The Cooper Storage Idiom[*]

Gregory M. Kobele

October 16, 2016

Abstract

Cooper storage is a widespread technique for associating sentences with their meanings, used (sometimes implicitly) in diverse linguistic and computational linguistic traditions. This paper encodes the data structures and operations of cooper storage in the simply typed linear λ-calculus, revealing the rich categorical structure of a parameterized applicative functor. In the case of finite cooper storage, which corresponds to ideas in current transformational approaches to syntax, the semantic interpretation function is a homomorphism, and thus, by results of Kanazawa (2007), generation can be done in polynomial time.

1 Introduction

Since Montague (1973), a guiding principle in the semantics of natural language has been to map sentences to meanings *homomorphically* based on their syntactic structure. The proper treatment of quantification has been challenging from the outset, as quantifier denoting expressions seem in general to be structurally embedded inside of the expressions whose meanings they should take as their arguments. The strategy of Montague (1973), adopted by much of the subsequent linguistic literature, is to change the structures of syntax so as to have the quantificational elements in as transparent a position as possible for semantic interpretation.

Cooper storage (Cooper, 1983) is a technique for interpreting sentences with quantificational elements based on structures where these elements are not in the positions which straightforwardly correspond to their semantic scopes. It involves assigning to each node of a syntax tree, in a non-deterministic recursive bottom-up manner, a pair consisting of an expression in some logical language with variables which will here be called the *main expression*, and a data structure, called the *store*, containing pairs of a free variable and a logical expression. The main expression associated with a node indicates the intended meaning of

[*]My thanks to Chris Barker, Dylan Bumford, Simon Charlow, Itamar Francez, Philippe de Groote, and Carl Pollard for their feedback.

the syntax tree rooted at that node, whereas the store contains expressions representing the meanings of parts of the syntax tree rooted at that node whose relative scope with respect to the entire syntax tree have yet to be determined.

The main formal problem surrounding cooper storage is that it requires some mechanism for avoiding accidental variable capture (§1.3), and thus, among other things, this means that the map from parse tree to meaning cannot be represented as a λ-homomorphism (de Groote, 2001a). This makes difficult an understanding of the complexity of the form-meaning relation expressed by grammars making use of cooper storage.

This paper

- provides a general characterization of cooper storage in terms of parameterized applicative functors. This characterization has as special cases the variations on the cooper storage theme present in the literature.
- provides a sequent notation for cooper storage. As this notation is very close to that of Cooper (1983), it can be viewed as putting this latter on solid logical ground.
- interprets cooper storage in the linear lambda calculus. This makes available access to general complexity theoretic results in particular on parsing and generation (Kanazawa, 2016).

From a type-logical perspective, cooper storage seems like the mirror image of hypothetical reasoning; instead of using hypotheses to saturate predicates, and only introducing quantified expressions in their scope positions, predicates are directly saturated with quantified expressions. This situation, while logically somewhat backwards, allows the otherwise higher order proof structures to be simulated with simpler second order terms (i.e. trees).

From a transformational, LF-interpretation based, perspective, it is intuitive to think of the main expression as the meaning of the LF-tree rooted in that node, with variables for traces, and of the store as containing the meanings of the quantifiers which have not yet reached their final landing site (Larson, 1985). Indeed, recent proposals about compositional semantics in minimalist grammars (Kobele, 2006; Hunter, 2010; Kobele, 2012) implement quantifier-raising using cooper storage. These approaches exploit the 'logical backwardness' (as described above) of cooper storage to account for empirical constraints on scope possibilities in natural language.

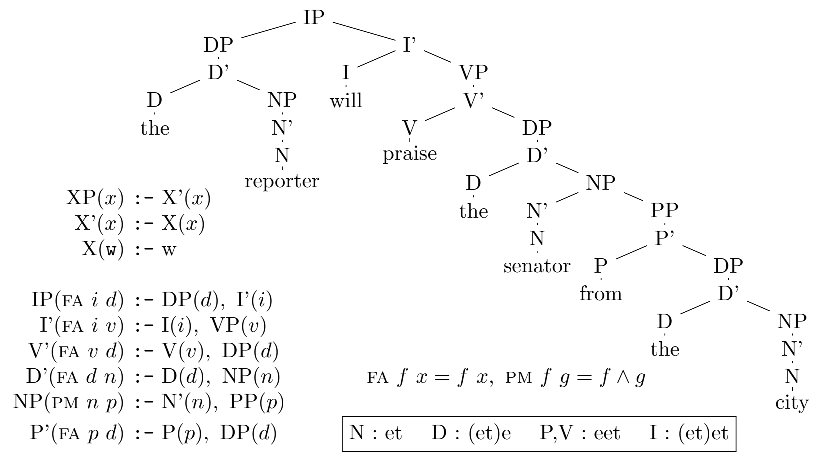
1.1 An example

Consider a linguist analysing a language (English), who for various reasons has decided to analyze the syntactic structure of a sentence like 1 as in figure 1.

1. The reporter will praise the senator from the city.

The figure uses a Horn-clause like notation for context-free grammars, where a production rule of the form $X \to Y_1 \dots Y_n$ is written instead as $X \;\text{:-}\; Y_1, \dots, Y_n$.

The linguist has also come up with a compositional semantic interpretation for this analysis. Clauses are semantically annotated; a clause is of the form $X(M)$ `:-` $Y_1(y_1), \ldots, Y_n(y_n)$, where $y_1, \ldots, y_n$ are variables, and M is a term whose free variables are among $y_1, \ldots, y_n$. Such a clause is to be understood as having as its semantic contribution a function $\lambda y_1, \ldots, y_n.M$. Perhaps more intuitively, when it is used to construct an expression of category X from expressions of categories Y_1 through Y_n with meanings y_1 through y_n, the meaning of the resulting expression is M.



$$\begin{aligned}
\text{XP}(x) &:\!- \text{X'}(x)\\
\text{X'}(x) &:\!- \text{X}(x)\\
\text{X}(\texttt{w}) &:\!- \text{w}\\[1ex]
\text{IP}(\textsc{fa}\ i\ d) &:\!- \text{DP}(d),\ \text{I'}(i)\\
\text{I'}(\textsc{fa}\ i\ v) &:\!- \text{I}(i),\ \text{VP}(v)\\
\text{V'}(\textsc{fa}\ v\ d) &:\!- \text{V}(v),\ \text{DP}(d)\\
\text{D'}(\textsc{fa}\ d\ n) &:\!- \text{D}(d),\ \text{NP}(n)\\
\text{NP}(\textsc{pm}\ n\ p) &:\!- \text{N'}(n),\ \text{PP}(p)\\
\text{P'}(\textsc{fa}\ p\ d) &:\!- \text{P}(p),\ \text{DP}(d)
\end{aligned}$$

$$\textsc{fa}\ f\ x = f\ x, \quad \textsc{pm}\ f\ g = f \wedge g$$

$$\boxed{\text{N : et} \quad \text{D : (et)e} \quad \text{P,V : eet} \quad \text{I : (et)et}}$$

Figure 1: A grammar and syntactic analysis of sentence 1

In figure 1, the expressions of category DP have been analysed as being of type e, that is, as denoting individuals.[1] While this is not an unreasonable analytical decision in this case (where *the* can be taken as denoting a choice function), it is untenable in the general case. Consider the linguist's reaction upon discovering sentence 2, and concluding that determiners are in fact of type $(et)(et)t$ (i.e. that they denote relations between sets).

2. No reporter will praise a senator from every city.

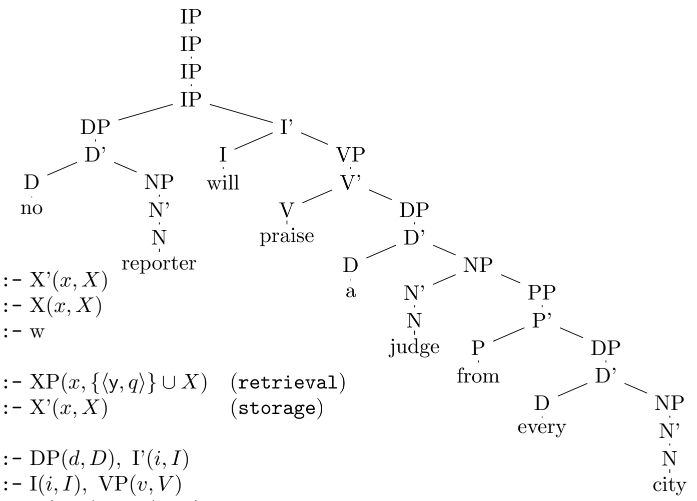
The immediate problem is that the clauses with heads P'($\textsc{fa}\ p\ d$), V'($\textsc{fa}\ v\ d$), and IP($\textsc{fa}\ i\ d$) are no longer well-typed; the variable d now has type $(et)t$ and not e. While in the clause with head IP($\textsc{fa}\ i\ d$) this could be remedied simply by switching the order of arguments to FA, there is no such simple solution for the others, where the coargument is of type eet.[2]

The linguist might first investigate solutions to this problem that preserve the syntactic analysis. One diagnosis of the problem is that whereas the syntax

[1]Our linguist is using the atomic types e and t (which correspond to the ι and o of Church (1940)). The complex type $\alpha\beta$ is elsewhere written as $\alpha \to \beta$, and juxtaposition is right associative; $\alpha\beta\gamma$ is $\alpha(\beta\gamma)$.

[2]The obvious minimal solution, namely, allowing a operation which combines a term of type $(et)t$ with one of type eet (for example $\lambda m, f, k.m(\lambda x.fxk)$), will not extend to an account of the ambiguity of sentences with quantifiers.

was set up to deal with DPs of type e, they are now of type $(et)t$. A solution is to make them behave locally as though they were of type e by adding an operation (`storage`) which converts an expression of type $(et)t$ into one which behaves like something of type e. This is shown in figure 2. Note that, while the



$$\begin{array}{rl}
\text{XP}(x,X) & \text{:- X'}(x,X)\\
\text{X'}(x,X) & \text{:- X}(x,X)\\
\text{X}(\mathtt{w},\emptyset) & \text{:- w}
\end{array}$$

$$\begin{array}{rll}
\text{XP}(q(\lambda \mathsf{y}.x),X) & \text{:- XP}(x,\{\langle \mathsf{y},q\rangle\}\cup X) & (\mathtt{retrieval})\\
\text{XP}(\mathsf{y},X\cup\{\langle \mathsf{y},x\rangle\}) & \text{:- X'}(x,X) & (\mathtt{storage})
\end{array}$$

$$\begin{array}{rl}
\text{IP}(\textsc{fa}\ i\ d, I\cup D) & \text{:- DP}(d,D),\ \text{I'}(i,I)\\
\text{I'}(\textsc{fa}\ i\ v, I\cup V) & \text{:- I}(i,I),\ \text{VP}(v,V)\\
\text{V'}(\textsc{fa}\ v\ d, V\cup D) & \text{:- V}(v,V),\ \text{DP}(d,D)\\
\text{D'}(\textsc{fa}\ d\ n, D\cup N) & \text{:- D}(d,D),\ \text{NP}(n,N)\\
\text{NP}(\textsc{pm}\ n\ p, N\cup P) & \text{:- N'}(n,N),\ \text{PP}(p,P)\\
\text{P'}(\textsc{fa}\ p\ d, P\cup D) & \text{:- P}(p,P),\ \text{DP}(d,D)
\end{array}$$

$\textsc{fa}\ f\ x = f\ x,\ \ \textsc{pm}\ f\ g = f \wedge g$

$\boxed{\text{N : et} \quad \text{D : (et)(et)t} \quad \text{P,V : eet} \quad \text{I : (et)et}}$

Figure 2: Preserving syntactic structure via cooper storage

syntactic structure is now different, as there are multiple unary branches at the root (one for each quantifier retrieved from storage), this difference is not of the kind that syntactic argumentation is usually sensitive to. Thus, this plausibly preserves the syntactic insights of our linguist.

The linguist's strategy was to globally enrich meanings to include a set (a quantifier store) containing some number of variables paired with quantificational expressions. An expression with meaning $\langle x, X\rangle$ can, via the `storage` rule, become something with the meaning $\langle \mathsf{y}, \{\langle \mathsf{y}, x\rangle\} \cup X\rangle$; here its original meaning, $x : (et)t$, has been replaced with a variable $\mathsf{y} : e$, and has been pushed into the second component of the expression's meaning. The `retrieval` rule allows for a pair $\langle \mathsf{y}, x\rangle$ in the second meaning component to be taken out, resulting in the generalized quantifier representation x to take as argument a function created by abstracting over the variable y.

1.2 Another example

The previous section presented cooper storage in its traditional guise; quantificational expressions can be interpreted higher, but not lower, than the position they are pronounced in. More importantly, in the traditional presentation of cooper storage, the quantifiers in the store are completely dissociated from the syntax. Much work in linguistic semantics (particularly in the tradition of transformational generative grammar) attempts to identify constraints on the scope-taking possibilities of quantificational expressions in terms of their syntactic properties (see, e.g. Johnson (2000)). In this tradition, nominal phrases (among others) are typically syntactically dependent on multiple positions (their *deep*, *surface*, and *logical* positions).

A linguist might, in order to have a simple syntactic characterization of selectional restrictions across sentences like 3 and 4, analyze there as being a single structural configuration in which the selectional restrictions between subject and verb obtain, which is present in both sentences.

3. A dog must bark.

4. A dog must seem to bark.

Figure 3 again uses a Horn-clause like notation, and a production has the form $X(\vec{x})$:- $Y_1(\vec{y_1}), \ldots, Y_n(\vec{y_n})$. The $\vec{y_i}$ on the right hand side of such a rule are finite sequences of pairwise distinct variables, and the $\vec{x}$ on the left is a finite sequence consisting of exactly the variables used on the right. Instead of deriving a set of strings, a non-terminal derives a set of finite sequences of strings. (This is a bottom-up presentation due to Kanazawa (2009) of multiple context-free grammars (Seki et al, 1991).) Categories will in this example consist of either pairs or singletons of what are called in the transformational syntax literature feature bundles,[3] and heavy use of polymorphism will be made (the polymorphic category $[f\alpha, \beta]$ is unifiable with any instantiated category of the form $[fg, h]$, for any g and h). The basic intuition behind the analysis in figure 3 is that a noun phrase (*a dog*) is first combined syntactically with its predicate (*bark*), and is then put into its pronounced position when this becomes available.

A straightforward semantic interpretation scheme simply maps the derivation tree homomorphically to a meaning representation, with binary branching rules corresponding to (either forward or backward) function application, and unary branching rules to the identity function, as shown in figure 4. Here the literals are of the form $X(\vec{x})(x')$, where X and $\vec{x}$ are as before, and x' is a meaning representation (on the left x' is a term, and on the right a variable).This allows the linguist to assign the meaning in 5 to sentence 3.

5. It must be the case that a dog barks. $\Box(\exists(\mathsf{dog})(\mathsf{bark}))$

However, sentence 3 is ambiguous; another reading of this sentence is as in 6.

