (that is, the predicate P selects for an argument of any type and attempts to apply an optional term g_i afterwards, which would yield an output of type τ). One might think of $\Pi_\tau(x)$ as "x viewed as of type τ", and the construct results in a type clash if it fails.

As the semantic change of the argument is local to its selection by the concerned predicate, the object in itself remains intact, and thus, that process is like *passing a variable by value* in programming.

This mechanism can express the phenomenon known as *co-predication*: if two or more predicates select the same lexical item, they may enforce different types upon it. It happens frequently with *complex* types, i.e., objects with more than one aspect (such as *book* or *town*, where we would have a select-project operator available for every alternate type).

Recall that when *self-adaptation* is required, conversely, the item changes its type for *every occurrence*, and co-predication in sentences such as (3) is impossible :

Example 3. ?? The tuna we had yesterday night was lightning fast and delicious

For an exemple of type change after selection in a co-predicative sentence, we might have a type "town", Tn, a type "people", Pp, with an operator $g : Tn \to Pp$ that represent the relationship between a city and its inhabitants. Then we could have:

Example 4. Boston is a large city that mostly votes Democrat
$\exists x^{Tn} \; \lambda y^{Tn}.City(y)[x] \; \wedge \; \lambda z^\top . Vote(\Pi_{Pp}(z))[x]$
$\exists x^{Tn} \; City(x) \; \wedge \; Vote(\Pi_{Pp}(x^{Tn})$
$\exists x^{Tn} \; City(x) \; \wedge \; Vote(g(x))$

This change of type after selection, which enables to reference the newly selected aspect later on in the discourse, can be used to analyze some of the most complex quantificational puzzles in co-predicative sentences.

4   Expanding the framework beyond the lexicon

We believe our model to be sufficient to take into account most of the phenomena that are the target of GL. However, some additional areas could be further explored in the search for actual automated understanding of the language.

In this section, we will explain how our system could be extended in order to include some of those theoretical and practical points.

4.1   A possible implementation in functional programming

The simple co-compositional logical framework we have described here can very easily be implemented in functional programming such as Lisp or CaML, as functional application is the one operation needed, and the addition of optional terms corresponds to methods attached to the classes associated with the type of the variable in an object-oriented view. A translation module, which would extract the optional terms from the definitions of the types in GL, would also be quite straightforward.

4.2   Integrating multiple adjustments

The model outlined here is also quite generic in that it proposes simple adjustments for co-composition and coercion via the information encoded in each lexical entry, but can also be adapted to other kinds of linguistic and non-linguistic data. Thus, additional, optional terms might be deduced from:

- the current discourse (e.g., if someone is defined as a teacher, the associated lexical operators should be linked to the name of the person for future use),
- the situation (non-verbal signs might trigger transfers),
- some cultural assumption (the lexical definition of *village* might contain very different constituents depending on the cultural group),
- or additional pragmatic reasoning.

The system in itself remains the same, it is a simple matter of adding optional terms corresponding to the different aspects wanted, while keeping a reasonable total number of possible combinations. Of particular interest would be a translation of SDRT and λ-DRT in that framework.

4.3   Metaphors and idiomatic expressions

There are two ways to account for idiomatic expressions and metaphors specific to a given language or dialect.

The most obvious is to list, for each language or fragment, every such expression and use, and to associate additional terms that would map the entire expression (as a fixed string) into its logical equivalent. This approach would be costly, both in establishing a complete lexicon containing all such information and in terms of complexity, but it would probably be useful for complete understanding and necessary for machine-driven translation.

The other approach, more economic and possibly more efficient in a day-to-day basis, is to consider that each (valid syntactic-wise) utterance is semantically correct, and to try and derive a plausible, if incomplete, interpretation. This

method would introduce unspecified operators as a last resort in a type clash, and the resulting sense might later be clarified through machine-assisted dialogue.

In any case, taking into account idiomatic expressions will prove necessary in order to grasp the correct meaning of many common sentences.

4.4   Interpretation choice and scoring

During the computation of the actual meaning of a given logical form, when several operators f_i, g_i are available for use, several defeasible interpretations might be possible. A specialized module might have to *choose* between the possibilities: supposing a "town" can be either a geographical entity, the set of its inhabitants, the ruling body (either mayor or council), or the set of its civil servants, then in

Example 5. Philadelphia wants a new bridge, but the mayor is opposing it

the prominent typing (geographical entity) should be rejected, as well as the type change associated to "mayor", but any other typing operators associated to subtypes of "people" can be valid interpretations.

It is also often the case that several interpretations remain available when everything is taken into account. To help classify interpretations, a notion of "semantic cost" might be added to each type-coercing operator, and a minimal cost would indicate higher likeliness of the final logical form (even as any of the other ones could still be considered as "correct").

4.5   Generative power and complexity

As this model is a front-end to semantic and pragmatic theories rather than a generative grammar in itself, it presupposes some kind of syntactic formalism in order to generate the bare logical form (i.e., what lexical items are arguments of one another). GL, as well as most other theories that we might model, will not by itself change the generative capacity of that syntactic formalism, being intended primarily for analysis and understanding.

The semantic power, i.e., the number of correct interpretations for the same syntactically valid utterances, however, will be greater. The extent of the semantic power of the overall formalism depends upon the number of choices added by the available optional terms, and this also very much affects the computational complexity of any implementation: for the process of interpretation to remain feasible, the number of available operators to any type and, most importantly, of arguments to any predicate, should be bounded – and tightly so. The general complexity of the computing of interpretations for any given interpretations should remain, at most, polynomial in space and time in order to be useful.

In the real world, heuristics shall be employed in order to check only the most likely derivations and to use presuppositions in order to help ensure the efficiency

of the process. Moreover, some moderate use of underspecification and symbolic synthetical reasoning (not deriving every ambiguous reading at each step of the calculus) should be helpful in the establishment of a balanced strategy for actual computing, for which further research is needed.

Conclusion

The model outlined herein is a simple way to implement both GL and other theories of the modifications to the semantics of lexical items. We think that a complete formalization is not so difficult to attain, and that it should at the least be an efficient formulation of GL, with possible practical applications.

References

[Asher and Pustejovsky, 2005] Asher, N. and Pustejovsky, J. (2005). Word Meaning and Commonsense Metaphysics. Semantics Archive.

[Nunberg, 1993] Nunberg, G. (1993). Transfers of meaning. In *Proceedings of the 31st annual meeting on Association for Computational Linguistics*, pages 191–192, Morristown, NJ, USA. Association for Computational Linguistics.

[Pustejovsky, 1995] Pustejovsky, J. (1995). *The Generative Lexicon*. MIT Press.

[Pustejovsky, 2006] Pustejovsky, J. (2006). Type Theory and Lexical Decomposition. Semantics Archive.

[Ranta, 2004] Ranta, A. (2004). Computational semantics in type theory. *Mathématiques et Sciences Sociales*, 165:31–57.