6. There is a dog which must bark. $\exists(\mathsf{dog})(\lambda x.\Box(\mathsf{bark}(x)))$

The linguist might be tantalized by the fact that the currently underivable reading 6 fits naturally with the surface word order, and indeed, in the derivation of sentence 3 in figure 3, the string *a dog*, although introduced before *must*,

[3]The present syntax is a variant of the notation used in (Stabler and Keenan, 2003) for minimalist grammars.

$$\begin{array}{rl}
[\alpha;\beta]\,(x^\frown y, z) & \texttt{:-}\ [\texttt{=x}.\alpha]\,(x), [\texttt{x};\beta]\,(y,z).\\
[\alpha;\beta]\,(x, y) & \texttt{:-}\ [\texttt{=x}.\alpha]\,(x), [\texttt{x}.\beta]\,(y).\\
[\alpha]\,(y^\frown x) & \texttt{:-}\ [\texttt{+x}.\alpha;\texttt{-x}]\,(x,y)
\end{array}$$

$$\begin{array}{rr}
[\texttt{=n.d.-k}](\mathrm{a})\ \texttt{:-}\ . & [\texttt{=v.i}](\mathrm{to})\ \texttt{:-}\ .\\
[\texttt{n}](\mathrm{dog})\ \texttt{:-}\ . & [\texttt{=v.+k.s}](\mathrm{must})\ \texttt{:-}\ .\\
[\texttt{=d.v}](\mathrm{bark})\ \texttt{:-}\ . & [\texttt{=i.v}](\mathrm{seem})\ \texttt{:-}\ .
\end{array}$$

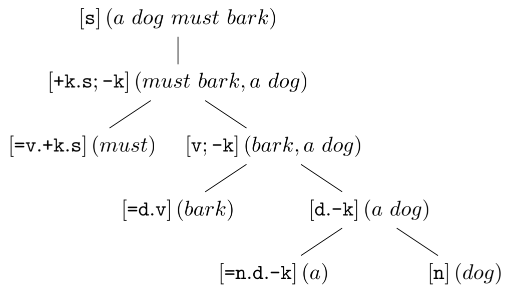


Figure 3: A transformational grammar and analysis of sentences 3 and 4

$$\begin{array}{rl}
[\alpha;\beta]\,(x^\frown y, z)(\textsc{fa}\ x'\ y') & \texttt{:-}\ [\texttt{=x}.\alpha]\,(x)(x'), [\texttt{x};\beta]\,(y,z)(y').\\
[\alpha;\beta]\,(x, y)(\textsc{fa}\ y'\ x') & \texttt{:-}\ [\texttt{=x}.\alpha]\,(x)(x'), [\texttt{x}.\beta]\,(y)(y').\\
[\alpha]\,(y^\frown x)(x') & \texttt{:-}\ [\texttt{+x}.\alpha;\texttt{-x}]\,(x,y)(x')
\end{array}$$

$$\begin{array}{rr}
[\texttt{=n.d.-k}](\mathrm{a})(\exists)\ \texttt{:-}\ . & [\texttt{=v.i}](\mathrm{to})(\texttt{id})\ \texttt{:-}\ .\\
[\texttt{n}](\mathrm{dog})(\mathsf{dog})\ \texttt{:-}\ . & [\texttt{=v.+k.s}](\mathrm{must})(\Box)\ \texttt{:-}\ .\\
[\texttt{=d.v}](\mathrm{bark})(\mathsf{bark})\ \texttt{:-}\ . & [\texttt{=i.v}](\mathrm{seem})(\mathsf{seem})\ \texttt{:-}\ .
\end{array}$$

Figure 4: Interpreting deep structures

is prepended to it in the last step. To allow the quantificational aspect of *a dog* to remain active as long as its phonetic aspect is, the linguist extends meanings with a finite store, whose elements are in correspondence with the derived string parts of the expression, as shown in figure 5. Here the literals are of the form $X(\vec{x})(\vec{x}')$, where the meaning component $\vec{x}'$ is a sequence of meaning representations. The two readings (5 and 6) of sentence 3 are shown in figure 6. (Just the meaning components are shown, as the strings are identical.)

1.3 Making it compositional

As presented above, the data structures involved in cooper storage are not semantic; the objects of the theory are syntactic representations of semantic objects: free variables (or indices of variables) are used to maintain a link between the objects in the store and the semantic arguments they should bind in the main expression. It is of course possible to 'semanticize' variables by reifying assignment functions (as is done explicitly in the textbook of Kreisel and Krivine (1967)), and to reconceptualize variables as functions from assignment

$$\begin{array}{rl}
[\alpha;\beta]\,(x^\frown y, z)(\textsc{fa}\ x'\ y', z') & \texttt{:-}\ [\texttt{=x}.\alpha]\,(x)(x'), [\texttt{x};\beta]\,(y,z)(y',z'). \\
[\alpha;\beta]\,(x,y)(\textsc{fa}\ x'\ \texttt{v}, \langle \texttt{v}, y'\rangle) & \texttt{:-}\ [\texttt{=x}.\alpha]\,(x)(x'), [\texttt{x}.\beta]\,(y)(y'). \\
[\alpha]\,(y^\frown x)(y'(\lambda\texttt{v}.x')) & \texttt{:-}\ [\texttt{+x}.\alpha;\texttt{-x}]\,(x,y)(x', \langle \texttt{v}, y'\rangle).
\end{array}$$

$$\begin{array}{rl}
[\alpha;\beta]\,(x^\frown y, z)(\textsc{fa}\ x'\ y') & \texttt{:-}\ [\texttt{=x}.\alpha]\,(x)(x'), [\texttt{x};\beta]\,(y,z)(y'). \\
[\alpha;\beta]\,(x,y)(\textsc{fa}\ y'\ x') & \texttt{:-}\ [\texttt{=x}.\alpha]\,(x)(x'), [\texttt{x}.\beta]\,(y)(y'). \\
[\alpha]\,(y^\frown x)(x') & \texttt{:-}\ [\texttt{+x}.\alpha;\texttt{-x}]\,(x,y)(x').
\end{array}$$

$$\begin{array}{rlrl}
[\texttt{=n.d.-k}](\text{a})(\exists) & \texttt{:-}\ . & [\texttt{=v.i}](\text{to})(\texttt{id}) & \texttt{:-}\ . \\
[\texttt{n}](\text{dog})(\textsf{dog}) & \texttt{:-}\ . & [\texttt{=v.+k.s}](\text{must})(\Box) & \texttt{:-}\ . \\
[\texttt{=d.v}](\text{bark})(\textsf{bark}) & \texttt{:-}\ . & [\texttt{=i.v}](\text{seem})(\textsf{seem}) & \texttt{:-}\ .
\end{array}$$

Figure 5: Additional rules for surface scope interpretation

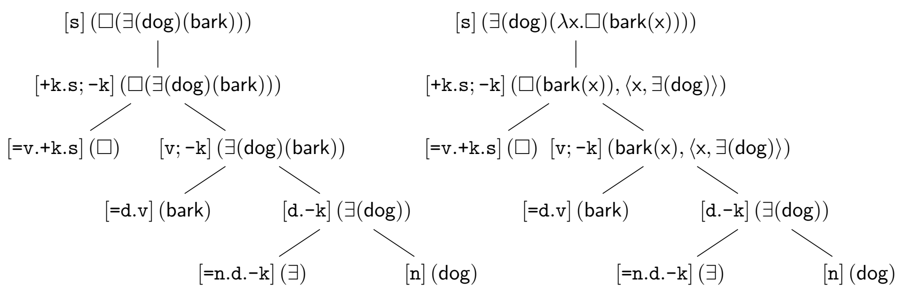


Figure 6: Two readings of 3

functions to individuals. Indeed, both Cooper (1983) and (much later) Kobele (2006) assign the same denotation to objects in the store. Letting g be the type of assignment functions, the representation $\langle x_i, [\![NP]\!]\rangle$ is viewed as a model-theoretic object of type $(gt)gt$ mapping over sentence denotations dependent on a variable assignments in the following way:[4]

$$\langle x_i, [\![NP]\!]\rangle(\phi)(h) := [\![NP]\!]\,(\{a : \phi(h[i := a])\})$$

The deeper problem is that there is no mechanism to ensure freshness of variables; each time the **storage** rule is used a *globally unique* variable name should be produced. Kobele (2006), exploiting the fact that variable names can be uniquely associated with nodes in the derivation tree (the point at which the **storage** rule is used), uses combinators to encode pairs of assignment functions as single assignment functions in a way that allows stored elements to correctly identify the variable they should 'bind'. This convoluted move requires variable binding operators to be simulated via model theoretic objects (of type e.g. $(gt)egt$. When it seems one is reinventing well-understood machinery, it is reasonable to try to recast the problem being addressed so as to take advantage of

[4]The `retrieval` operation is redefined so: XP($\textsc{fa}\ q\ x, X$) :- X'($x, \{q\} \cup X$)

what already exists.

The problem is that free variables are being used (either syntactically or semantically), and these necessitate a complicated sort of bookkeeping. In particular, 1. free variables appear in the main expression, and 2. stored items are paired with free variables. Given the intended use of these free variables, which is that the variable paired with a stored item be abstracted over in the main expression when this stored item is retrieved, the resolution to both of these problems is simple and in retrospect obvious: this lambda abstraction takes place immediately, and is not deferred to some later point in time. Eliminating free variables then obviates the need for fresh variables. The basic idea of this paper is:

> An expression of the form $M, \{\langle x_1, N_1\rangle, \ldots, \langle x_k, N_k\rangle\}$ should instead be replaced by one of the form $\langle N_1, \ldots, N_k, \lambda x_1, \ldots, x_k.M\rangle$.

Crucially, the M and N_is in the first expression are syntactic objects (formulae in some logical language), while in the second expression they are semantic objects (whatever those may be). The intent of Cooper's store is to have all and only those variables free in M which are paired with some expression in the store; pairing a variable with an expression in the store is simply a means to keep track of which argument position this latter expression should be binding. Thus there is a systematic relation between the type of the expressions in the original cooper store and the type of their reformulation here; roughly, if $M, \langle x_1, N_1\rangle, \ldots, \langle x_k, N_k\rangle$ is such that $M : \alpha$, and for all $1 \leq i \leq k$ $x_i : \beta_i$ and $N_i : \gamma_i$, then $\lambda x_1, \ldots, x_k.M$ has type $\beta_1 \to \cdots \to \beta_k \to \alpha$, and the objects N_i' in the store have type γ_i. Generally, there will be some systematic relation between β_i and γ_i; typically γ_i is the type of a function taking a continuation of something of type β_i; $\gamma_i = (\beta_i \to \eta) \to \eta$. I will call β the *trace type* of γ, and write $\mathtt{t}_\gamma := \beta$. The intent behind the introduction of the terminology of trace types is to abstract away from the precise relation between the type of a stored expression and the type of the variable associated with it.

All relevant information about the type of an expression *cum* store is therefore given by the list of types $p := \gamma_1, \ldots, \gamma_k$ of expressions in the store, together with the result type α of the main expression. The type $\Diamond_p \alpha := \mathtt{t}_{\gamma_1} \to \cdots \to \mathtt{t}_{\gamma_n} \to \alpha$ is the type of the main expression, and $\Box_p \alpha$ is the type associated with expressions *cum* stores with list of stored expression types p and main expression type α (this will be revised in a later section).[5]

$$\bigcirc_p \alpha := \Box_p(\Diamond_p \alpha) = \gamma_1 \times \cdots \times \gamma_n \times (\mathtt{t}_{\gamma_1} \to \cdots \to \mathtt{t}_{\gamma_k} \to \alpha)$$

While I will show that the cooper store data structure *can* be encoded in the lambda calculus in the above way, the crucial contribution of this paper is to observe that this type theoretic encoding reveals a non-trivial structure, that of a parameterized applicative functor. Thus all of the operations usually performed on cooper storage expressions are also definable in the simply typed lambda

[5]These types can be viewed as *right folds* over the list p. In particular, $\Diamond_p = \mathbf{foldr}\ (\to \circ\, \mathtt{t})\ p$, and $\Box_p = \mathbf{foldr}\ (\otimes)\ p$, where $(\to \circ\, \mathtt{t})\ x\ y = \mathtt{t}_x \to y$.

calculus, and moreover the fact that expressions with attached stores behave for most intents and purposes as though they had the type α (as opposed to $\bigcirc_p \alpha$), is a consequence of this structure.

1.4 Related work

De Groote (2001b) presents linguistic applications of the $\lambda\mu$-calculus of Parigot (1992).[6] In particular, the $\lambda\mu$-term $\lambda P.\mu\alpha.\mathtt{every}(P)(\lambda x.\alpha x)$, proposed as the meaning representation of the word *every*, has type $(et)e$. He notes that cooper storage can be thought of in these terms; here `storage` is built in to lexical meaning representations using μ-abstraction, and the reductions for μ behave like `retrieval`. In order to account for scope ambiguities, de Groote proposes to use a non-confluent reduction strategy. Crucially, μ-reduction is completely divorced from syntactic structure (just as is retrieval in Cooper (1983)). This means that alternative versions of cooper storage which enforce a tighter connection between syntactic operations and stored elements, as §1.2, are not straightforwardly implementable using the $\lambda\mu$-calculus.

A recent type-logical presentation of cooper storage is given in Pollard (2011). There a sequent is of the form $\Gamma \vdash M : \alpha \dashv \Delta$, where Γ is a variable context, Δ is a set of pairs of the form $\langle x, N \rangle$, where x is a variable, and M and N are what Pollard calls RC-terms with free variables among those in Γ and Δ. (RC-terms are not quite λ-terms, but are straightforwardly interpretable as such.) Here, Δ is simply a quantifier store, exactly as in Cooper (1983); indeed Pollard (2011) is explicitly trying to give a direct type-logical implementation of cooper storage. There are two substantive differences between Pollard's proposal and the one in this paper. First, in Pollard (2011), stored elements may contain free variables. From a more categorical perspective, an RC-sequent of the form $\Gamma \vdash M : \alpha \dashv \Delta$ can be thought of as a term of type $\Gamma \to \mathtt{t}_\Delta \to (\alpha \times \Delta)$. Thus a rule of hypothetical reasoning would be invalid (as variables in either of Γ or $\mathtt{t}_\Delta$ may occur in Δ). Indeed, no rule for implication elimination from Γ is proposed in Pollard (2011), and the corresponding rule for Δ is restricted so as to be applicable only in case the variable does not occur in any terms in Δ. The lack of such a rule is noted in De Groote et al (2011). The presentation here simply rebrackets so as to obtain $\Gamma \to ((\mathtt{t}_\Delta \to \alpha) \times \Delta)$. Second, Pollard uses variables to coordinate the stored expressions with the positions they should ultimately bind into. The proposal here takes advantage of the additional structure in this problem made explicit in the above categorical presentation of types to eschew variables. Namely, the expressions in the store are in a bijective correspondance with the positions they are intended to bind into, which allows this coordination to be achieved by introducing and enforcing an order invariant between the abstractions $\mathtt{t}_\Delta$ and the store Δ.

[6] De Groote (1994) presents a translation of the $\lambda\mu$-calculus into the λ-calculus, using a continuation passing style transform. From this perspective, continuation-based proposals (developed extensively in Barker and Shan (2014), although there the focus is on *delimited* continuations) can be viewed as related to the $\lambda\mu$-calculus, and thus to cooper-storage.