The Semantics of Intensionalization[*]

Gilad Ben-Avi[1] and Yoad Winter[2][**]

[1] Technion, `bagilad@cs.technion.ac.il`
[2] Technion/NIAS, `winter@cs.technion.ac.il`

Abstract. This paper introduces a procedure that takes a simple version of extensional semantics and generates from it an equivalent possible-world semantics that is suitable for treating intensional phenomena in natural language. This process of *intensionalization* allows to treat intensional phenomena as stemming exclusively from the lexical meaning of words like *believe*, *need* or *fake*. We illustrate the proposed intensionalization technique using an extensional toy fragment. This fragment is used to show that independently motivated extensional mechanisms for scope shifting and verb-object composition, once properly intensionalized, are strictly speaking responsible for certain intensional effects, including *de dicto/de re* ambiguities and coordinations containing intensional transitive verbs. While such extensional-intensional relations have often been assumed in the literature, the present paper offers a formal sense for this claim, facilitating the dissociation between extensional semantics and intensional semantics.

1 Introduction

The simplicity and elegance of extensional higher-order logics make them attractive for treating many phenomena in natural language. The arguments for intensional (and hyper-intensional) semantics are of course compelling, but we would not like these considerations to complicate the analysis of properly-extensional phenomena. Unfortunately this is often the case, and especially in Montague's classical treatment in [1] (PTQ). In order to address this tension between extensional semantics and intensional semantics, this paper studies the relations between elementary extensional semantics and intensional semantics such as

Montague's IL or Gallin's Ty2 ([2]). We propose a general process of *intension-alization* that maps an extensional framework to such an intensional framework, and illustrate its architectural benefits using a toy fragment. More generally, we argue that also in other frameworks there are methodological and empirical reasons for taking intensionalization procedures to be a central part of the study of intensional phenomena.

The distinction between parts of a language that exhibit intensional effects and parts that do not can often be reduced to a simple distinction between two kinds of lexical items: those that create an intensional context and those that do not. In this paper, expressions like the verbs *seek*, *need* and *believe* and the adjective *fake*, which create an intensional context are called *intension-sensitive*. Expressions that do not create an intensional context, such as the verb *kiss* or the adjective *red*, are called *intension-insensitive*. We assume that an extensional semantics is sufficiently adequate for expressions that consist solely of intension-insensitive lexical items, while an intensional semantics is only needed for expressions with intension-sensitive lexical items.

In this paper we propose a modular approach to the architecture of intensional systems that is based on this assumption. We start out by introducing a simple grammatical framework with a standard extensional semantics, and then add intension-sensitive words to the lexicon. Since the intension-sensitive lexical items (semantically) select intensional objects, the extensional types and meanings of the intension-insensitive lexical items need to be shifted to intensional types and meanings. Following [3], we refer to this shifting process as *intensionalization*.

The strategy of intensionalization that we suggest can be traced back to the type shifting strategy of [4]. But, as far as we know, the only full-fledged intensionalization procedure in the literature, even if not under this label, was defined by Keenan and Faltz in [5]. A simple intensionalization procedure can be inferred from the introduction of intensional semantics in [6] (Ch. 12). This intensionalization procedure is similar to the "intensionality monad" that Shan defines in [7]. Shan partly follows an early version of [8], who uses a comparable architectural approach for treating natural language quantifiers using the notion of *continuations*.

The novel feature of our intensionalization procedure, as opposed to previous proposals, is that the shifting process that we employ from extensional types and meanings to intensional types and meanings is a general one, and applies uniformly to all syntactic categories and semantic types. Beside types and meanings of expressions, the intensionalization process changes very little in the extensional system. For example, *de dicto/de re* ambiguities and coordinations of intension-sensitive with intension-insensitive transitive verbs are treated as manifestations of purely extensional mechanisms in their intensionalized guise.

One central formal aspect of the intensionalization process is truth-conditional *soundness*. In order to preserve the insights of an extensional semantics, we need to guarantee that its intensionalized version is descriptively equivalent to it. In more exact terms: both the extensional semantics and its

image under intensionalization should provably describe the same entailment relations between sentences. This paper sets the background for a soundness theorem, which is stated for the extensional framework we define. The details of the proof appear in [9].

The paper is organized as follows. Section 2 defines a setting for an extensional grammar, and illustrates it using a toy grammar. The intensionalization process is the subject of Section 3, which also states its soundness and demonstrates its application to the toy grammar of Section 2. Section 4 demonstrates how intension-sensitive lexical items are added to the intensionalized grammar, and illustrates the resulting grammatical interactions between such items and intension-insensitive expressions and extensional mechanisms.

2   Formalizing the extensional setting

To formalize an intensionalization procedure we first need to make explicit assumptions about the extensional semantics. We define a simple undirected typelogical grammar that includes the bare semantic details necessary for defining the intensionalization procedure. Syntactic, pragmatic and phonological details are ignored.

2.1   Extensional setting

The *extensional grammars* that we assume are of the form $\langle \Sigma, \mathbf{type}, \{\!\!\{\cdot\}\!\!\} \rangle$. Σ is the *lexicon* ('alphabet') – a finite set of *words* ('lexical items', 'terminal symbols'). Every word $\alpha \in \Sigma$ is assigned an *extensional type* $\mathbf{type}(\alpha)$. The extensional types that we assume are the standard functional types over basic types e and t. The set of all extensional types is denoted by $\mathcal{T}_{\mathrm{ex}}$. The role of the third component of a grammar is to determine the possible interpretations for each word, as described below.

Lexical items are directly interpreted by models. Standardly, a *model* $\mathcal{M}$ is a pair $\langle \mathcal{F}_{\mathcal{M}}, \mathcal{I}_{\mathcal{M}} \rangle$, where $\mathcal{F}_{\mathcal{M}}$ is an *extensional frame* and $\mathcal{I}_{\mathcal{M}}$ is an *interpretation function*. The *extensional frame* is a collection of *domains* D_{σ} for every extensional type σ. Standardly, we assume that D_t is the set $\{0, 1\}$ of *truth values*, D_e is an arbitrary nonempty set E of *entities*, and $D_{(\tau\sigma)}$ is the set $D_{\sigma}^{D_{\tau}}$ of all functions from D_{τ} to D_{σ}. Thus, an extensional frame is uniquely determined by the set E of entities. For every E we refer to the induced frame as the *E-based (extensional) frame*. The *interpretation function* $\mathcal{I}_{\mathcal{M}}$ maps every word α to a member of $D_{\mathbf{type}(\alpha)} \in \mathcal{F}_{\mathcal{M}}$.

To derive interpreted sentences from interpreted lexical items, let us add a simple notion of a grammar, keeping in mind that, as before, we only introduce the bare semantic notions that are necessary in order to define an intensionalization process in a most general way. A derivation is either a lexical item (rule (1) below) or a compound derivation. Compound derivations are obtained from simpler derivations by one of two rules: *functional application* (rule (2) below) or *conjunction* (rule (3) below).