These differences notwithstanding, the present paper (especially given the sequent presentation of cooper storage in §4.3) can be thought of as a continuation of the logical approach to cooper storage initiated in (Pollard, 2011), offering in particular an embedding of the cooper-storage proof system into the linear lambda calculus, as well as a formal specification of the underlying algebraic structures involved.

1.5 Structure of the paper

The remainder of the paper is structured as follows. In the next section are formal preliminaries. The following section introduces the relevant category theoretic notion of applicative functors (but without category theory), defines their parameterized variants, and proves that they enjoy familiar formal properties. Then it is shown that the data structure underlying cooper storage is in fact a parameterized applicative functor, which supports the operations particular to the use of cooper storage in semantics (in particular, retrieval and storage). Various instantiations of the general cooper-storage scheme are presented, which allow for the recasting of the examples in §1.1 and §1.2 in these terms.

2 Formal preliminaries

2.1 Partial functions and algebras

Given a set A, let $\perp$ be a symbol not in A and define $A_\perp = A \cup \{\perp\}$. A partial function with domain A and codomain B is here identified with a total function $f : A \to B_\perp$ where f is said to be *undefined* on $a \in A$ if $f(a) = \perp$. I write $\mathsf{def}(f) = f^{-1}(B)$ for the subset of elements in its domain on which f is defined, and $[A \hookrightarrow B]$ for the set of all partial functions from A to B. Given $A \subseteq B$, a partial function $f : A \hookrightarrow C$ can be coerced to one with domain B by setting $f(b) = \perp$ for all $b \in B - A$. Note that this preserves $\mathsf{def}(f)$. The *empty* function is undefined on all objects in its domain. (Equivalently, it is (a coercion of) the unique map with domain $\emptyset$.) Two partial functions $f, g : A \hookrightarrow B$ are *compatible* iff $\mathsf{def}(f) \cap \mathsf{def}(g) = \emptyset$. Given compatible f and g, their *superposition* is defined to be $f \oplus g : A \hookrightarrow B$ where $(f \oplus g)(a) = \textbf{if } a \in \mathsf{def}(f) \textbf{ then } f(a) \textbf{ else } g(a)$. Note that $f \oplus g$ is just the set theoretic union of f and g viewed as sets of pairs.

I assume familiarity with basic notions of algebra. A *monoid* $M = \langle M, \mathsf{+}, \mathsf{0}\rangle$ consists of a set M together with a designated element $\mathsf{0}$ and an associative operation $\mathsf{+}$ for which $\mathsf{0}$ is an identity. A monoid homomorphism between M and M' is a function $h : M \to M'$ such that $h(\mathsf{0}) = \mathsf{0}'$ and $h(a \mathsf{+} b) = h(a) \mathsf{+}' h(b)$. A *partial monoid* is a monoid which contains an absorbing element $\perp$ such that $a \mathsf{+} \perp = \perp = \perp \mathsf{+} a$. Often the absorbing element will be left implicit (i.e. the carrier set of the partial monoid will be given as M instead of as $M_\perp$). Homomorphisms between partial monoids are required to in addition map absorbing elements to absorbing elements. A monoid is *abelian* iff $\mathsf{+}$ is

$$\frac{}{x : \alpha \vdash x : \alpha}\text{Ax}$$

$$\frac{\Gamma, x : \alpha \vdash M : \beta}{\Gamma \vdash \lambda x.M : \alpha \to \beta}\to\text{I} \qquad \frac{\Gamma \vdash M : \alpha \to \beta \qquad \Delta \vdash N : \alpha}{\Gamma, \Delta \vdash (MN) : \beta}\to\text{E}$$

Figure 7: Linear typing rules

commutative.

Note that for any sets A, B, the set $[A \hookrightarrow B]$ together with $\oplus$ and the empty function forms a partial abelian monoid.

2.2 λ-calculus and types

Here I briefly recall the simply typed λ-calculus (Barendregt et al, 2013). I will write typed terms in the Curry style, but will, if convenient, indicate the type of a variable in a binding construct with a superscript (e.g. $\lambda x^{\alpha}.M$).

Given a finite set A of *atomic types*, the set $\mathcal{T}(A)$ of *(simple) types* over A is the smallest superset of A such that $\alpha, \beta \in \mathcal{T}(A)$ implies that $(\alpha \to \beta) \in \mathcal{T}(A)$.

$$\mathcal{T}_A := A \mid \mathcal{T}_A \to \mathcal{T}_A$$

As is common, parentheses are omitted whenever possible, writing $\alpha \to \beta$ for $(\alpha \to \beta)$. Implication associates to the right; thus $\alpha \to \beta \to \gamma$ stands for $\alpha \to (\beta \to \gamma)$.

Given a countably infinite set X of *variables*, the set Λ of λ-*terms* is the smallest set containing X which is closed under application and abstraction

$$\Lambda := X \mid (\lambda X.\Lambda) \mid (\Lambda\Lambda)$$

Parentheses are omitted under the convention that application associates to the left, i.e. MNO is $((MN)O)$, and multiple abstractions are written as one, i.e. $\lambda x, y.M$ is $\lambda x.(\lambda y.M)$. The simultaneous capture avoiding substitution of $N_1, \ldots, N_k$ for $x_1, \ldots, x_k$ in M is written $M[x_1 := N_1, \ldots, x_k := N_k]$. Terms are identified up to renaming of bound variables. The standard notions of β and η conversion are as follows: $(\lambda x.M)N \Rightarrow_{\beta} M[x := N]$ and, provided x is not free in M, $M \Rightarrow_{\eta} \lambda x.Mx$. A term M is equivalent to N, written $M \equiv_{\beta\eta} N$ just in case M and N can be reduced to the same term O in some finite number of β or η conversion steps.

A term is *linear* just in case every λ abstraction binds exactly one variable (i.e. in every subterm of the form $\lambda x.M$, x occurs free exactly once in M). An important property of linear terms (up to equivalence) is that they are uniquely determined by their principal (most general) type (Babaev and Soloviev, 1982). A linear λ-term M has a type α (when its free variables are assigned types as per a variable context Γ) just in case the sequent $\Gamma \vdash M : \alpha$ can be derived using the inference rules in figure 7. A *(variable) context* $\Gamma : X \hookrightarrow \mathcal{T}_A$ is a partial function such that $|\mathsf{def}(\Gamma)| \in \mathbb{N}$; it is defined only on a finite subset of X. A context Γ will be sometimes represented as a list $x_1 : \alpha_1, \ldots, x_n : \alpha_n$, which is to be understood as indicating that Γ is defined only on $x_1, \ldots, x_n$ and maps each x_i to α_i. If contexts Γ, Δ are compatible, I write Γ, Δ instead of $\Gamma \oplus \Delta$.

3 (Parameterized) Applicative Functors

A fundamental intuition behind cooper storage is that, while a more intuitive meaning is not in fact the meaning of a parse tree node, it *is* somehow *contained* in the meaning of that node. For example, whereas a predicate might intuitively denote a function of type $e \rightarrow t$, this is only the denotation of the main expression, which comes together with a store. This notion of containment is, in the functional programming literature, familiar in the context of *functors*, *applicative functors* and *monads*, where it is fleshed out in an intuitive, and useful, way.

Our setting can be recast in the following way. We see an object of some type α (the main expression), which is somehow embedded in an object of some richer type $\bigcirc\alpha$, for some function $\bigcirc : \mathcal{T}_A \rightarrow \mathcal{T}_A$ over types. Part of our intuitions about this embedding come from the fact that (some of) our semantic operations are stated over these simpler types, yet are given as input more complicated objects – we would like our grammatical operations to be (by and large) insensitive to the contents of the stores; they should be systematically derived from simpler operations acting on the main expressions. The notion of an applicative functor will allow us to do exactly this.

$\bigcirc : \mathcal{T}_A \rightarrow \mathcal{T}_A$ is an *applicative functor* (McBride and Paterson, 2008) if there are operations $\cdot^{\overset{\uparrow}{\circ}}$ and $\cdot^{\underset{\downarrow}{\circ}}$ such that $\cdot^{\overset{\uparrow}{\circ}}$ turns objects of type α into ones of type $\bigcirc\alpha$, for every type α, and $\cdot^{\underset{\downarrow}{\circ}}$ allows expressions of type $\bigcirc(\alpha \rightarrow \beta)$ to be treated as functions from $\bigcirc\alpha$ to $\bigcirc\beta$, for every pair of types α, β, subject to the conditions in figure 8.[7] While an applicative functor does not permit a function $f : \alpha \rightarrow \beta$ to be applied directly to an α-container $a : \bigcirc\alpha$ to yield a β-container $b : \bigcirc\beta$, it *does* allow f to be turned into an $(\alpha \rightarrow \beta)$-container $f^{\overset{\uparrow}{\circ}}$, which can be combined with a via $(\cdot)^{\underset{\downarrow}{\circ}}$:

$$f^{\overset{\uparrow}{\circ}\underset{\downarrow}{\circ}}\ a : \bigcirc\beta$$

This basic structure, where a simple function is lifted into a container type, and then combined with containers of its arguments one by one, is described by McBride and Paterson (2008) as the 'essence of applicative programming,'

[7]Notation has been changed from McBride and Paterson (2008). The operator $(\cdot)^{\overset{\uparrow}{\circ}}$ (there called pure) *lifts* a value into a functor type. This is reflected notationally by having the arrow point up. The operator $(\cdot)^{\underset{\downarrow}{\circ}}$ (there written as a binary infix operator <*> and known as apply) *lowers* its argument from a function-container to a function over containers, and so the arrow points down. Viewing $\bigcirc$ as a necessity operator, the type of $(\cdot)^{\underset{\downarrow}{\circ}}$ is familiar as the **K** axiom, and viewing it as a possibility operator, the type of $(\cdot)^{\overset{\uparrow}{\circ}}$ is the axiom **T**. *Lax logic* (Fairtlough and Mendler, 1997) is the (intuitionistic) modal logic which to the axioms above adds $\bigcirc\bigcirc\alpha \rightarrow \bigcirc\alpha$ and corresponds via the Curry-Howard correspondance to monads (Moggi, 1991; Benton et al, 1998), which are applicative functors enriched with an operation *join* : $\bigcirc\bigcirc\alpha \rightarrow \bigcirc\alpha$ satisfying certain conditions.

$$\cdot^{\overset{\uparrow}{\circ}} : \alpha \to \bigcirc\alpha \qquad (\mathbf{T})$$
$$\cdot^{\underset{\downarrow}{\circ}} : \bigcirc(\alpha \to \beta) \to \bigcirc\alpha \to \bigcirc\beta \qquad (\mathbf{K})$$

$$\mathtt{id}^{\overset{\uparrow}{\circ}\underset{\downarrow}{\circ}} = \mathtt{id} \qquad \text{(identity)}$$
$$((\circ^{\overset{\uparrow}{\circ}\underset{\downarrow}{\circ}}\ u)^{\underset{\downarrow}{\circ}}\ v)^{\underset{\downarrow}{\circ}} = u^{\underset{\downarrow}{\circ}} \circ v^{\underset{\downarrow}{\circ}} \qquad \text{(composition)}$$
$$f^{\overset{\uparrow}{\circ}\underset{\downarrow}{\circ}}\ x^{\overset{\uparrow}{\circ}} = (f\ x)^{\overset{\uparrow}{\circ}} \qquad \text{(homomorphism)}$$
$$u^{\underset{\downarrow}{\circ}}\ x^{\overset{\uparrow}{\circ}} = (\lambda f.fx)^{\overset{\uparrow}{\circ}\underset{\downarrow}{\circ}}\ u \qquad \text{(interchange)}$$

Figure 8: applicative functors: operations (above) and laws (below)

$$\llparenthesis \mathtt{id} \rrparenthesis^{\underset{\downarrow}{\circ}} = \mathtt{id} \qquad \text{(identity)}$$
$$\llparenthesis u \circ v \rrparenthesis^{\underset{\downarrow}{\circ}} = u^{\underset{\downarrow}{\circ}} \circ v^{\underset{\downarrow}{\circ}} \qquad \text{(composition)}$$
$$\llparenthesis f\ x^{\overset{\uparrow}{\circ}} \rrparenthesis = (f\ x)^{\overset{\uparrow}{\circ}} \qquad \text{(homomorphism)}$$
$$u^{\underset{\downarrow}{\circ}}\ x^{\overset{\uparrow}{\circ}} = \llparenthesis \lambda f.fx\ u \rrparenthesis \qquad \text{(interchange)}$$

Figure 9: applicative functors: abbreviated laws

and is abbreviated as $\llparenthesis f\ a \rrparenthesis$. In general, $((f^{\overset{\uparrow}{\circ}\underset{\downarrow}{\circ}}\ a_1)^{\underset{\downarrow}{\circ}}\ \ldots)^{\underset{\downarrow}{\circ}}\ a_n$ is abbreviated as $\llparenthesis f\ a_1\ \ldots\ a_n \rrparenthesis$; as a special case, $\llparenthesis f \rrparenthesis = f^{\overset{\uparrow}{\circ}}$. Making use of this abbreviation, the applicative functor laws from figure 8 can be succinctly given as in figure 9.

An important property of applicative functors is that they are closed under composition.

Lemma 1 ((McBride and Paterson, 2008))**.** *If* $\Box, \Diamond : \mathcal{T}_A \to \mathcal{T}_A$ *are applicative functors, then so too is* $\Box \circ \Diamond$*.*

Figure 10 provides a list of notation that shall be used in the remainder of this paper.

$\Box$	$(\cdot)^{\overset{\uparrow}{\Box}}$	$(\cdot)^{\underset{\downarrow}{\Box}}$	$\boxdot$	$(\cdot)^{\overset{\uparrow}{\boxdot}}$	$(\cdot)^{\underset{\downarrow}{\boxdot}}$	$\llbracket f\ a_1\ \ldots\ a_n \rrbracket$
$\Diamond$	$(\cdot)^{\overset{\uparrow}{\Diamond}}$	$(\cdot)^{\underset{\downarrow}{\Diamond}}$	$\Diamonddot$	$(\cdot)^{\overset{\uparrow}{\Diamonddot}}$	$(\cdot)^{\underset{\downarrow}{\Diamonddot}}$	$\langle\!\vert f\ a_1\ \ldots\ a_n \vert\!\rangle$
$\bigcirc$	$(\cdot)^{\overset{\uparrow}{\circ}}$	$(\cdot)^{\underset{\downarrow}{\circ}}$	$\odot$	$(\cdot)^{\overset{\uparrow}{\odot}}$	$(\cdot)^{\underset{\downarrow}{\odot}}$	$\llparenthesis f\ a_1\ \ldots\ a_n \rrparenthesis$
			$\#$	$(\cdot)^{\Uparrow}$	$(\cdot)^{\Downarrow}$	$[\![\![f\ a_1\ \ldots\ a_n]\!]\!]$

Figure 10: Notation for applicative functors and associated operators. $\#$ will be used exclusively as a metavariable ranging over applicative functors.