(1)   a. Every lexical item α is a derivation of type $\mathbf{type}(\alpha)$ of the expression α.
      b. For every model $\mathcal{M}$: $[\![\alpha]\!]^{\mathcal{M}} = \mathcal{I}_{\mathcal{M}}(\alpha)$.

(2)   a. If Δ_1 is a derivation of type $(\sigma\tau)$ of an expression ε_1 and Δ_2 is a deriva-
         tion of type σ of an expression ε_2, then $[\Delta_1\ \Delta_2]$ (respectively, $[\Delta_2\ \Delta_1]$)
         is a derivation of the expression $\varepsilon_1\ \varepsilon_2$ (respectively, $\varepsilon_2\ \varepsilon_1$) of type τ.
      b. For every model $\mathcal{M}$: $[\![[\Delta_1\ \Delta_2]]\!]^{\mathcal{M}} = [\![[\Delta_2\ \Delta_1]]\!]^{\mathcal{M}} = [\![\Delta_1]\!]^{\mathcal{M}}([\![\Delta_2]\!]^{\mathcal{M}})$.

Regarding the conjunction rule, one of our main concerns is to enable a coordination of intension-insensitive words with intension-sensitive words, like in *Mary sought, found and ate a fish.* The syncategorematic introduction of conjunction that we use here is convenient for the sake of exposition of our intensionalization procedure, as it does not require adding a polymorphic entry to the lexicon or using several entries for an arbitrary number of types. Similar derivation rules can be formulated for other Boolean words such as *or* and *not.* For the purposes of this paper, however, it is enough to restrict attention to the conjunctive *and.* In order for two expressions to be conjoinable, they must be of the same type. Furthermore, this type must be *Boolean* (or *t-ending*). In a type system that contains the basic type t (e.g., our extensional types $\mathcal{T}_{\mathrm{ex}}$), a type σ is called *Boolean* iff either $\sigma = t$ or $\sigma = (\sigma_1\sigma_2)$ for a Boolean type σ_2. For the semantic composition in the conjunctive rule (3) we use the well-known *Generalized Conjunction* operator from [4], denoted here '$\sqcap$'.

(3)   a. If Δ_1 is a derivation of an expression ε_1 and Δ_2 is a derivation of an
         expression ε_2, both of a Boolean type σ, then $[\Delta_1\ \text{and}\ \Delta_2]$ is a derivation
         of type σ of the expression $\varepsilon_1\ and\ \varepsilon_2$.
      b. For every intended model $\mathcal{M}$, $[\![[\Delta_1\ \text{and}\ \Delta_2]]\!]^{\mathcal{M}} = \sqcap_{\sigma(\sigma\sigma)}([\![\Delta_1]\!]^{\mathcal{M}})([\![\Delta_2]\!]^{\mathcal{M}})$,
         where '$\sqcap_{\sigma(\sigma\sigma)}$' is recursively defined for Boolean types σ as follows:

$$\sqcap_{\sigma(\sigma\sigma)} = \begin{cases} \wedge \text{ (standard propositional conjunction)} & \sigma = t \\ \lambda X_{\sigma}\lambda Y_{\sigma}\lambda Z_{\sigma_1}.\sqcap_{\sigma_2(\sigma_2\sigma_2)}(X(Z))(Y(Z)) & \sigma = (\sigma_1\sigma_2) \end{cases}$$

Among all possible models for a grammar $G = \langle \Sigma, \mathbf{type}, \{\!|\cdot|\!\} \rangle$ we are interested only in those *intended* models that respect lexical restrictions on the meaning of individual words. This is obtained using the an operator $\{\!|\cdot|\!\}$ that assigns to every word $\alpha \in \Sigma$ a functional $\{\!|\alpha|\!\}$ that maps a frame $\mathcal{F}$ to the subset $\{\!|\alpha|\!\}^{\mathcal{F}}$ of $\bigcup \mathcal{F}$ that consists of all admissible interpretations for α. Formally, a model $\mathcal{M} = \langle \mathcal{F}_{\mathcal{M}}, \mathcal{I}_{\mathcal{M}} \rangle$ is an *intended model* for a grammar $G = \langle \Sigma, \mathbf{type}, \{\!|\cdot|\!\} \rangle$ iff $\mathcal{I}_{\mathcal{M}}(\alpha) \in \{\!|\alpha|\!\}^{\mathcal{F}_{\mathcal{M}}}$ for every $\alpha \in \Sigma$. The class of all intended models for a grammar G is denoted $\mathfrak{M}_G$.

For simplicity, we henceforth assume that for every lexical item α either (4) or (5) holds.

(4) For every frame $\mathcal{F}$: $\{\!|\alpha|\!\}^{\mathcal{F}} = D_{\mathbf{type}(\alpha)} \in \mathcal{F}$.
(5) For every frame $\mathcal{F}$ there is $\varphi \in D_{\mathbf{type}(\alpha)} \in \mathcal{F}$ such that $\{\!|\alpha|\!\}^{\mathcal{F}} = \{\varphi\}$.

Whenever (4) (respectively, (5)) holds for a lexical item α we will say that α is a *nonlogical constant* (respectively, *logical constant*).[3]

---

[3] There are at least two other kinds of possible logical restrictions on the meaning of lexical items. First, there are lexical items α for which $\{\!|\alpha|\!\}^{\mathcal{F}}$ is a non-singleton

2.2   Example

As an example consider the simple grammar in Table 1. The determiners *every* and *a* are standardly treated as logical constants. For convenience, the single function that a logical constant is assumed to denote is described by a lambda-term in the 4th column. On the other hand, nouns (both proper and common) and (in)transitive verbs, as well as other expressions of open lexical categories, are standardly treated as nonlogical constants. *Intersective* adjectives like *bald* or *red* are also treated as nonlogical constants of type (et). For the sake of the example, we assume that all intension-insensitive adjectives are intersective.

Table 1. Extensional lexical entries

word α	Type	$\{\!\{\alpha\}\!\}^{\mathcal{F}}$	λ-term
Mary, John	e	D_e	
red, bald	et	D_{et}	
king, queen	et	D_{et}	
smile, jump	et	D_{et}	
kiss, eat	$e(et)$	$D_{e(et)}$	
every	$(et)((et)t)$	$\{\textsc{every}\}$	$\textsc{every} = \lambda A^{et}\lambda B^{et}.\forall x^e[A(x) \rightarrow B(x)]$
a	$(et)((et)t)$	$\{\textsc{some}\}$	$\textsc{some} = \lambda A^{et}\lambda B^{et}.\exists x^e[A(x) \wedge B(x)]$

The toy grammar in (1)-(3), together with the lexical entries in Table 1, does not allow to derive all grammatical strings over the given lexicon. For instance, they do not allow to derive a transitive sentence like (6) below.

(6) A queen kissed every king.

Similarly, intersective modification with adjectives (e.g. *bald king*) and coordination of proper names with quantifiers (e.g., *Mary and every queen*) are not treated by the assumptions introduced so far. In order to deal with such examples without complicating too much the introduction of our proposed grammatical architecture, we use some phonologically-silent lexical items. These phonologically-silent lexical items (also called *empty words* or *operators*) are introduced in Table 2 as an *ad hoc* extension of the lexicon from Table 1. Note that all operators in this extension are treated as logical constants.

---

proper subset of the domain of α's type. An example for such a lexical item is the TV *follow*. To account for the entailment from (ia) below to (ib), it is reasonable to assume that *follow* is interpreted as an arbitrary *transitive relation* in $D_{e(et)}$.

(i)   a. Jack follows Cole and Cole follows Jessica.
        b. Jack follows Jessica.

Second, there are lexical items for which $\{\!\{\alpha\}\!\}^{\mathcal{F}}$ is possibly the whole domain of its type, but there are restrictions on the relation between α's interpretation and the interpretation of other lexical items (e.g., *kill* and *die*).

Table 2. Extending the lexicon from Table 1 with empty words as type shifting operators.

word α	Type	$\{\!\{\alpha\}\!\}^{\mathcal{F}}$	λ-term
ϵ_{ONS}	$(e(et))(((et)t)(et))$	$\{\textbf{ONS}\}$	$\lambda R^{e(et)}\lambda F^{(et)t}\lambda x^e.F(\lambda y^e.R(y)(x))$ mapping a binary predicate between entities to a binary predicate between entities and quantifiers (the quantifier taking narrow scope)
ϵ_{OWS}	$(((et)t)(et))$ $(((et)t)(((et)t)t))$	$\{\textbf{OWS}\}$	$\lambda R^{(((et)t)(et))}\lambda F^{(et)t}\lambda Q^{(et)t}.$ $F(\lambda y^e.Q(\lambda x^e.R(\lambda A^{et}.A(y))(x)))$ mapping a binary predicate between entities and quantifiers to a binary predicate between quantifiers (the object quantifier taking wide scope)
ϵ_{lift}	$e((et)t)$	$\{\textbf{LIFT}\}$	$\lambda x^e\lambda A^{et}.A(x)$ lifting an entity to a quantifier
ϵ_{adj}	$(et)((et)(et))$	$\{\textbf{ADJ}\}$	$\lambda A^{et}\lambda B^{et}\lambda x^e.A(x)\wedge B(x)$ mapping a set to an intersective modifier

The operation of the empty words ϵ_{ONS} and ϵ_{OWS} can be demonstrated with sentence (6) above. This sentence can be derived in two different ways. In both derivations, (7a) and (8a), the operator ϵ_{ONS} is applied to the TV *kissed*. The difference is that derivation (8a) also uses the operator ϵ_{OWS}. In an intended model $\mathcal{M}$, derivation (7a) denotes the *object narrow scope* (ONS) interpretation (7b), and derivation (8a) denotes the *object wide scope* (OWS) interpretation (8b). To facilitate readability, we henceforth let **word** (in boldface) stand for $\mathcal{I}_{\mathcal{M}}(word)$ whenever *word* is a lexical item, $\mathcal{M}$ is an arbitrary model, and there is no other model in the context.

(7) a. $[[a\ queen]\ [[\epsilon_{\text{ONS}}\ kissed]\ [every\ king]]]$

    b. $\exists x^e[\textbf{queen}(x)\wedge\forall y^e[\textbf{king}(y)\rightarrow\textbf{kiss}(y)(x)]]$

(8) a. $[[a\ queen]\ [[\epsilon_{\text{OWS}}\ [\epsilon_{\text{ONS}}\ kissed]]\ [every\ king]]]$

    b. $\forall y^e[\textbf{king}(y)\rightarrow\exists x^e[\textbf{queen}(x)\wedge\textbf{kiss}(y)(x)]]$

This use of (extensional) operators on predicates in order to derive ONS and OWS analyses essentially follows the (intensional) operators proposed in [10].

The empty word ϵ_{adj} is used to shift the set denoted by an intension-insensitive adjective to an intersective function of type $(et)(et)$. The latter can modify the set denoted by a common noun in the usual way.

The empty word ϵ_{lift} allows a proper noun like *Mary* to be of the same type as that of a quantified noun phrase like *every queen*. Such a typing is necessary for the treatment of coordinations like in *Mary and every queen smiled*.

3   A sound intensionalization procedure

Having defined a setting for an extensional semantics, we can now introduce our proposed intensionalization procedure for this setting and discuss its implications.

   To enable the introduction of intension-sensitive lexical items we should let their arguments denote intensional objects. For example, it is well known that for a reasonable analysis of a sentence like *Mary sought a unicorn* the noun phrase *a unicorn* should not have an extensional meaning of the kind that was introduced in Section 2. In this section we introduce our proposed semantics of *intensionalization*, by which the extensional types and meanings of lexical items in an extensional grammar can be modified, so that the resulting grammar is equivalent to the original extensional grammar, and at the same time can be extended to a properly intensional grammar by only adding intension-sensitive items to its lexicon.

3.1   Intensionalization

The *intensional types* that we assume are the functional types over e, s and t. The set of all intensional types is denoted by $\mathcal{T}_{\mathrm{in}}$. *Intensional frames* are defined similarly to extensional frames. But while an extensional frame is determined by a nonempty set E of entities as the domain of type e, an intensional frame is determined both by such a set E *and* a nonempty set W of *possible worlds* as the domain of type s. For a fixed choice of such E and W, the induced intensional frame is called the E, W-*based frame*. An *intensional grammar* is defined like an extensional grammar as a triple $\langle \Sigma, \mathbf{type}, \{\!|\cdot|\!\} \rangle$, where for every $\alpha \in \Sigma$: $\mathbf{type}(\alpha) \in \mathcal{T}_{\mathrm{in}}$, and $\{\!|\alpha|\!\}$ maps every intensional frame $\mathcal{F}$ to a subset $\{\!|\alpha|\!\}^{\mathcal{F}}$ of $D_{\mathbf{type}(\alpha)}$. The derivation rules remain the same.

   The intensionalization that we propose follows Van Benthem's typing recipe in [3]. Formally, the intensionalization procedure is defined as a mapping $\mathcal{I}$ from extensional grammars to intensional grammars. Given an extensional grammar $G = \langle \Sigma, \mathbf{type}_G, \{\!|\cdot|\!\}_G \rangle$, its intensionalization $\mathcal{I}(G) = \langle \Sigma, \mathbf{type}_{\mathcal{I}(G)}, \{\!|\cdot|\!\}_{\mathcal{I}(G)} \rangle$ is obtained by a systematic modification of the typing function $\mathbf{type}_G$ and the meaning functional $\{\!|\cdot|\!\}_G$. Following Van Benthem, we modify an extensional type σ by substituting (st) for every occurrence of t. The resulting intensional type is denoted $\ulcorner \sigma \urcorner$. Thus, for every $\alpha \in \Sigma$ we define $\mathbf{type}_{\mathcal{I}(G)}(\alpha) = \ulcorner \mathbf{type}_G(\alpha) \urcorner$.