3.1 Parameters

We would like to view cooper storage in terms of applicative functors; so that there should be a type mapping $\bigcirc : \mathcal{T}_A \to \mathcal{T}_A$ such that $\bigcirc\alpha$ is a cooper store with main expression of type α. However, the type of a cooper store must depend not only on the type of the main expression, but also on the types of the stored expressions. Thus for each possible store type p, we need a possibly different type mapping $\bigcirc_p : \mathcal{T}_A \to \mathcal{T}_A$; the type $\bigcirc_p\alpha$ is the type of a cooper storage expression with main expression type α and store type p. With this intended interpretation of the $\bigcirc_p$, we see that none of these are applicative functors on their own; in particular, the only reasonable way to inject an expression of type α into the world of cooper storage is to associate it with an empty store. Thus we would like the operation $\cdot^{\overset{\uparrow}{\circ}}$ to map an expression of type α to one of type $\bigcirc_{\mathtt{0}}\alpha$. Similarly, if two expressions with their own stores are somehow combined, the store of the resulting expression includes the stores of both. Thus the desired operation $\cdot^{\underset{\downarrow}{\circ}}$ must relate the family of type mappings $\bigcirc_p$ to one another in the following way:

$$\cdot^{\underset{\downarrow}{\circ}} : \bigcirc_p(\alpha \to \beta) \to \bigcirc_q\alpha \to \bigcirc_{p\mathtt{+}q}\beta$$

The necessary generalization of applicative functors can be dubbed *parameterized* applicative functors, after the *parameterized monads* of Melliès (2016).[8] Given a monoid (of parameters) $\mathbf{P} := \langle P, \mathtt{0}, \mathtt{+}\rangle$, a *parameterized applicative functor* is a function $\bigcirc : P \to \mathcal{T}_A \to \mathcal{T}_A$ together with maps $\cdot^{\overset{\uparrow}{\circ}} : \alpha \to \bigcirc_{\mathtt{0}}\alpha$ and $(\cdot)^{\underset{\downarrow}{\circ}}_{p,q} : \bigcirc_p(\alpha \to \beta) \to \bigcirc_q\alpha \to \bigcirc_{p\mathtt{+}q}\beta$ for every $\alpha, \beta \in \mathcal{T}_A$ and $p, q \in P$ such that $p \mathtt{+} q$ is defined satisfying the equations in figure 11.[9] These equations are the same as those in figure 8, though their types are different. These equations require that $\mathtt{0}$ is an identity for $\mathtt{+}$, and that $\mathtt{+}$ is associative; in other words, that $\mathbf{P}$ is in fact a monoid.[10] In our present context, the elements of P represent the possible types of stores, with $\mathtt{0}$ the type of the empty store, and $\mathtt{+}$ the function describing the behaviour of the mode of store combination at the type level.

Note that it it not necessary that a parameterized applicative functor $\bigcirc : P \to \mathcal{T}_A \to \mathcal{T}_A$ be such that for some $p \in P$ $\bigcirc_p$ is an applicative functor in its own right (although each $\bigcirc_p$ is a (parameterized) functor). Non-parameterized applicative functors are the special case of parameterized applicative functors where the parameters are ignored (i.e. the applicative functor is a constant mapping from P into $\mathcal{T}_A \to \mathcal{T}_A$).

New parameterized applicative functors can be constructed out of old ones in various regular ways. In particular, parameters may be pulled back along a homomorphism, and functors may be composed.

[8]The identically named parameterized monads of Atkey (2009), which are more popular in the functional programming literature, can be seen to be (thanks to their reformulation by McBride (2011) in terms of slice categories) special cases of this one.

[9]The parameter arguments will sometimes be suppressed for readability; it is always possible to reconstruct them from the context.

[10]Rather, the equations require only that $\bigcirc_{\mathtt{0}\mathtt{+}p} = \bigcirc_p = \bigcirc_{p\mathtt{+}\mathtt{0}}$ and that $\bigcirc_{p\mathtt{+}(q\mathtt{+}r)} = \bigcirc_{(p\mathtt{+}q)\mathtt{+}r}$. This is automatic if P is in fact a monoid, but would also be satisfied if, for example, $\bigcirc$ were the constant function from P into $\mathcal{T}_A \to \mathcal{T}_A$.

$$\cdot^{\overset{\uparrow}{\circ}} : \alpha \to \bigcirc_{0}\alpha \qquad (\mathbf{T})$$

$$(\cdot)^{\underset{\downarrow}{\circ}}_{p,q} : \bigcirc_{p}(\alpha \to \beta) \to \bigcirc_{q}\alpha \to \bigcirc_{p+q}\beta \qquad (\mathbf{K})$$

$$\underbrace{(\mathtt{id}^{\overset{\uparrow}{\circ}})^{\underset{\downarrow}{\circ}}_{0,p}}_{\bigcirc_p\alpha\to\bigcirc_{0+p}\alpha} = \underbrace{\mathtt{id}}_{\bigcirc_p\alpha\to\bigcirc_p\alpha} \qquad \text{(identity)}$$

$$\underbrace{(((\circ^{\overset{\uparrow}{\circ}})^{\underset{\downarrow}{\circ}}_{0,p}\ u)^{\underset{\downarrow}{\circ}}_{p,q}\ v)^{\underset{\downarrow}{\circ}}_{p+q,r}}_{\bigcirc_r\alpha\to\bigcirc_{((0+p)+q)+r}\beta} = \underbrace{(u)^{\underset{\downarrow}{\circ}}_{p,q+r} \circ (v)^{\underset{\downarrow}{\circ}}_{q,r}}_{\bigcirc_r\alpha\to\bigcirc_{p+(q+r)}\beta} \qquad \text{(composition)}$$

$$\underbrace{(f^{\overset{\uparrow}{\circ}})^{\underset{\downarrow}{\circ}}_{0,0}\ x^{\overset{\uparrow}{\circ}}}_{\bigcirc_{0+0}\alpha} = \underbrace{(f\ x)^{\overset{\uparrow}{\circ}}}_{\bigcirc_0\alpha} \qquad \text{(homomorphism)}$$

$$\underbrace{(u)^{\underset{\downarrow}{\circ}}_{p,0}\ x^{\overset{\uparrow}{\circ}}}_{\bigcirc_{p+0}\alpha} = \underbrace{((\lambda f.fx)^{\overset{\uparrow}{\circ}})^{\underset{\downarrow}{\circ}}_{0,p}\ u}_{\bigcirc_{0+p}\alpha} \qquad \text{(interchange)}$$

Figure 11: parameterized applicative functors: operations (above) and laws (below)

Theorem 2. *Let* $\mathbf{P}, \mathbf{Q}$ *be monoids, and let* $h : Q \to P$ *be a monoid homomorphism. Then for any parameterized applicative functor* $\bigcirc : P \to \mathcal{T}_A \to \mathcal{T}_A$, $\bigcirc \circ h : Q \to \mathcal{T}_A \to \mathcal{T}_A$ *is a parameterized applicative functor.*

Proof. This follows from the fact that h is a monoid homomorphism by a simple inspection of the parameters in the laws in figure 11. $\square$

Parameterized applicative functors are closed under composition.

Theorem 3. *Let* $\mathbf{P}$ *be a monoid, and let* $\square, \Diamond : P \to \mathcal{T}_A \to \mathcal{T}_A$ *be parameterized applicative functors. Then* $\bigcirc$ *is a parameterized applicative functor, where* $\bigcirc_p = \square_p \circ \Diamond_p$*, with*

$$u^{\overset{\uparrow}{\circ}} = u^{\overset{\uparrow}{\diamond}\overset{\uparrow}{\square}}$$

$$\begin{aligned}(u)^{\underset{\downarrow}{\circ}}_{p,q} &= (((\lambda x.(x)^{\underset{\downarrow}{\diamond}}_{p,q})^{\overset{\uparrow}{\square}})^{\underset{\downarrow}{\square}}_{0,p}\ u)^{\underset{\downarrow}{\square}}_{p,q} \\ &= \lambda y.[\![\lambda x.x^{\underset{\downarrow}{\diamond}}\ u\ y]\!]\end{aligned}$$

The proof of theorem 3 follows from tedious algebraic manipulation and has been deferred to the appendix.

Note that the definitions of the applicative operations $\cdot^{\overset{\uparrow}{\circ}}$ and $\cdot^{\underset{\downarrow}{\circ}}$ given in theorem 3 are just the parameterized versions of the ones given by McBride and Paterson (2008) in their construction for the composition of two non-parameterized applicative functors.

3.2 Why monoids?

It may seem strange that the parameters should be monoidal. This is made more natural when we consider an alternative presentation of applicative functors in terms of *monoidal functors*, presented in (McBride and Paterson, 2008), and explored in greater detail in (Paterson, 2012). This makes specific reference to the fact that the space of types and terms is itself monoidal with respect the standard cartesian operations (product $\times$ and unit $\mathbf{1}$). In addition to a map $\bigcirc : (A \to B) \to \bigcirc A \to \bigcirc B$ which is part of every functor, a monoidal functor $\bigcirc$ also has the following operations.

- $0 : \bigcirc \mathbf{1}$
- $+ : \bigcirc A \to \bigcirc B \to \bigcirc (A \times B)$

The laws which 0 and + must satisfy require $\mathbf{1}$ to be a unit for $\times$. Paterson (2012) shows the equivalence between the presentation given previously (based on (McBride and Paterson, 2008)) and this one.[11] Of course, the benefit of the applicative functor presentation is that it requires only implication.

Parameterized applicative functors then give rise to the following operations.

- $0 : \bigcirc_0 \mathbf{1}$
- $+ : \bigcirc_p A \to \bigcirc_q B \to \bigcirc_{p+q} (A \times B)$

Here one sees immediately that the behaviour of the parameters exactly mirrors the behaviour of the types – in 0, the parameter is the unit, as is the type, and in + the parameters are multiplied together just as are the types. Indeed, the product of two monoids is itself a monoid (with operations defined pointwise), and so a parameterized monoidal functor can be viewed simply as a (non-parameterized) monoidal functor whose domain is a product monoid.

4 Implementing cooper storage

Cooper storage is here reconceptualized in terms of parameterized applicative functors, with parameters representing the types of the contents of the store.

[11] The applicative functor operations are interdefinable with these, as follows ($\mathbb{K} = \lambda x, y.x$, $(,) = \lambda x, y.\langle x, y\rangle$, $\mathsf{uncurry} = \lambda f, x.f(\pi_1 x)(\pi_2 x)$, $\mathsf{app} = \lambda x, y.xy$, and $\langle\rangle$ is the empty tuple – the monoidal unit for the product operation).

$$\begin{aligned} u^{\overset{\uparrow}{\circ}} &= \bigcirc(\mathbb{K}u)\ 0 & 0 &= \langle\rangle^{\overset{\uparrow}{\circ}} \\ u^{\underset{\downarrow}{\circ}}\ v &= \bigcirc(\mathsf{app} \circ \mathsf{uncurry})\ (u + v) & u + v &= (\!|(,)\ u\ v|\!) \end{aligned}$$

Section 4.1, begins with the case of unbounded cooper storage (where there is no *a priori* size limit on how large a store can be), which is followed in §4.2 by nested cooper storage (Keller, 1988), and in §4.6 by finite cooper storage. Section 4.3 presents a useful sequent calculus notation for cooper storage.

4.1 Basic cooper storage

I define here two maps $\Diamond, \Box : P \to \mathcal{T}_A \to \mathcal{T}_A$, where P is the free monoid over types, $P = \mathcal{T}_A^*$, together with the associated applicative functions. The map $\Diamond_w$, for $w \in P$, is intended to represent the type of an expression of type α, which contains $|w|$ free variables of types $w_1, \ldots, w_{|w|}$, intended to be bound by elements in the cooper store.

Definition 4.

$$\Diamond_\epsilon \alpha = \alpha$$
$$\Diamond_{aw} \alpha = a \to \Diamond_w \alpha$$

It is convenient to represent terms of applicative functor type in a uniform way, one which facilitates both the visualization of the relevant manipulations of these terms, as well as comparison to more traditional cooper-storage notation; this will be explored in more depth in §4.3. I will write $x_1 : u_1, \ldots, x_n : u_n \vdash_\Diamond M : \alpha$ as a representation of the term $\lambda x_1, \ldots, x_n.M : \Diamond_{u_1 \ldots u_n} \alpha$.

Definition 5.

$$M^{\overset{\uparrow}{\Diamond}} = M$$
$$(M)^{\overset{\Diamond}{\downarrow}}_{u,v}\ N = \lambda x_1, \ldots, x_{|u|}, y_1, \ldots, y_{|v|}.M\ x_1\ \ldots\ x_{|u|}\ (N\ y_1\ \ldots\ y_{|v|})$$

Note that for $M : \alpha$, $M^{\overset{\uparrow}{\Diamond}} = \ \vdash_\Diamond M : \alpha$, and that, leaving parameters implicit, $(\Gamma \vdash_\Diamond M : \alpha \to \beta)^{\overset{\Diamond}{\downarrow}}\ (\Delta \vdash_\Diamond N : \alpha) = \Gamma, \Delta \vdash_\Diamond (M\ N) : \beta$.

Theorem 6. *$\Diamond$ is a parameterized applicative functor.*

The map $\Box_w$, for $w \in P$, is intended to represent the type of an expression which is associated with a store containing $|w|$ elements of types $w_1, \ldots, w_{|w|}$. One way of implementing this idea (suggested in the introduction) is to encode a store as an n-tuple of expressions, $\langle M_1, \ldots, M_n \rangle$. Instead, I will encode products using implication in the standard way, using continuations; a pair $\langle M, N \rangle$ is encoded as the term $\lambda k.kMN$. When $M : \alpha$ and $N : \beta$, $\lambda k.kMN : (\alpha \to \beta \to o) \to o$ for some type o.

Definition 7.

$$\Box_w \alpha := (\Diamond_{w\alpha}\ o) \to o$$

Something of type $\Box_w \alpha$ is a term of type $(w_1 \to \ldots \to w_{|w|} \to \alpha \to o) \to o$. Again, as a convenient notation, $N_1 : u_1, \ldots, N_n : u_n \vdash_\Box M : \alpha$ represents the term $\lambda k.kN_1 \ldots N_n M : \Box_{u_1 \ldots u_n} \alpha$.

Definition 8.

$$M^{\overset{\uparrow}{\Box}} = \lambda k.kM$$

$$\begin{aligned}(M)^{\underset{\downarrow}{\Box}}_{u,v}\ N = \lambda k.M(\lambda x_1, \ldots, x_{|u|}, m.\\ N(\lambda y_1, \ldots, y_{|v|}, n.kx_1 \ldots x_{|u|} y_1 \ldots y_{|v|}(mn)))\end{aligned}$$

Note that, once again, for $M : \alpha$, $M^{\overset{\uparrow}{\Box}} = \ \vdash_{\Box} M : \alpha$, and that, leaving parameters implicit, $(\Gamma \vdash_{\Box} M : \alpha \to \beta)^{\underset{\downarrow}{\Box}}\ (\Delta \vdash_{\Box} N : \alpha) = \Gamma, \Delta \vdash_{\Box} (M\ N) : \beta$.