   Using this global type-change recipe, we should now intensionalize the *meanings* of lexical items so that $\mathcal{I}(G)$ is equivalent to G. To facilitate the intensionalization of meanings, we made the simplifying assumption that each lexical item is either a logical or a nonlogical constant. With this assumption, the intensionalization of meanings is defined so that (non)logical constants in G remain (non)logical constants also in $\mathcal{I}(G)$. The question of how to treat other kinds of lexical items may be more complicated, and is left for future research.[4]

---

[4] Makoto Kanazawa (p.c.), based on a discussion with Philippe de Groote and Reinhard Muskens, suggests a modification of our intensionalization procedure that

The case of nonlogical constant is simple. If α is a nonlogical constant in G, we define $\{\!\{\alpha\}\!\}_{\mathcal{I}(G)}$ to be the functional that maps every intensional frame $\mathcal{F}$ to the domain $D_{\mathbf{type}_{\mathcal{I}(G)}(\alpha)}$ in $\mathcal{F}$. For a logical constant α in G to remain logical constant also in $\mathcal{I}(G)$, we need to define $\{\!\{\alpha\}\!\}_{\mathcal{I}(G)}$ in such a way that for every E, W-based intensional frame $\mathcal{F}$ there is an object $g \in D_{\mathbf{type}_{\mathcal{I}(G)}(\alpha)}$ such that $\{\!\{\alpha\}\!\}_{\mathcal{I}(G)}^{\mathcal{F}} = \{g\}$. It is expected that this object g is systematically derived from the unique interpretation of α in the corresponding E-based extensional frame $\mathcal{F}'$ using some mapping $L(\cdot)$ from $D_{\mathbf{type}_G(\alpha)} \in \mathcal{F}'$ to $D_{\mathbf{type}_{\mathcal{I}(G)}(\alpha)} \in \mathcal{F}$.

To motivate our proposed definition of this mapping, consider for example the determiner *every* as appearing in the extensional lexicon from Table 1. For this determiner, we need to map the object $\text{EVERY} \stackrel{def}{=} \lambda A^{et} \lambda B^{et}.\forall x^e[A(x) \to B(x)]$ in $D_{(et)((et)t)}$ to a unique member of the intensional domain $D_{\ulcorner(et)((et)t)\urcorner}$. The intensional denotation that we are after is similar to the denotation of *every* in PTQ.[5] This is the function that when applying to two properties P and Q, returns the proposition that is *true* in a world w just in case the predicate *extensions* in w of P and Q satisfy the containment requirement of EVERY. In symbols, we would like to end up with $L(\text{EVERY}) = \lambda w^s \lambda P^{e(st)} \lambda Q^{e(st)}.\text{EVERY}(P^w)(Q^w)$, where P^w is $\lambda x^e.P(w)(x)$ – the extension of the property P in a given index w – and similarly for Q^w.

To generalize this relatively simple example to any logical constant of any type, we first define an *extensionalization mapping* from intensional domains to extensional domains (Definition 3). We then use this notion in Definition 4 of the intensionalization mapping. To facilitate the definition of these mappings, we follow a tentative proposal in [3], and restrict our attention to the *quasi-relational* types of [11]. Definition 2 below of the quasi-relational types refers to the set $\mathcal{T}_e$ of *e-based types*.

Definition 1 (e-based types). *The set $\mathcal{T}_e$ of e-based types is the smallest set that satisfies:*

1. *$e \in \mathcal{T}_e$, and*
2. *If $\sigma_1, \sigma_2 \in \mathcal{T}_e$ then $(\sigma_1 \sigma_2) \in \mathcal{T}_e$.*

Thus, the e-based types are simply the standard functional types over a single primitive type e of entities. A *quasi-relational* type is a Boolean type in which every argument is either quasi-relational or e-based. Formally:

Definition 2 (Quasi-relational types). *The set $\mathcal{T}_{\mathrm{qr}} \subset \mathcal{T}_{\mathrm{ex}}$ of quasi-relational types is the smallest set that satisfies:*

1. *$t \in \mathcal{T}_{\mathrm{qr}}$, and*

---

does not need to stipulate different treatments for logical constants and non-logical constants. Furthermore, Kanazawa et al's proposal may be preferable to ours in some other important respects. We are currently studying the implications of their proposal.

[5] Determiners are introduced syncategorematically in [1], whereas here they are part of the lexicon. But this hardly matters for the semantic analysis.

2. *If both $\sigma_1 \in \mathcal{T}_e \cup \mathcal{T}_{\mathrm{qr}}$ and $\sigma_2 \in \mathcal{T}_{\mathrm{qr}}$, then $(\sigma_1\sigma_2) \in \mathcal{T}_{\mathrm{qr}}$.*

Note that the domain D_σ of a quasi-relational type $\sigma = (\sigma_1 \cdots (\sigma_n t) \cdots)$ is isomorphic to the cartesian product $D_{\sigma_1} \times \cdots \times D_{\sigma_n}$.

The definition of a w-extension of an object is as follows.

Definition 3 (w-extension). *Let $\mathcal{F}$ be an E, W-based intensional frame and $\mathcal{F}'$ the corresponding E-based extensional frame. Let $w \in W$, and $g \in D_{\ulcorner\sigma\urcorner} \in \mathcal{F}$ for some $\sigma \in \mathcal{T}_{\mathrm{qr}} \cup \mathcal{T}_e$. The w-extension of g is the object $g^w \in D_\sigma \in \mathcal{F}'$ that satisfies:*

1. *If $\sigma \in \mathcal{T}_e$ then $g^w = g$;*
2. *if $\sigma = t$ then $g^w = g(w)$;*
3. *if $\sigma = (\sigma_1 \cdots (\sigma_n t) \cdots)$, $n \geq 1$, then*

$$g^w = \lambda x_1^{\sigma_1} \cdots \lambda x_n^{\sigma_n}.\exists z_1 \cdots \exists z_n. \bigwedge_{i=1}^{n} ((z_i)^w = x_i) \wedge g(z_1) \cdots (z_n)(w)$$

In words, a tuple $\langle x_1, \ldots, x_n \rangle$ is in the w-extension of an intensional relation g, iff there is a tuple $\langle z_1, \ldots, z_n, w \rangle$ in g such that the w-extensions of the z_is are the x_is, respectively.

The intensionalization mapping $L(\cdot)$ is now defined as follows.