Theorem 9. $\Box$ *is a parameterized applicative functor.*

Let $\mathtt{t} : \mathcal{T}_A \to \mathcal{T}_A$ be arbitrary. The type $\mathtt{t}_\alpha$, to be read as "the trace type of α", is intended to represent the type of a variable which is to be bound by an expression in the store of type α. Let map extend functions over a set X homomorphically over X^*. By theorems 2 and 6, $\Diamond \circ \mathsf{map}\ \mathtt{t}$ is a parameterized applicative functor. The desired structure is the composition of $\Box$ and $(\Diamond \circ \mathsf{map}\ \mathtt{t})$.

Definition 10.

$$\bigcirc_w := \Box_w \circ \Diamond_{\mathsf{map}\ \mathtt{t}\ w}$$

$$M^{\overset{\uparrow}{\circ}} = M^{\overset{\uparrow}{\Diamond}\overset{\uparrow}{\Box}} = \lambda k.kM$$

$$\begin{aligned}(M)^{\underset{\downarrow}{\circ}}_{u,v}\ N &= [\![\lambda x.(x)^{\underset{\downarrow}{\Diamond}}_{u,v}\ M\ N]\!]\\ &\equiv \lambda k.M(\lambda x_1, \ldots, x_{|u|}, m.\\ &\qquad N(\lambda y_1, \ldots, y_{|v|}, n.\\ &\qquad\quad kx_1 \ldots x_{|u|} y_1 \ldots y_{|v|}(\lambda p_1, \ldots, p_{|u|}, q_1, \ldots, q_{|v|}.\\ &\qquad\qquad mp_1 \ldots p_{|u|}(nq_1 \ldots q_{|v|}))))\end{aligned}$$

An expression of type $\bigcirc_w \alpha$ has type $(w_1 \to \ldots \to w_{|w|} \to (\mathtt{t}_{w_1} \to \ldots \to \mathtt{t}_{w_{|w|}} \to \alpha) \to o) \to o$. While the sequent-like notation suggested previously would yield $N_1 : u_1, \ldots, N_n : u_n \vdash_{\Box} (x_1 : \mathtt{t}_{u_1}, \ldots, x_n : \mathtt{t}_{u_n} \vdash_{\Diamond} M : \alpha) : \Diamond_{u_1 \ldots u_n}\alpha$, it is more convenient to write instead the following, which takes advantage of the fact that the parameters are shared across the two composands of $\bigcirc_w = \Box_w \circ \Diamond_{\mathsf{map}\ \mathtt{t}\ w}$:

$$[N_1 : u_1]_{x_1}, \ldots, [N_n : u_n]_{x_n} \vdash_{\bigcirc} M : \alpha$$

Then for $M : \alpha$, $M^{\overset{\uparrow}{\circ}} = \ \vdash_{\bigcirc} M : \alpha$, and, still leaving parameters implicit, $(\Gamma \vdash_{\bigcirc} M : \alpha \to \beta)^{\underset{\downarrow}{\circ}}\ (\Delta \vdash_{\bigcirc} N : \alpha) = \Gamma, \Delta \vdash_{\bigcirc} (M\ N) : \beta$.

Corollary 11. $\bigcirc$ *is a parameterized applicative functor.*

Corollary 11 demonstrates that expressions of type $\bigcirc_w\alpha$ can be manipulated as though they were of type α. This is only half of the point of cooper storage. The other half is that the store must be manipulable; expressions should be able to be put into (storage) and taken out of (retrieval) the store.

Formulating these operations at first in terms of the sequent representation is more congenial to intuition. First, with retrieval, given an expression $\Gamma, [M : \alpha]_x, \Delta \vdash_{\bigcirc} N : \beta$, the goal is to combine M with $\lambda x.N$ to obtain a sequent of the form $\Gamma, \Delta \vdash_{\bigcirc} fM(\lambda x.N) : \gamma$, where $f : \alpha \to (\mathtt{t}_\alpha \to \beta) \to \gamma$ is some antecedently given way of combining expressions M and $\lambda x.N$. In the canonical case, $\alpha = (e \to t) \to t$, $\mathtt{t}_\alpha = e$, $\beta = t$, and f is function application.[12]

Definition 12.

$$\begin{aligned}
&\mathsf{retrieve}_{\bigcirc}\ u\ \alpha\ v : (\alpha \to (\mathtt{t}_\alpha \to \beta) \to \gamma) \to \bigcirc_{u\alpha v}\beta \to \bigcirc_{uv}\gamma\\
&\mathsf{retrieve}_{\bigcirc}\ u\ \alpha\ v\ f\ M\\
&\qquad = \lambda k.M(\lambda x_1,\ldots,x_{|u|},n,y_1,\ldots,y_{|v|},m.\\
&\qquad\qquad kx_1\ldots x_{|u|}y_1\ldots y_{|v|}(\lambda p_1,\ldots,p_{|u|},q_1,\ldots,q_{|v|}.\\
&\qquad\qquad\qquad fn(\lambda r.mp_1\ldots p_{|u|}rq_1\ldots q_{|v|})))
\end{aligned}$$

An expression which cannot be interpreted in its surface position must be put into the store, until such time as it can be retrieved. In the sequent-style notation, $\vdash_{\bigcirc} M : \alpha$ is mapped to $[M : \alpha]_x \vdash_{\bigcirc} x : \mathtt{t}_\alpha$; an expression of type $\bigcirc_0\alpha$ can be turned into one of type $\bigcirc_\alpha\mathtt{t}_\alpha$ simply by putting the expression itself into the store.

Definition 13.

$$\begin{aligned}
\mathsf{store}_{\bigcirc} &: \bigcirc_0\alpha \to \bigcirc_\alpha\mathtt{t}_\alpha\\
\mathsf{store}_{\bigcirc}\ M &= \lambda k.kM(\lambda x.x)
\end{aligned}$$

This is not faithful to Cooper's original proposal, as here only expressions associated with empty stores are allowed to be stored. Cooper's original proposal simply copies the main expression of type α directly over to the store. From the perspective advocated for here, this cannot be done simply because there *is* no closed term of type α in an expression of type $\bigcirc_w\alpha$;[13] only closed terms of type $\Diamond_w\alpha$ and of type w_i, for $1 \leq i \leq |w|$, are guaranteed to exist.[14] This is taken up again in the next section.

[12]While the operations and types involving cooper storage are linear, there is no such guarantee about the objects being so manipulated. A natural way to think about this involves treating the types being manipulated as abstract types (as in abstract categorial grammars (de Groote, 2001a)), the internal details of which are irrelevant to the storage mechanisms.

[13]Except in the uninteresting case where $w_i = \alpha$ for some i.

[14]A misguided attempt to generalize the current proposal to arbitrary stores is, when attempting to store something of type $\bigcirc_{uv}\alpha = \Box_{uv}(\Diamond_{uv}\alpha)$, to put the entire expression of type $\Diamond_{uv}\alpha$ into the store (Kobele, 2006). This would yield an alternative storage operator $\mathsf{store'}_{\bigcirc}\ u\ v : \bigcirc_{uv}\alpha \to \bigcirc_{u(\Diamond_{uv}\alpha)v}\mathtt{t}_{\Diamond_{uv}\alpha}$. (The given $\mathsf{store}_{\bigcirc}$ would correspond to $\mathsf{store'}_{\bigcirc}\ e\ e$.) While such a generalization is logically possible, it is problematic in the sense that there is no obvious way for the other elements in the store to bind what should intuitively be their arguments, which have been abstracted over in the newly stored expression.

4.2 Nested stores

Cooper's original proposal, in which syntactic objects with free variables were manipulated, suffered from the predictable difficulty that occasionally variables remained free even when the store was emptied. In addition to being artificial (interpreting terms with free variables requires making semantically unnatural distinctions), this is problematic because the intent behind the particular use of free variables in cooper storage is that they should ultimately be bound by the expression connected to them in the store.

Keller (1988) observed that Cooper's two-step generate-and-test semantic construction process could be replaced by a direct one if the store data type was changed from a list of expressions to a forest of expressions. An expression was stored by making it the root of a tree whose daughters were the trees on its original store. Thus if an expression in the store contained free variables, they were intended to be bound by expressions it dominated. An expression could only be retrieved if it were at the root of a tree. These restrictions together ensured that no expression with free variables could be associated with an empty store.

From the present type-theoretic perspective, the structure of the store must be encoded in terms of types. The monoid of parameters is still based on sequences (with the empty sequence being the identity element of the monoid), except that now the elements of these sequences are not types, but trees of types.[15] The operation `rt` maps a tree to (the label of) its root, and `dtrs` maps a tree to the sequence of its daughters. Given a tree $t = a(t_1, \ldots, t_n)$, $\mathtt{rt}\ t = a$, and $\mathtt{dtrs}\ t = t_1, \ldots, t_n$.

Note the following about nested stores. First, all and only the roots of the trees in the store bind variables in the main expression. Second, for each tree in the store, the expression at any node in that tree may bind a variable only in the expressions at nodes dominating it. These observations motivate the following type definitions.

As the type of the main expression is determined by the types of the traces of the roots of the trees in the sequence only, the type function $\Diamonddot$ can be defined in terms of $\Diamond$ in the previous section, and is by theorem 2 itself a parameterized applicative functor.

Definition 14.

$$\Diamonddot = \Diamond \circ (\mathsf{map}\ \mathsf{t}) \circ (\mathsf{map}\ \mathtt{rt})$$

In contrast to the previous, non-nested setting, an expression in the store may very well be an expression with an associated store (and so on). This is reflected in terms of the set of parameters having a recursive structure. Accordingly, the type function for stores ($\boxdot$) is defined in terms of the type function for (nested) cooper storage ($\odot$), which is, just as before, the composition of $\boxdot$ and $\Diamonddot$.

[15]More precisely, $P = \epsilon \mid \mathcal{T}_A(P), P$ is a forest of unranked trees. For $a, b, c, d \in \mathcal{T}_A$, ϵ, $a(\epsilon)$, and $a(b(\epsilon), c(\epsilon)), d(\epsilon)$ are elements of P. The term $a(\epsilon)$ will be written as a, and so these elements of P will be represented rather as ϵ, a, and $a(b, c), d$.

Definition 15.

$$\begin{aligned}\odot_{\epsilon}\alpha &= \alpha \\ \odot_{w}\alpha &= \boxdot_{w}(\dot{\Diamond}_{w}\alpha)\end{aligned}$$

$$\boxdot \ := \square \circ (\mathsf{map}\ (\lambda t.\odot_{\mathtt{dtrs}\ t}\mathtt{rt}\ t))$$

Given a parameter $w = w_1 \cdots w_k$, where $w_i = a_i(t^i_1, \ldots, t^i_{n_i})$ for each $1 \leq i \leq k$,

$$\odot_w \alpha = (\odot_{t^1_1 \cdots t^1_{n_1}} a_1 \to \cdots \to \odot_{t^k_1 \cdots t^k_{n_k}} a_k \to \Diamond_{a_1 \cdots a_k}\alpha \to o) \to o$$

As before, a sequent-style notation aids the understanding; observe that sequents for nested stores have as antecedents sequents for nested stores! A sequent $\vdash_{\odot} M : \alpha$ represents an expression $\overline{\vdash_{\odot} M : \alpha} = \lambda k.kM : \odot_{0}\alpha = (\alpha \to o) \to o$, and a sequent $[\Gamma_1 \vdash_{\odot} M_1 : a_1]_{x_1}, \ldots, [\Gamma_n \vdash_{\odot} M_n : a_n]_{x_n} \vdash_{\odot} M : \alpha$ represents an expression $\lambda k.k(\overline{\Gamma_1 \vdash_{\odot} M_1 : a_1}) \ldots (\overline{\Gamma_n \vdash_{\odot} M_n : a_n})(\lambda x_1, \ldots, x_n.M)$.

The type function $\odot : P \to \mathcal{T}_A \to \mathcal{T}_A$ is a parameterized applicative functor; indeed, modulo the types, its applicative functor operations are the same as those of $\bigcirc$.

In the nested context, storage is straightforward, and fully general; it should simply map an expression $\Gamma \vdash_{\odot} M : \alpha$ to $[\Gamma \vdash_{\odot} M : \alpha]_x \vdash_{\odot} x : \mathtt{t}_{\alpha}$. Indeed, this is just $\mathsf{store}_{\bigcirc}$ at every parameter value:

Definition 16.

$$\begin{aligned}\mathsf{store}_{\odot}\ w &: \odot_w\alpha \to \odot_{\alpha(w)}\mathtt{t}_{\alpha} \\ \mathsf{store}_{\odot}\ w\ M &:= \lambda k.kM(\lambda x.x)\end{aligned}$$

Retrieval should, given a mode of combination $f : \alpha \to (\mathtt{t}_{\alpha} \to \beta) \to \gamma$, turn an expression $\Gamma, [\Xi \vdash_{\odot} M : \alpha]_x, \Delta \vdash_{\odot} N : \beta$ into $\Gamma, \Xi, \Delta \vdash_{\odot} fM(\lambda x.N) : \gamma$.

Definition 17.

$$\begin{aligned}&\mathsf{retrieve}_{\odot}\ u\ \alpha(w)\ v : (\alpha \to (\mathtt{t}_{\alpha} \to \beta) \to \gamma) \to \odot_{u\alpha(w)v}\beta \to \odot_{uwv}\gamma \\ &\mathsf{retrieve}_{\odot}\ u\ \alpha(w)\ v\ f\ M \\ &\qquad = \lambda k.M(\lambda x_1, \ldots, x_{|u|}, N, y_1, \ldots, y_{|v|}, m. \\ &\qquad\qquad N(\lambda z_1, \ldots, z_{|w|}, n. \\ &\qquad\qquad\qquad kx_1 \ldots x_{|u|} \\ &\qquad\qquad\qquad\ z_1 \ldots z_{|w|} \\ &\qquad\qquad\qquad\ y_1 \ldots y_{|v|} \\ &\qquad\qquad\qquad\ (\lambda p_1, \ldots, p_{|u|}, q_1, \ldots, q_{|w|}, r_1, \ldots, r_{|v|}. \\ &\qquad\qquad\qquad\qquad f\ (nq_1 \ldots q_{|w|}) \\ &\qquad\qquad\qquad\qquad\quad (\lambda x.mp_1 \ldots p_{|u|}xr_1 \ldots r_{|v|}))))\end{aligned}$$

4.3 Sequent notation for cooper storage

The cooper storage idiom is succinctly manipulated using the sequent notation, as presented in figure 12. It is easy to see that basic cooper storage (§4.1) is the special case of nested cooper storage (§4.2) where the store rule requires that Γ be empty. Somewhat perversely, the usual natural deduction proof system for minimal (implicational) logic can be viewed as the special case of the system in figure 12, where 1. $\mathtt{t}_{\alpha\to\alpha} = \alpha$, 2. the axiom rule at a type α is simulated by first injecting the identity function at type α using $\Uparrow$, and then using the store rule, and 3. implication introduction is simulated by the rule of retrieval, constrained in such a manner as to always use the Church encoding of zero as its first argument (i.e. $M = \lambda x, y.y$). Alternatively, the cooper storage system is just the usual natural deduction system where assumptions are associated with closed terms, and upon discharge of an assumption its associated term is applied to the resulting term.

$$\frac{M : \alpha}{\vdash_{\#} M : \alpha}\Uparrow \qquad \frac{\Gamma \vdash_{\#} M : \alpha \to \beta \qquad \Delta \vdash_{\#} N : \alpha}{\Gamma, \Delta \vdash_{\#} MN : \beta}\Downarrow$$

$$\frac{\Gamma \vdash_{\#} M : \alpha}{[\Gamma \vdash_{\#} M : \alpha]_x \vdash_{\#} x : \mathtt{t}_\alpha}\,\mathsf{store}$$

$$\frac{M : \alpha \to (\mathtt{t}_\alpha \to \beta) \to \gamma \qquad \Gamma, [\Xi \vdash_{\#} N : \alpha]_x, \Delta \vdash_{\#} O : \beta}{\Gamma, \Xi, \Delta \vdash_{\#} M\ N\ (\lambda x.O) : \gamma}\,\mathsf{retrieve}$$

Figure 12: A sequent notation for cooper storage

Moving away from the implementation of these abstract operations in the previous sections, observe that a sequent $\Gamma \vdash_{\#} M : \alpha$ corresponds to an expression of a particular type in the following way.

$$\mathtt{ty}([\Gamma_1 \vdash_{\#} M_1 : \alpha_1]_{x_1}, \ldots, [\Gamma_n \vdash_{\#} M_n : \alpha_n]_{x_n} \vdash_{\#} M : \alpha) = \#_{\alpha_1(\mathtt{ty}(\Gamma_1))+\cdots+\alpha_n(\mathtt{ty}(\Gamma_n))}\alpha$$

In other words, the monoid of parameters of the expression is determined by the types of the elements in the antecedent, and the comma (,) connective in the antecedent corresponds to the $\mathtt{+}$ operation in the monoid of parameters, with the empty antecedent corresponding to the monoidal $\mathtt{0}$.

The sequent representation facilitates proving a type function $\bigcirc$ to be a parameterized applicative functor.

Definition 18. Given a monoid P, a sequent representation is determined by a set Φ of possible antecedent formulae and a function $\mathsf{ty} : \Phi \to P$. The extension of ty over sequences of elements of Φ is also written ty.

As an example, the set $\Phi_\Diamond$ of possible antecedent formulae for the function $\Diamond$ is $X \times \mathcal{T}_A$, and $\mathsf{ty}(\langle x, \alpha\rangle) = \alpha$. In the case of $\Box$, $\Phi_\Box = \{\langle M, \alpha\rangle : \ \vdash M : \alpha\}$, and $\mathsf{ty}(\langle M, \alpha\rangle) = \alpha$.

Definition 19. Given a monoid P, *an interpretation* of a sequent representation determined by Φ is a map $\phi : \Phi^* \times \Lambda \to \Lambda$.

In the case of $\Diamond$, $\phi_\Diamond(\langle x_1, \alpha_1\rangle, \ldots, \langle x_n, \alpha_n\rangle, M) = \lambda x_1, \ldots, x_n.M$. For the case of $\Box$, $\phi_\Box(\langle N_1, \alpha_1\rangle, \ldots, \langle N_n, \alpha_n\rangle, M) = \lambda k.kN_1 \ldots N_nM$.

Definition 20. Given a map $\bigcirc : P \to \mathcal{T}_A \to \mathcal{T}_A$, an interpretation ϕ *respects* $\bigcirc$ just in case for any sequent $\Gamma \vdash M : \alpha$, $\phi(\Gamma, M) : \bigcirc_{\mathsf{ty}(\Gamma)}\alpha$. An interpretation ϕ is *full* for $\bigcirc$ just in case for all parameters p and types α, for every $M : \bigcirc_p\alpha$, there is some sequent $\Gamma \vdash N : \alpha$ such that $\phi(\Gamma, N) \equiv_{\beta\eta} M$. An interpretation is *complete* for $\bigcirc$ just in case it respects $\bigcirc$ and is full for $\bigcirc$.

It is straightforward to see that $\phi_\Diamond$ and $\phi_\Box$ are complete for $\Diamond$ and $\Box$ respectively. Respect is immediate. Fullness follows from the fact that the sequence representations can be viewed (via $\phi_\Diamond$ and $\phi_\Box$) as manipulating η-long forms: given a term of type $\Diamond_{\alpha_1,\ldots,\alpha_n}\alpha = \alpha_1 \to \cdots \to \alpha_n \to \alpha$, its η-long form has the shape $\lambda x_1, \ldots, x_n.N$, and similarly, for a term of type $\Box_{\alpha_1,\ldots,\alpha_n}\alpha = (\alpha_1 \to \cdots \to \alpha_n \to \alpha \to o) \to o$, its η-long form has the shape $\lambda k.kN_1 \ldots N_nN$. (Recall that these are linear terms, whence k does not occur free in any $N_1, \ldots, N_n, N$.) Both long forms are the images under $\phi_\Diamond$ (resp. $\phi_\Box$) of the sequent $\psi_1, \ldots, \psi_n \vdash N$, where ψ_i is $\langle x_i, \alpha_i\rangle$ (resp. $\langle N_i, \alpha_i\rangle$).

Theorem 21. *Given a complete sequent representation for $\bigcirc$, if $\phi(\Gamma \vdash_\bigcirc M : \alpha) \equiv_{\beta\eta} \phi(\Gamma \vdash_\bigcirc N : \alpha)$ whenever $M \equiv_{\beta\eta} N$, then $\bigcirc$ is an applicative functor with operations $\cdot^{\overset{\uparrow}{\circ}}$ and $\cdot^{\overset{\circ}{\downarrow}}$.*

Proof. As the sequent representation is complete for $\bigcirc$, expressions of type $\bigcirc_p\alpha$ can be converted back and forth to sequents of the form $\Gamma \vdash M : \alpha$, where $\mathsf{ty}(\Gamma) = p$.

Thus, by inspection of figure 12, and making implicit use of conversions between expressions and sequents, observe that $\cdot^{\overset{\uparrow}{\circ}} : \alpha \to \bigcirc_{\mathsf{0}}\alpha$, and that $(\cdot)^{\overset{\circ}{\downarrow}}_{\mathsf{ty}(\Gamma),\mathsf{ty}(\Delta)} : \bigcirc_{\mathsf{ty}(\Gamma)}(\alpha \to \beta) \to \bigcirc_{\mathsf{ty}(\Delta)}\alpha \to \bigcirc_{\mathsf{ty}(\Gamma)+\mathsf{ty}(\Delta)}\beta$.

I now show that the four applicative functor equations are satisifed. I assume function extensionality (that $f = g$ iff for all x, $fx = gx$), and convert implicitly between terms and sequents. The types of expressions in the sequent representation is suppressed for concision.

identity: $(\texttt{id}^{\overset{\uparrow}{\circ}})^{\overset{\circ}{\downarrow}}_{\mathsf{0},p} = \texttt{id}$

$$\dfrac{\dfrac{\texttt{id}}{\vdash_\bigcirc \texttt{id}}\,\overset{\uparrow}{\circ} \qquad \Gamma \vdash_\bigcirc M}{\dfrac{\Gamma \vdash_\bigcirc \texttt{id}\ M}{\Gamma \vdash_\bigcirc M}\,\equiv_{\beta\eta}}\,\overset{\circ}{\downarrow} \;=\; \texttt{id}(\Gamma \vdash_\bigcirc M) = \Gamma \vdash_\bigcirc M$$

composition: $(((\circ^{\overset{\uparrow}{\circ}})^{\overset{\circ}{\downarrow}}_{\mathsf{0},p}\ u)^{\overset{\circ}{\downarrow}}_{p,q}\ v)^{\overset{\circ}{\downarrow}}_{p+q,r} = (u)^{\overset{\circ}{\downarrow}}_{p,q+r} \circ (v)^{\overset{\circ}{\downarrow}}_{q,r}$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\circ}{\vdash_{\circ} \circ}\,\overset{\uparrow}{\circ} \qquad \Gamma \vdash_{\circ} M}{\Gamma \vdash_{\circ} \circ\ M}\,\underset{\downarrow}{\circ} \qquad \Delta \vdash_{\circ} N}{\Gamma, \Delta \vdash_{\circ} M \circ N}\,\underset{\downarrow}{\circ} \qquad \Xi \vdash_{\circ} O}{\Gamma, \Delta, \Xi \vdash_{\circ} (M \circ N)O}\,\underset{\downarrow}{\circ}}{\Gamma, \Delta, \Xi \vdash_{\circ} M(NO)}\,\equiv_{\beta\eta}$$

$$=$$

$$\dfrac{\Gamma \vdash_{\circ} M \qquad \dfrac{\Delta \vdash_{\circ} N \qquad \Xi \vdash_{\circ} O}{\Delta, \Xi \vdash_{\circ} NO}\,\underset{\downarrow}{\circ}}{\Gamma, \Delta, \Xi \vdash_{\circ} M(NO)}\,\underset{\downarrow}{\circ}$$

homomorphism: $(f^{\overset{\uparrow}{\circ}})^{\underset{\downarrow}{\circ}}_{0,0}\ x^{\overset{\uparrow}{\circ}} = (f\ x)^{\overset{\uparrow}{\circ}}$

$$\dfrac{\dfrac{f}{\vdash_{\circ} f}\,\overset{\uparrow}{\circ} \qquad \dfrac{x}{\vdash_{\circ} x}\,\overset{\uparrow}{\circ}}{\vdash_{\circ} fx}\,\underset{\downarrow}{\circ} \quad = \quad \dfrac{fx}{\vdash_{\circ} fx}\,\overset{\uparrow}{\circ}$$

interchange: $(u)^{\underset{\downarrow}{\circ}}_{p,0}\ x^{\overset{\uparrow}{\circ}} = ((\lambda f.fx)^{\overset{\uparrow}{\circ}})^{\underset{\downarrow}{\circ}}_{0,p}\ u$

$$\dfrac{\Gamma \vdash_{\circ} M \qquad \dfrac{x}{\vdash_{\circ} x}\,\overset{\uparrow}{\circ}}{\Gamma \vdash_{\circ} Mx}\,\underset{\downarrow}{\circ} \quad = \quad \dfrac{\dfrac{\dfrac{\lambda f.fx}{\vdash_{\circ} \lambda f.fx}\,\overset{\uparrow}{\circ} \qquad \Gamma \vdash_{\circ} M}{\Gamma \vdash_{\circ} (\lambda f.fx)M}\,\underset{\downarrow}{\circ}}{\Gamma \vdash_{\circ} Mx}\,\equiv_{\beta\eta}$$

□

4.4 An example with nesting

The motivating example in §1.1 can be recast using the type theoretical machinery of this section as in figure 13. The parse tree in the figure represents the derivation in which storage takes place at each *DP*. The interesting aspect of the derivation of this sentence lies in the application of the storage rule to the object DP *a judge from every city.* The types of expressions in the sequent notation is suppressed for legibility. The denotation of the *D'* is

$$[\,\vdash_{\odot} \textsf{every city}]_z \vdash_{\odot} \textsf{a(judge} \wedge \textsf{from}\ z)$$

After applying $\textsf{store}_{\odot}$, the denotation of the *DP* is

$$[[\,\vdash_{\odot} \textsf{every city}]_z \vdash_{\odot} \textsf{a(judge} \wedge \textsf{from}\ z)]_x \vdash_{\odot} x$$

The denotation of the lowest *IP* is given below.

$$[[\,\vdash_{\odot} \textsf{every city}]_z \vdash_{\odot} \textsf{a(judge} \wedge \textsf{from}\ z)]_x, [\,\vdash_{\odot} \textsf{no reporter}]_y \vdash_{\odot} \textsf{will(praise}\ x)\ y$$

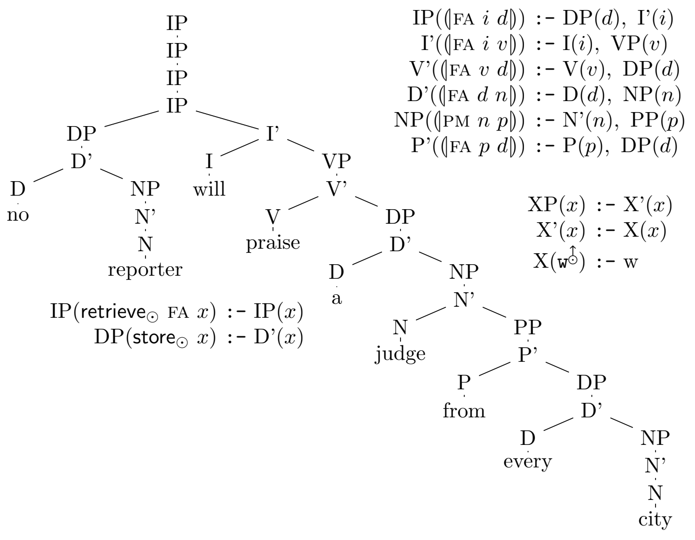

IP((FA i d)) :- DP(d), I'(i)
I'((FA i v)) :- I(i), VP(v)
V'((FA v d)) :- V(v), DP(d)
D'((FA d n)) :- D(d), NP(n)
NP((PM n p)) :- N'(n), PP(p)
P'((FA p d)) :- P(p), DP(d)
XP(x) :- X'(x)
X'(x) :- X(x)
X(w) :- w
IP(retrieve FA x) :- IP(x)
DP(store x) :- D'(x)
no
reporter
will
praise
a
judge
from
every
city

Figure 13: Revisiting the grammar of fig 2 with nested cooper storage

There are exactly two possibilities for retrieval: 1. the subject, or 2. the object. Crucially, the embedded prepositional argument (*every city*) to the object is *not* able to be retrieved at this step. Retrieving the object first, the denotation of the next *IP* node is the below.

$$[\ \vdash_\odot \mathsf{every\ city}]_z\,, [\ \vdash_\odot \mathsf{no\ reporter}]_y \vdash_\odot \mathsf{a}(\mathsf{judge} \wedge \mathsf{from}\ z)(\lambda x.\mathsf{will}(\mathsf{praise}\ x)\ y)$$

There are again two possibilities for retrieval. Retrieving the subject first, the denotation of the penultimate *IP* node is as follows.

$$[\ \vdash_\odot \mathsf{every\ city}]_z \vdash_\odot \mathsf{no\ reporter}(\lambda y.\mathsf{a}(\mathsf{judge} \wedge \mathsf{from}\ z)(\lambda x.\mathsf{will}(\mathsf{praise}\ x)\ y))$$

Finally, the denotation of the entire parse tree is below.

$$\vdash_\odot \mathsf{every\ city}(\lambda z.\mathsf{no\ reporter}(\lambda y.\mathsf{a}(\mathsf{judge} \wedge \mathsf{from}\ z)(\lambda x.\mathsf{will}(\mathsf{praise}\ x)\ y)))$$

This is claimed in the linguistic literature not to be a possible reading of this sentence, as quantificational elements from the same nested sequent (*cities* and *judges*) are separated by an intervening quantificational element from a different nested sequent (*reporter*). Section 4.5 takes this up again.

4.5 Avoiding nesting via composition

Keller (1988) proposes the use of nested stores in particular in the context of noun phrases embedded within noun phrases, as in the example sentences below.

7. *An agent of every company* arrived.

8. They disqualified *a player belonging to every team.*

9. *Every attempt to find a unicorn* has failed miserably.