Definition 4 (Intensionalization mapping). *Let $\mathcal{F}$ be an E, W-based intensional frame and $\mathcal{F}'$ the corresponding E-based extensional frame. Let $f \in D_\sigma \in \mathcal{F}'$ for some $\sigma \in \mathcal{T}_{\mathrm{qr}} \cup \mathcal{T}_e$. The intensionalization of f is the object $L(f) \in D_{\ulcorner\sigma\urcorner} \in \mathcal{F}$ that satisfies:*

1. *if $\sigma \in \mathcal{T}_e$ then $L(f) = f$;*
2. *if $\sigma = (\sigma_1 \cdots (\sigma_n t) \cdots)$, $n \geq 0$, then*

$$L(f) = \lambda x_1^{\ulcorner\sigma_1\urcorner} \cdots \lambda x_n^{\ulcorner\sigma_n\urcorner} \lambda w^s.f((x_1)^w) \cdots ((x_n)^w)$$

In words, a tuple $\langle x_1, \ldots, x_n, w \rangle$ is in the intensionalization of a relation f, iff the w-extensions of $x_1, \ldots, x_n$ are in f.

We are now ready to define the intensionalization of the meanings of logical constants, which completes the definition of the intensionalization process. Let α be a logical constant in an extensional grammar G. For every intensional E, W-based frame $\mathcal{F}$ we define $\{\!|\alpha|\!\}_{\mathcal{I}(G)}^{\mathcal{F}} = \{L(f)\}$, where f is the single element in $\{\!|\alpha|\!\}_G^{\mathcal{F}'}$ for the extensional E-based frame $\mathcal{F}'$.

We claim that the intensionalization process that we have just defined is *sound*, in the sense that it preserves entailment between derivations of sentences. Entailment over an extensional grammar G is defined in the usual way: Δ_1 *extensionally entails* Δ_2 iff for every intended model $\mathcal{M} \in \mathfrak{M}_G$: $[\![\Delta_1]\!]^{\mathcal{M}} \leq [\![\Delta_2]\!]^{\mathcal{M}}$. Over an intensional grammar G, entailment is defined as a relation between derivations of type st: Δ_1 *intensionally entails* Δ_2 iff for every intended model $\mathcal{M} \in \mathfrak{M}_G$ and for every $w \in D_s$: $[\![\Delta_1]\!]^{\mathcal{M}}(w) \leq [\![\Delta_2]\!]^{\mathcal{M}}(w)$.

The soundness of the intensionalization process is formally stated in Theorem 5 below. The proof appears in [9]. For the proof we assume that whenever a nonlogical constant has a quasi-relational type, all its arguments are of an e-based type. This restriction reflects our assumption that intension-insensitive relational nonlogical constants are basically relations between entities, or functions defined in terms of which.

Theorem 5 (Soundness of the intensionalization procedure). *Let G be an extensional grammar in which (i) every lexical item is either nonlogical or logical constant; (ii) every lexical item is of a quasi-relational or e-based type; and (iii) if a nonlogical constant has a quasi-relational type, then all its arguments are of an e-based type. Let Δ_1 and Δ_2 be two derivations of type t over G. Then Δ_1 extensionally entails Δ_2 over G iff Δ_1 intensionally entails Δ_2 over $\mathcal{I}(G)$.*

3.2   Example

To see the benefits of the proposed intensionalization procedure, let us get back to the lexical entries from Tables 1 and 2. The intensionalizations of these entries are shown in Table 3. For a logical constant α in this table, we write its constant interpretation as $L(f)$, where f is its constant extensional interpretation. A routine but somewhat tedious calculation shows that the relevant functions are as follows:

$$L(\text{EVERY}) = \lambda\mathcal{A}\lambda\mathcal{B}\lambda w.\forall x[\mathcal{A}(x)(w) \rightarrow \mathcal{B}(x)(w)]$$

$$L(\text{SOME}) = \lambda\mathcal{A}\lambda\mathcal{B}\lambda w.\exists x[\mathcal{A}(x)(w) \wedge \mathcal{B}(x)(w)]$$

$$L(\mathbf{ONS}) = \lambda R\lambda\mathcal{F}\lambda x\lambda w.\mathcal{F}^w(\lambda y.R(y)(x)(w))$$

$$L(\mathbf{OWS}) = \lambda\mathcal{R}\lambda\mathcal{F}\lambda\mathcal{Q}\lambda w.\mathcal{F}^w(\lambda y.\mathcal{Q}^w(\lambda x.\mathcal{R}^w(\lambda A.A(y))(x)))$$

$$L(\mathbf{LIFT}) = \lambda x\lambda\mathcal{A}.\mathcal{A}(x)$$

$$L(\mathbf{ADJ}) = \lambda\mathcal{A}\lambda\mathcal{B}\lambda x\lambda w.\mathcal{A}(x)(w) \wedge \mathcal{B}(x)(w)$$

In the lambda-terms above, x and y are of type e, w is of type s, A is of type et, $\mathcal{A}$ and $\mathcal{B}$ are of type $e(st)$, $\mathcal{Q}$ and $\mathcal{F}$ are of type $(e(st))(st)$, R is of type $e(e(st))$, and $\mathcal{R}$ is of type $(((e(st))(st))(e(st)))$.

The soundness of the intensionalization process is demonstrated below with two simple examples. The sentences *every king smiled* and *every bald king smiled* have the derivations in (9a) and (10a), respectively. The denotations of these derivations are shown in (9b) and (10b), for the extensional grammar, and in (9c) and (10c), for the intensionalized grammar. These denotations support entailment from derivation (9a) to derivation (10a) both in the extensional grammar and in its intensionalization.

(9)   a. [[*every king*] *smiled*]
        b. $\forall x^e[\mathbf{king}(x) \rightarrow \mathbf{smile}(x)]$

Table 3. Intensionalization of the lexical entries from Tables 1 and 2.

word α	Type	$\{\!\{\alpha\}\!\}^{\mathcal{L}}$
Mary, John,...	e	D_e
red, sick,...	$e(st)$	$D_{e(st)}$
king, queen,...	$e(st)$	$D_{e(st)}$
smile,...	$e(st)$	$D_{e(st)}$
kiss,...	$e(e(st))$	$D_{e(e(st))}$
every	$(e(st))((e(st))(st))$	$\{L(\text{EVERY})\}$
a	$(e(st))((e(st))(st))$	$\{L(\text{SOME})\}$
ϵ_{ONS}	$(e(e(st)))(((e(st))(st))(e(st)))$	$\{L(\mathbf{ONS})\}$
ϵ_{OWS}	$(((e(st))(st))(e(st)))(((e(st))(st))(((e(st))(st))(st)))$	$\{L(\mathbf{OWS})\}$
ϵ_{lift}	$e((e(st))(st))$	$\{L(\mathbf{LIFT})\}$
ϵ_{adj}	$(e(st))((e(st))(e(st)))$	$\{L(\mathbf{ADJ})\}$

      c. $\lambda w^s.\forall x^e[\mathbf{king}(x)(w) \to \mathbf{smile}(x)(w)]$
(10) a. $[[every\ [[\epsilon_{\text{adj}}\ bald]\ king]]\ smiled]$
      b. $\forall x^e[(\mathbf{bald}(x) \wedge \mathbf{king}(x)) \to \mathbf{smile}(x)]$
      c. $\lambda w^s.\forall x^e[(\mathbf{bald}(x)(w) \wedge \mathbf{king}(x)(w)) \to \mathbf{smile}(x)(w)]$

The second example involves derivations (7a) and (8a) of the sentence *a queen kissed every king*, repeated here as (11a) and (12a) respectively. The respective extensional interpretations (7b) and (8b) of these derivations exhibit an extensional entailment from the ONS derivation to the OWS derivation. In the intensionalized grammar, derivation (11a) has the ONS interpretation in (11b), while (12a) has the OWS interpretation in (12b). From the soundness theorem it follows that these intensionalized interpretations preserve the extensional entailment, as can be independently verified.