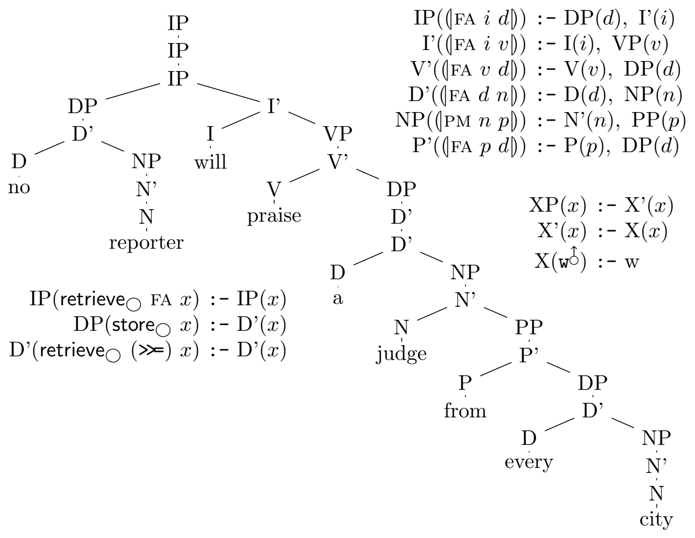
This sort of configuration is widely acknowledged in the linguistic literature as a *scope island*; the scope of a quantified NP external to another cannot intervene between the scope of this other quantified NP and the scope of a quantified contained within it (May and Bale, 2005). In unpublished work, Larson (1985) proposes a version of nested stores which enforces this restriction; upon retrieval of something containing a nested store, all of its sub-stores are recursively also immediately retrieved.

These ideas can be implemented without using nested stores at all, if certain restrictions on types are imposed. First note that the canonical type of expression on stores is $(\alpha \to t) \to t$, for some type α, and designated type t, and that the canonical value of $\mathtt{t}_{(\alpha\to t)\to t}$ is α. Assume for the remainder of this section that all elements in stores have a type of this form, and that $\mathtt{t}$ is as just described. For convenience, I will write $\mathbf{c}\ a$ for $(a \to t) \to t$.

Now consider an expression of type $\bigcirc_u\ \mathbf{c}\ a$ with a simple (i.e. non-nested) store; assume as well $u_i = \mathbf{c}\ a_i$ for each $1 \leq i \leq |u| = n$. This will be represented as a sequent $[\,\vdash_\bigcirc N_1 : u_1]_{x_1}, \ldots, [\,\vdash_\bigcirc N_n : u_n]_{x_n} \vdash_\bigcirc N : \mathbf{c}\ a$. In order to put the main expression into storage using $\mathsf{store}_\bigcirc$, the current store must first be emptied out ($\mathsf{store}_\bigcirc$ requires that $\Gamma = \emptyset$). In order to use $\mathsf{retrieve}_\bigcirc$, some operation $M : u_i \to (\mathtt{t}_{u_i} \to \mathbf{c}\ a) \to \alpha$ must be supplied which allows the retrieved element to be combined with the main expression. As the resulting type should be something which can be stored, $\alpha = \mathbf{c}\ b$ for some b; as the type of an expression in the context of cooper storage should be the same regardless of what it may have in the store, $b = a$. Given the present assumptions about u_i and $\mathtt{t}$, the desired operation has type $\mathbf{c}\ a_i \to (a_i \to \mathbf{c}\ a) \to \mathbf{c}\ a$. Unpacking abbreviations, expressions of type $(a_i \to t) \to t$ and $a_i \to (a \to t) \to t$ should be combined in such a manner as to obtain an expression of type $(a \to t) \to t$. The obvious mode of combination involves composing the first expression with the second with its arguments exchanged (and so of type $(a \to t) \to a_i \to t$); using combinators $\mathbb{B}xyz := x(yz)$ and $\mathbb{C}xyz := xzy$, the desired term is $\lambda x, y.\mathbb{B}x(\mathbb{C}y)$. This is familiar in the programming language theory literature as the bind operation of the continuation monad (Moggi, 1991), and in the linguistic literature as *argument saturation* (Büring, 2004). I will write it here as the infix operator $\gg\!=$.

This procedure can be iterated until the store is empty, and an expression of type $\bigcirc_0((a \to t) \to t)$ remains. The $\mathsf{store}_\bigcirc$ operation can then be applied to this expression. Clearly, the order in which the elements of the store are retrieved is irrelevant to this procedure, although it will of course give rise to different functions (corresponding to different scope-taking behaviours of the stored QNPs).

This is illustrated in figure 14. The grammar of the figure differs from that of figure 13 by the addition of the rule allowing *D'* to be immediately derived from a *D'*, which in turn allows for the recursive retrieval of any elements in storage before an expression can be stored. Recall that $m \gg\!= f = \lambda k.m(\lambda x.fxk)$. The



$$\begin{aligned} \mathrm{IP}((\!|\textsc{fa}\ i\ d|\!)) &\ \texttt{:-}\ \mathrm{DP}(d),\ \mathrm{I'}(i) \\ \mathrm{I'}((\!|\textsc{fa}\ i\ v|\!)) &\ \texttt{:-}\ \mathrm{I}(i),\ \mathrm{VP}(v) \\ \mathrm{V'}((\!|\textsc{fa}\ v\ d|\!)) &\ \texttt{:-}\ \mathrm{V}(v),\ \mathrm{DP}(d) \\ \mathrm{D'}((\!|\textsc{fa}\ d\ n|\!)) &\ \texttt{:-}\ \mathrm{D}(d),\ \mathrm{NP}(n) \\ \mathrm{NP}((\!|\textsc{pm}\ n\ p|\!)) &\ \texttt{:-}\ \mathrm{N'}(n),\ \mathrm{PP}(p) \\ \mathrm{P'}((\!|\textsc{fa}\ p\ d|\!)) &\ \texttt{:-}\ \mathrm{P}(p),\ \mathrm{DP}(d) \end{aligned}$$

$$\begin{aligned} \mathrm{XP}(x) &\ \texttt{:-}\ \mathrm{X'}(x) \\ \mathrm{X'}(x) &\ \texttt{:-}\ \mathrm{X}(x) \\ \mathrm{X}(\mathtt{w}^{\uparrow}_{\bigcirc}) &\ \texttt{:-}\ \mathrm{w} \end{aligned}$$

$$\begin{aligned} \mathrm{IP}(\mathsf{retrieve}_{\bigcirc}\ \textsc{fa}\ x) &\ \texttt{:-}\ \mathrm{IP}(x) \\ \mathrm{DP}(\mathsf{store}_{\bigcirc}\ x) &\ \texttt{:-}\ \mathrm{D'}(x) \\ \mathrm{D'}(\mathsf{retrieve}_{\bigcirc}\ (\gg\!=)\ x) &\ \texttt{:-}\ \mathrm{D'}(x) \end{aligned}$$

Figure 14: Eliminating the need for nested cooper storage

interesting aspect of this derivation centers around the object *D'*. The meaning of the lower of the two nodes labeled *D'*, of type $\bigcirc_{(et)t}(et)t$, is given below.

$$[\vdash_{\bigcirc} \mathsf{every\ city}]_z \vdash_{\bigcirc} \mathsf{a}(\mathsf{judge} \wedge \mathsf{from}\ z)$$

Then retrieval applies, giving rise to the following meaning of the higher *D'* node, which is of type $\bigcirc_{\mathsf{0}}(et)t$.

$$\vdash_{\bigcirc} \mathsf{every\ city} \gg\!= \lambda z.\mathsf{a}(\mathsf{judge} \wedge \mathsf{from}\ z)$$

The denotation of the object *DP* involves simply putting the above meaning into storage.

$$[\vdash_{\bigcirc} \mathsf{every\ city} \gg\!= \lambda z.\mathsf{a}(\mathsf{judge} \wedge \mathsf{from}\ z)]_x \vdash_{\bigcirc} x$$

4.6 Finite stores

Many restrictive grammar formalisms can be viewed as manipulating not strings but rather finite sequences of strings (Vijay-Shanker et al, 1987). In these cases, there is an *a priori* upper bound on the maximal number of components of the tuples. It is natural to attempt to connect the tuples of strings manipulated by the grammar to tuples of semantic values,[16] which then allows the interpretation of the elements on the store to be shifted from one where they are simply being held until they should be interpreted, to one where they are the meanings of particular strings in the string tuple. Now the position of an element in the

[16]In the case of minimalist grammars (Stabler, 1997) this has been made explicit in (Kobele, 2006, 2012).

store becomes relevant; it is the formal link between semantics and syntax. Accordingly, this information should be encoded in the monoid of store types. What is relevant in a store is two-fold: 1. what positions are available, and 2. what is in each position.

The positions available are indexed by a set I of names. The type of a store is given by specifying the type of element at each store position (or that there is no element at a particular position). This is modeled as a partial function $f : I \hookrightarrow \mathcal{T}_A$, where $f(i) = \alpha$ iff there is an expression at position i of type α. Such a function f is occupied at i if $i \in \mathtt{def}(f)$, and thus two compatible functions are never occupied at the same positions.

Intuitively, stores can be combined only if their elements are in complementary positions. In this case, combination is via superposition. The empty store has nothing at any position; its behaviour is given by the everywhere undefined function $\epsilon : I \hookrightarrow \mathcal{T}_A$.

In order to represent finite cooper storage using λ-terms, a linear order must be imposed on the index set I. For convenience, I will be identified with an initial subset of the positive integers with the usual ordering; $[n] = \{1, \ldots, n\}$. Then $f : [n] \hookrightarrow \mathcal{T}_A$ is identified with the sequence $\langle f(1), \ldots, f(n)\rangle \in (\mathcal{T}_A \cup \{\bot\})^n$ (where if $f(i)$ is undefined, then $\bot$ is in the i^{th} position in the sequence). In this notation, two functions $f, g : [n] \hookrightarrow \mathcal{T}_A$ are compatible just in case for every position i at least one of f and g has $\bot$ at that position.

Fix $I = [n]$. For any $S \subseteq I$, and $g : S \hookrightarrow \mathcal{T}_A$, let $\overline{g} : I \hookrightarrow \mathcal{T}_A$ such that $\mathsf{def}(\overline{g}) = \mathsf{def}(g)$ and for all $i \in S$, $\overline{g}(i) = g(i)$. For any sets X, Y and any $x \in X$, $y \in Y$, define $[x \mapsto y] : \{x\} \to Y$ to be the function that maps x to y. Given $f : I \hookrightarrow \mathcal{T}_A$, clearly $f = \overline{[1 \mapsto f(1)]} + \ldots + \overline{[n \mapsto f(n)]}$ (note that for some $i \in I$, $f(i)$ might be $\bot$).

The following type functions are not based on a free monoid of parameters, and thus the easy correlation between left hand sides of sequents and parameters breaks down in this setting. The obvious way to associate a sequent with left hand side $\psi_1, \ldots, \psi_n$ with a parameter is to treat each ψ_i as representing the function $\overline{[i \mapsto \mathsf{ty}(\psi_i)]}$; in other words, the linear order of the left hand side elements indicates the index they are associated with. To represent the function where some index $i \leq n$ is undefined, the set Φ of possible antecedent formulae must include an element $\bot$ representing undefinedness, with $\mathsf{ty}(\bot) = \bot$. There will be then many ways to represent the everywhere undefined function as a sequence of antecedent formulae; to show fullness of the sequent interpretation, the canonical sequent will not have $\bot$ as its right most antecedent formula (and so the everywhere undefined function will have as canonical sequent the one with an empty left hand side).

The definitions of the parameterized applicative functors and associated operations are changed slightly to reflect the different partial monoid of parameters. The symbols $\Diamond$ and $\Box$ are reused (with a different meaning!) in this new setting. For any applicative functor $\#$, $(\cdot)^{\Downarrow}_{u,v}$ is only defined when $u + v$ is.

Definition 22.

$$\Diamond_{\epsilon}\alpha = \alpha$$
$$\Diamond_{\perp w}\alpha = \Diamond_{w}\alpha$$
$$\Diamond_{aw}\alpha = a \to \Diamond_{w}\alpha$$

The set of antecedent formulae are $\Phi_{\Diamond} = (X \times \mathcal{T}_A)_{\perp}$, with $\mathsf{ty}(\langle x, \alpha\rangle) = \alpha$ and $\mathsf{ty}(\perp) = \perp$. The sequent $\psi_1, \ldots, \psi_n \vdash_{\Diamond} M : \alpha$ represents the term $\phi_{\Diamond}(\psi_1, \ldots, \psi_n, M)$ of type $\Diamond_f \alpha$, where $f = \bigoplus_{i=1}^{n} \overline{[i \mapsto \mathsf{ty}(\psi_i)]}$. Here, as before, $\phi_{\Diamond}(M) = M$ and $\phi_{\Diamond}(\langle x, \alpha\rangle, \psi_1, \ldots, \psi_n, M) = \lambda x.\phi_{\Diamond}(\psi_1, \ldots, \psi_n, M)$, while $\phi_{\Diamond}(\perp, \psi_1, \ldots, \psi_n, M) = \phi_{\Diamond}(\psi_1, \ldots, \psi_n, M)$. It is straightforward to see that $\phi_{\Diamond}$ respects $\Diamond$. Fullness again depends on long forms.

Corollary 23. $\Diamond$ *is a parameterized applicative functor.*

Proof. By theorem 21. $\square$

Definition 24.

$$\Box_w \alpha = (\Diamond_{w\alpha}\ o) \to o$$

The set of antecedent formulae are $\Phi_{\Box} = \{\langle M, \alpha\rangle : \ \vdash M : \alpha\}_{\perp}$, with $\mathsf{ty}(\langle M, \alpha\rangle) = \alpha$ and $\mathsf{ty}(\perp) = \perp$. The sequent $\psi_1, \ldots, \psi_n \vdash_{\Box} M : \alpha$ represents the term $\phi_{\Box}(\psi_1, \ldots, \psi_n, M) = \lambda k.\phi'(\psi_1, \ldots, \psi_n, M, k)$ of type $\Box_f \alpha$, where $f = \bigoplus_{i=1}^{n} \overline{[i \mapsto \mathsf{ty}(\psi_i)]}$. Here the last argument of ϕ' plays the role of an accumulator, and so $\phi'(M, N) = NM$, $\phi'(\perp, \psi_1, \ldots, \psi_n, M, N) = \phi'(\psi_1, \ldots, \psi_n, M, N)$, and $\phi'(\langle O, \alpha\rangle, \psi_1, \ldots, \psi_n, M, N) = \phi'(\psi_1, \ldots, \psi_n, M, NO)$. It is again straightforward to see that $\phi_{\Box}$ respects $\Box$. Fullness depends again on long forms.