(11) a. $[[a\ queen]\ [[\epsilon_{\text{ONS}}\ kissed]\ [every\ king]]]$
     b. $\lambda w^s.\exists x^e[\mathbf{queen}(x)(w) \wedge \forall y^e[\mathbf{king}(y)(w) \to \mathbf{kiss}(y)(x)(w)]]$
(12) a. $[[a\ queen]\ [[\epsilon_{\text{OWS}}\ [\epsilon_{\text{ONS}}\ kissed]]\ [every\ king]]]$
     b. $\lambda w^s.\forall y^e[\mathbf{king}(y)(w) \to \exists x^e[\mathbf{queen}(x)(w) \wedge \mathbf{kiss}(y)(x)(w)]]$

4   Extending the intensionalized system

Our main reason to develop a sound intensionalization procedure is to allow a simple introduction of intension-sensitive entries into the lexicon, without any further modification in the intensionalized grammar. Van Benthem's typing strategy in [3] that we have followed enables a simple and natural introduction of intension-sensitive words like *seek* and *need* without modifying the grammar, while allowing these intension-sensitive TVs (ITVs) to be of the same type as (intensionalized) intension-insensitive TVs (ETVs) like *kiss*. In this section we demonstrate this by integrating ITVs into the lexicon of Table 3. As we shall see, *de dicto*/*de re* ambiguities and coordinations of ITVs with ETVs are simply treated in the extended grammar.

One simple way to add ITVs to the lexicon from Table 3 is to let them denote nonlogical constants of type $((e(st))(st))(e(st))$. By this we treat ITVs as in PTQ, where the object of such verbs is assumed to denote an *intensional quantifier*. It should be emphasized, however, that this is not an assumption of our intensionalization procedure but a simple way to accommodate ITVs into the toy lexicon that we are using for exemplification. The treatment of *de dicto/de re* ambiguities under this technique is demonstrated using the two derivations in (13) below of the sentence *Mary sought a king.*

(13) a. [*Mary* [*sought* [*a king*]]]
     b. [[ϵ_{lift} *Mary*] [[ϵ_{OWS} *sought*] [*a king*]]]

(14) a. $\mathbf{seek}(\lambda\mathcal{B}^{e(st)}\lambda w^s.\exists y^e[\mathbf{king}(y)(w) \wedge \mathcal{B}(y)(w)])(\mathbf{mary})$
     b. $\lambda w^s \exists y^e[\mathbf{king}(y)(w) \wedge \exists Q^{(e(st))(st)}[Q^w = (\lambda A^{et}.A(y)) \wedge \mathbf{seek}(Q)(\mathbf{mary})]]$

The denotation of (13a) is the *de dicto* interpretation in (14a) above. The denotation of (13b), on the other hand, is the *de re* interpretation in (14b). Note that the latter is created by the same mechanism that creates object wide scope interpretations in the extensional grammar (cf. (8)). This is similar to PTQ, where the *quantifying in* mechanism is responsible both for the creation of scope ambiguities and for the creation of *de dicto/de re* ambiguities. However, in distinction with the proposals by Montague, [10] and others, intensionalization spares us the need to define an intricate intensional version of the scope shifting mechanism.

The typing strategy that we follow also enables a simple treatment of coordinations between ITVs and ETVs, like in the sentence *Mary sought and kissed a king.* Each of the two derivations in (15) is equivalent to another reading of the intensional conjunct in the (ambiguous) paraphrase *Mary sought a king and kissed a king.* Derivation (15a) represents the reading in which Mary sought a king *de dicto*, while derivation (15b) represents the reading in which Mary sought a king *de re*.

(15) a. [*Mary* [[*sought* and [ϵ_{ONS} *kissed*]] [*a king*]]]
     b. [*Mary* [[[ϵ_{OWS} *sought*] and [ϵ_{OWS} [ϵ_{ONS} *kissed*]]] [*a king*]]]

As mentioned above, our intensionalization process is not restricted to the Montagovian treatment of ITVs. An example of an alternative treatment is that of Zimmermann in [12], where the (in)definite object of an ITV is assumed to denote a property. To support Zimmermann's treatment, we assume that extensional indefinite NPs are of type *et*, and that the determiner *a(n)* is a logical constant of type $(et)(et)$, whose constant denotation is the identity function of this type. Thus, an indefinite like *a king* denotes the set denoted by the noun *king.*

To allow the composition of ETVs with predicative indefinites, it is customary to assume a process of *semantic incorporation* ([13],[14]). In this process, an ETV can compose with predicative indefinites by way of existential quantification. Formally, the extensional incorporation operator on ETVs is defined as follows.

(16) $\mathbf{INC} \stackrel{def}{=} \lambda R^{e(et)}\lambda P^{et}.\lambda y^e.\exists x^e[R(x)(y) \wedge P(x)]$

Thus, we assume that there is some phonologically-silent lexical item ϵ_{INC}, whose constant denotation is the function **INC**. The relevant additions to the extensional lexical entries from Tables 1 and 2 are given in (17) below.

(17)

word α	Type	$\{\!\{\alpha\}\!\}^{\mathcal{F}}$	λ-term
a(n)	$(et)(et)$	$\{\mathrm{A}\}$	$\lambda A^{et}.A$
ϵ_{INC}	$(e(et))((et)(et))$	$\{\mathbf{INC}\}$	$\lambda R^{e(et)}\lambda A^{et}\lambda x^e.\exists y^e[A(y) \wedge R(y)(x)]$

Since both the determiner $a(n)$ and the empty word ϵ_{INC} are logical constants, they are both intensionalized using the operator $L(\cdot)$. The constant interpretations of these two items are as follows:

$$L(\mathrm{A}) = \lambda \mathcal{A}^{e(st)}.\mathcal{A}$$

$$L(\mathbf{INC}) = \lambda \mathcal{R}^{e(e(st))}\lambda \mathcal{A}^{e(st)}\lambda x^e \lambda w^s.\exists y^e[\mathcal{A}(y)(w) \wedge \mathcal{R}(y)(x)(w)]$$

With these intensionalized values, an ITV like *seek* can be added to the lexicon as a nonlogical constant of type $(e(st))(e(st))$, whose object argument is a property. Meaning derivations for coordinations like *sought and kissed a king* are now easily obtained using the incorporation operator. For example, derivation (18a) of the sentence *Mary sought and kissed a king* is interpreted as (18b).

(18)   a. [*Mary* [[*sought* and [ϵ_{INC} *kissed*]] [*a king*]]]
       b. $\lambda w^s.\exists y^e[\mathbf{king}(y)(w) \wedge \mathbf{kiss}(y)(\mathbf{mary})(w)] \wedge \mathbf{seek}(\mathbf{king})(\mathbf{mary})(w)$

Arguably, this account is as natural as the derivation in (15a), based on the Montagovian treatment of ITVs.

Like derivation (15a), derivation (18a) amounts to the reading of the sentence *Mary sought and kissed a king* in which Mary sought a king *de dicto*. A *de dicto* reading is also easily derived for simple transitive sentences without coordinations. For example, derivation (19a) of the sentence *Mary sought a king* is interpreted as (19b).

(19)   a. [*Mary* [*sought* [*a king*]]]
       b. $\mathbf{seek}(\mathbf{king})(\mathbf{mary})$

However, as Zimmermann notes, *de re* readings with ITVs, and more generally object-wide-scope readings, require a scope shifting mechanism for predicative indefinites. One such mechanism is the incorporation operator, which allows a property (e.g., the denotation of *a king* in (18a)) to take a wide scope over a relation between two entities (e.g. the denotation of *kissed* in (18a)). The same strategy can be used to derive a *de re* reading for a sentence like *Mary sought a king*, but for this the denotation of the ITV *seek* (type $(e(st))(e(st))$ must be shifted to a suitable relation between two entities (type $e(e(st))$). This shifting can be achieved by the intensionalization of the operator **AL** in (20) below. In the definition of the extensional **AL**, **indent** is the operator $\lambda x^e \lambda y^e.y = x$ from e.g., [15].

(20) $\mathbf{AL} \stackrel{def}{=} \lambda R^{(et)(et)}\lambda x^e.R(\mathbf{indent}(x))$

Suppose that in the extensional system there is a zero morphology word ϵ_{al} that denotes **AL** in every model. With this operator, derivation (21a) of the sentence *Mary sought a king* is interpreted as (21b).

(21)   a. $[Mary \ [[\epsilon_{\mathrm{INC}} \ [\epsilon_{\mathrm{al}} \ sought]] \ [a \ king]]]$
       b. $(L(\mathbf{INC}))((L(\mathbf{AL}))(\mathbf{seek}))(\mathbf{king})(\mathbf{mary}) =$
          $\lambda w^s.\exists x^e[\mathbf{king}(x)(w) \wedge \exists P^{e(st)}[P^w = (\lambda y^e.y = x) \wedge \mathbf{seek}(P)(\mathbf{mary})]]$

In words, according to this interpretation the sentence is true in w just in case there is a king in w such that Mary sought a property that uniquely defines this king in w. It should be noted that the derivation of **AL** from **indent** is a result of hypothetical reasoning that is embodied in undirected type logical grammars as proposed in [16] and [17].

5   Conclusion

So far, the study of intensionalization has not been a central part of the massive semantic literature on intensionality. In this paper we argued that such a process is necessary if we want to understand better the separation between extensional semantics and intensional semantics. We propose that intensionality phenomena are lexically driven, and that it is mostly this fact that allowed Montague to use essentially extensional mechanisms for treating long-standing puzzles like *de dicto/de re* ambiguities. The study of intensionalization provides a missing link in this story: it explains what is "extensional" in those mechanisms. By doing that, it articulates the lexically-driven nature of intensionality phenomena in natural language.

References

1. Montague, R.: The proper treatment of quantification in ordinary English. In Hintikka, J., Moravcsik, J., Suppes, P., eds.: Approaches to Natural Languages: proceedings of the 1970 Stanford workshop on grammar and semantics. D. Reidel, Dordrecht (1973) Reprinted in R. Thomason, editor (1974), *Formal Philosophy: selected papers of Richard Montague*, Yale, New Haven.
2. Gallin, D.: Intensional and Higher-Order Modal Logic. Volume 19 of North-Holland Mathematics Studies. North-Holland, Amsterdam (1975)
3. Van Benthem, J.: Strategies of intensionalization. In Bodnár, I., Máté, A., Pólos, L., eds.: Intensional Logic, History of Philosophy, and Methodology: To Imre Ruzsa on the occasion of his 65th birthday. Department of Symbolic Logic, Eötvös University, Budapest (1988) 41–59
4. Partee, B., Rooth, M.: Generalized conjunction and type ambiguity. In Bauerle, R., Schwarze, C., von Stechow, A., eds.: Meaning, Use and Interpretation of Language. De Gruyter, Berlin (1983)
5. Keenan, E., Faltz, L.: Boolean Semantics for Natural Language. D. Reidel, Dordrecht (1985)
6. Heim, I., Kratzer, A.: Semantics in Generative Grammar. Blackwell Publishers, Oxford (1998)

7. Shan, C.C.: Monads for natural language semantics. In Striegnitz, K., ed.: Proceedings of the ESSLLI-2001 student session. (2001) 285–298 Downloadable from `http://arxiv.org/pdf/cs.CL/0205026`.
8. Barker, C.: Continuations and the nature of quantification. Natural Language Semantics **10** (2002) 211–242
9. Ben-Avi, G.: Types and Meanings in Quantifier Scope, Selection and Intensionality. PhD Thesis, Technion. In preparation. Draft forthcoming in `http://www.cs.technion.ac.il/~bagilad` (2007)
10. Hendriks, H.: Studied Flexibility: categories and types in syntax and semantics. PhD thesis, University of Amsterdam (1993)
11. Muskens, R.: A Relational Formulation of the Theory of Types. Linguistics and Philosophy **12** (1989) 325–346
12. Zimmermann, T.E.: On the proper treatment of opacity in certain verbs. Natural Language Semantics **11** (1993) 149–179
13. Van Geenhoven, V.: Semantic Incorporation and Indefinite Descriptions: semantic and syntactic aspects of noun incorporation in West Greenlandic. CSLI Publications, Stanford (1998)
14. Van Geenhoven, V., McNally, L.: On the property analysis of opaque complements. Lingua **115** (2005) 885–914
15. Partee, B.: Noun phrase interpretation and type shifting principles. In Groenendijk, J., de Jong, D., Stokhof, M., eds.: Studies in Discourse Representation Theories and the Theory of Generalized Quantifiers. Foris, Dordrecht (1987)
16. Van Benthem, J.: Essays in Logical Semantics. D. Reidel, Dordrecht (1986)
17. Muskens, R.: Language, Lambdas, and Logic. In Kruijff, G.J., Oehrle, R., eds.: Resource Sensitivity in Binding and Anaphora. Studies in Linguistics and Philosophy. Kluwer (2003) 23–54