Corollary 25. $\Box$ *is a parameterized applicative functor.*

Proof. By theorem 21. $\square$

Definition 26. $\bigcirc_w := \Box_w \circ \Diamond_{\mathtt{t}_w}$

Corollary 27. $\bigcirc$ *is a parameterized applicative functor.*

The grammar of figure 5 can be expressed more naturally in terms of finite storage, as illustrated in figure 15. There is still a deal of unnecessary clutter in this figure, which can be rectified, however, once the strings manipulated by the grammar are recast in type theoretic terms (following de Groote (2001a) and others). A string $/abc/$ is viewed as a λ-term of type $\mathbf{str} := s \to s$: $\lambda x^s.a(b(c\ s))$. The empty string $/\epsilon/ := \lambda x^s.x$, and concatenation is function composition: $/abc/^\frown /de/ := /abc/ \circ /de/ = \lambda x^s.a(b(c(d(e\ s))))$. Define $\mathtt{t}_{\mathbf{str}} := \mathbf{str}$, and define $\textsc{up} : \mathbf{str} \to (\mathbf{str} \to \mathbf{str}) \to \mathbf{str}$ such that $\textsc{up}\ w\ f := w^\frown(f\,/\epsilon/)$. Then cooper storage can be used on strings (based on $\bigcirc$). In particular, $\mathsf{store}_{\bigcirc}(\lambda k^{\mathbf{str}\to o}.k\,/w/) = \lambda k.k\,/w/\,(\lambda x^{\mathbf{str}}.x)$. Using cooper storage on the string side as well in figure 15 allows for a simpler presentation of the grammar, and is shown in figure 16. In the figure, it can be seen that there is a deep *symme-*

$$\begin{aligned}
[\alpha;\beta]\,(x\frown y,z)(\llparenthesis \textsc{fa}\ X\ Y\rrparenthesis) &\ \texttt{:-}\ [\texttt{=x}.\alpha]\,(x)(X),[\texttt{x};\beta]\,(y,z)(Y).\\[1ex]
[\alpha;\beta]\,(x,y)(\llparenthesis \textsc{fa}\ X\ (\mathsf{store}_{\bigcirc}\ Y)\rrparenthesis) &\ \texttt{:-}\ [\texttt{=x}.\alpha]\,(x)(X),[\texttt{x}.\beta]\,(y)(Y).\\
[\alpha;\beta]\,(x,y)(\llparenthesis \textsc{fa}\ Y\ X\rrparenthesis) &\ \texttt{:-}\ [\texttt{=x}.\alpha]\,(x)(X),[\texttt{x}.\beta]\,(y)(Y).\\[1ex]
[\alpha]\,(y\frown x)(\mathsf{retrieve}_{\bigcirc}\ \textsc{fa}\ X) &\ \texttt{:-}\ [\texttt{+x}.\alpha;\texttt{-x}]\,(x,y)(X).\\
[\alpha]\,(y\frown x)(X) &\ \texttt{:-}\ [\texttt{+x}.\alpha;\texttt{-x}]\,(x,y)(X).
\end{aligned}$$

$$\begin{aligned}
[\texttt{=n.d.-k}](\mathrm{a})(\llparenthesis\exists\rrparenthesis)\ \texttt{:-}\ . &\qquad [\texttt{=v.i}](\mathrm{to})(\llparenthesis\texttt{id}\rrparenthesis)\ \texttt{:-}\ .\\
[\texttt{n}](\mathrm{dog})(\llparenthesis\mathsf{dog}\rrparenthesis)\ \texttt{:-}\ . &\qquad [\texttt{=v.+k.s}](\mathrm{must})(\llparenthesis\Box\rrparenthesis)\ \texttt{:-}\ .\\
[\texttt{=d.v}](\mathrm{bark})(\llparenthesis\mathsf{bark}\rrparenthesis)\ \texttt{:-}\ . &\qquad [\texttt{=i.v}](\mathrm{seem})(\llparenthesis\mathsf{seem}\rrparenthesis)\ \texttt{:-}\ .
\end{aligned}$$

Figure 15: Recasting the grammar of figure 5 in terms of finite storage

$$\begin{aligned}
[\alpha;\beta]\,(\llparenthesis x\frown y\rrparenthesis)(\llparenthesis \textsc{fa}\ X\ Y\rrparenthesis) &\ \texttt{:-}\ [\texttt{=x}.\alpha]\,(x)(X),[\texttt{x};\beta]\,(y)(Y).\\[1ex]
[\alpha;\beta]\,(\llparenthesis x\frown(\mathsf{store}_{\bigcirc}\ y)\rrparenthesis)(\llparenthesis \textsc{fa}\ X\ (\mathsf{store}_{\bigcirc}\ Y)\rrparenthesis) &\ \texttt{:-}\ [\texttt{=x}.\alpha]\,(x)(X),[\texttt{x}.\beta]\,(y)(Y).\\
[\alpha;\beta]\,(\llparenthesis x\frown(\mathsf{store}_{\bigcirc}\ y)\rrparenthesis)(\llparenthesis \textsc{fa}\ Y\ X\rrparenthesis) &\ \texttt{:-}\ [\texttt{=x}.\alpha]\,(x)(X),[\texttt{x}.\beta]\,(y)(Y).\\[1ex]
[\alpha]\,(\mathsf{retrieve}_{\bigcirc}\ \textsc{up}\ x)(\mathsf{retrieve}_{\bigcirc}\ \textsc{fa}\ X) &\ \texttt{:-}\ [\texttt{+x}.\alpha;\texttt{-x}]\,(x)(X).\\
[\alpha]\,(\mathsf{retrieve}_{\bigcirc}\ \textsc{up}\ x)(X) &\ \texttt{:-}\ [\texttt{+x}.\alpha;\texttt{-x}]\,(x)(X).
\end{aligned}$$

$$\begin{aligned}
[\texttt{=n.d.-k}](\llparenthesis a\rrparenthesis)(\llparenthesis\exists\rrparenthesis)\ \texttt{:-}\ . &\qquad [\texttt{=v.i}](\llparenthesis to\rrparenthesis)(\llparenthesis\texttt{id}\rrparenthesis)\ \texttt{:-}\ .\\
[\texttt{n}](\llparenthesis dog\rrparenthesis)(\llparenthesis\mathsf{dog}\rrparenthesis)\ \texttt{:-}\ . &\qquad [\texttt{=v.+k.s}](\llparenthesis must\rrparenthesis)(\llparenthesis\Box\rrparenthesis)\ \texttt{:-}\ .\\
[\texttt{=d.v}](\llparenthesis bark\rrparenthesis)(\llparenthesis\mathsf{bark}\rrparenthesis)\ \texttt{:-}\ . &\qquad [\texttt{=i.v}](\llparenthesis seem\rrparenthesis)(\llparenthesis\mathsf{seem}\rrparenthesis)\ \texttt{:-}\ .
\end{aligned}$$

Figure 16: Cooper storage on strings *and* meanings

try between the operations on the strings and those on the meanings, which is broken in two instances. The reason for the broken symmetry stems from the fact that meanings, in this grammar, are designed to have a *wider* distribution than strings; a quantifier can be interpreted *either* in its surface position (corresponding to its position in the string), *or* in its deep position (corresponding to its position in the derivation tree). Strings, of course, are only pronounced in their surface positions; this is implemented by the operation $\textsc{up}\ w\ f$, which uniformly puts the string w in its upper, 'surface', position, and the empty string $/\epsilon/$ in the lower, 'deep', position.

The two readings (5 and 6) of sentence 3 are shown in at the top of figure 17. Both derivations give rise to the same pronunciation, the incremental construction of which is shown at the bottom of the figure.

5 Conclusion

I have shown that cooper storage, in many variations, can be given a simple treatment in the linear λ-calculus. Working within the simply typed λ-calculus has forced us to confront and address problems plaguing more traditional presentations involving free variables, which allow for the (undesired) generation of ill-formed meaning representations. One of the interests of linearity lies in

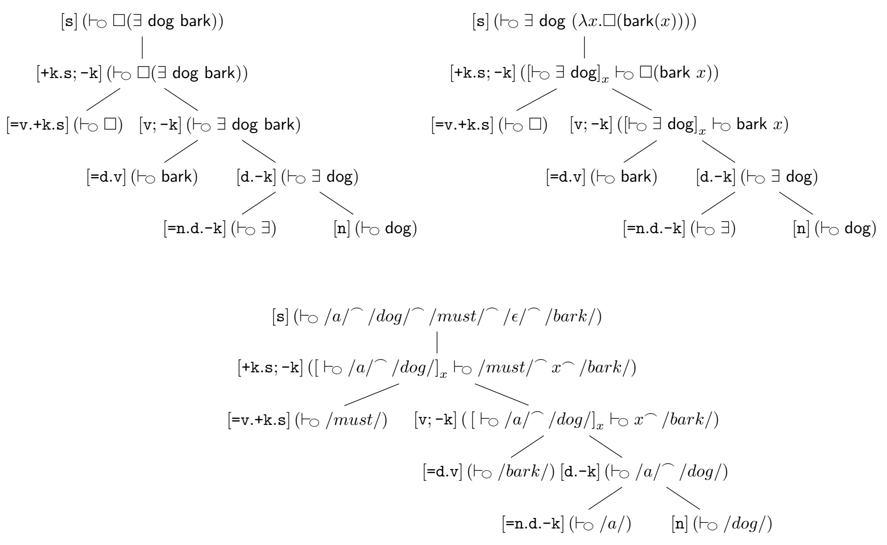


Figure 17: Two readings of 3 (top) and their common pronunciation (bot.)

the fact that linear λ-homomorphisms (acting on trees) are particularly well-behaved for the purposes of parsing and generation (Kanazawa, 2007).

This work allows a straightforward and directly compositional semantics for frameworks utilizing finite cooper storage, such as the minimalist grammar semantics of Kobele (2012). While finite cooper storage may seem somewhat arbitrary, it is here that the type theoretic approach really pays off. By limiting in advance the size of the store,[17] the parameter information can be encoded in the syntactic category. This allows for a truly homomorphic interpretation scheme for tree-like syntactic structures. In contrast, full cooper storage requires a richer, polymorphic, type theory in order to have a finitely presented homomorphic interpretation scheme.

A Proofs

The purpose of this section is to provide a proof of theorem 3, that parameterized applicative functors are closed under composition. It is helpful to first prove a lemma that, for any applicative functor $\Box$, $(\cdot)^{\overset{\uparrow}{\Box}\overset{\Box}{\downarrow}}$ distributes over composition.

Lemma 28. *If $\Box : P \to \mathcal{T}_A \to \mathcal{T}_A$ is a parameterized applicative functor, then for any $p \in P$, $g : \beta \to \gamma$ and $f : \alpha \to \beta$, $((g \circ f)^{\overset{\uparrow}{\Box}})^{\overset{\Box}{\downarrow}}_{0,p} = (g^{\overset{\uparrow}{\Box}})^{\overset{\Box}{\downarrow}}_{0,p} \circ (f^{\overset{\uparrow}{\Box}})^{\overset{\Box}{\downarrow}}_{0,p}$*

[17]Where size is measured in terms of the sum of the sizes of the types is the store; this bounds as well the maximal size of stored types.

Proof.

$$\begin{aligned}
(g^{\overset{\uparrow}{\Box}})^{\overset{\Box}{\downarrow}}_{0,p} \circ (f^{\overset{\uparrow}{\Box}})^{\overset{\Box}{\downarrow}}_{0,p} &= ((\underline{(\circ^{\overset{\uparrow}{\Box}})^{\overset{\Box}{\downarrow}}_{0,0}\ g^{\overset{\uparrow}{\Box}}})^{\overset{\Box}{\downarrow}}_{0,0}\ f^{\overset{\uparrow}{\Box}})^{\overset{\Box}{\downarrow}}_{0,p} && \text{(composition)}\\
&= (\underline{((\circ\ g)^{\overset{\uparrow}{\Box}})^{\overset{\Box}{\downarrow}}_{0,0}\ f^{\overset{\uparrow}{\Box}}})^{\overset{\Box}{\downarrow}}_{0,p} && \text{(homomorphism)}\\
&= ((g \circ f)^{\overset{\uparrow}{\Box}})^{\overset{\Box}{\downarrow}}_{0,p} && \text{(homomorphism)}
\end{aligned}$$

□

Theorem 3 is repeated below.

Theorem 3. *Let* $\mathbf{P}$ *be a monoid, and let* $\Box, \Diamond : P \to \mathcal{T}_A \to \mathcal{T}_A$ *be parameterized applicative functors. Then* $\bigcirc$ *is a parameterized applicative functor, where* $\bigcirc_p = \Box_p \circ \Diamond_p$, *with*

$$\begin{aligned}
u^{\overset{\uparrow}{\bigcirc}} &= u^{\overset{\uparrow}{\Diamond}\overset{\uparrow}{\Box}}\\
(u)^{\overset{\bigcirc}{\downarrow}}_{p,q} &= (((\lambda x.(x)^{\overset{\Diamond}{\downarrow}}_{p,q})^{\overset{\uparrow}{\Box}})^{\overset{\Box}{\downarrow}}_{0,p}\ u)^{\overset{\Box}{\downarrow}}_{p,q}
\end{aligned}$$

The following lemma identifies a useful equality.

Lemma 29. $(u^{\overset{\uparrow}{\bigcirc}})^{\overset{\bigcirc}{\downarrow}}_{0,p} = ((u^{\overset{\uparrow}{\Diamond}})^{\overset{\Diamond}{\downarrow}}_{0,p}{}^{\overset{\uparrow}{\Box}})^{\overset{\Box}{\downarrow}}_{0,p}$

Proof.

$$\begin{aligned}
(u^{\overset{\uparrow}{\bigcirc}})^{\overset{\bigcirc}{\downarrow}}_{0,p} &= (((\lambda x.(x)^{\overset{\Diamond}{\downarrow}}_{0,p})^{\overset{\uparrow}{\Box}})^{\overset{\Box}{\downarrow}}_{0,0}\ (u^{\overset{\uparrow}{\Diamond}\overset{\uparrow}{\Box}}))^{\overset{\Box}{\downarrow}}_{0,p} && \text{(def)}\\
&= (((\lambda x.(x)^{\overset{\Diamond}{\downarrow}}_{0,p})\ u^{\overset{\uparrow}{\Diamond}})^{\overset{\uparrow}{\Box}})^{\overset{\Box}{\downarrow}}_{0,p} && (\text{homomorphism}_{\Box})\\
&= ((u^{\overset{\uparrow}{\Diamond}})^{\overset{\Diamond}{\downarrow}}_{0,p}{}^{\overset{\uparrow}{\Box}})^{\overset{\Box}{\downarrow}}_{0,p} && (\equiv_\beta)
\end{aligned}$$

□

Proof of theorem 3. Note first that $(\cdot)^{\overset{\uparrow}{\bigcirc}} : \alpha \to (\Box_0 \circ \Diamond_0)\alpha$, and that $(\cdot)^{\overset{\bigcirc}{\downarrow}} : (\Box_p \circ \Diamond_p)(\alpha \to \beta) \to (\Box_q \circ \Diamond_q)\alpha \to (\Box_{p+q} \circ \Diamond_{p+q})\beta$, as can be seen by inspection of the definitions.

identity

$$\begin{aligned}
(\texttt{id}^{\overset{\uparrow}{\bigcirc}})^{\overset{\bigcirc}{\downarrow}}_{0,p} &= ((\texttt{id}^{\overset{\uparrow}{\Diamond}})^{\overset{\Diamond}{\downarrow}}_{0,p}{}^{\overset{\uparrow}{\Box}})^{\overset{\Box}{\downarrow}}_{0,p} && \text{(lemma 29)}\\
&= (\texttt{id}^{\overset{\uparrow}{\Box}})^{\overset{\Box}{\downarrow}}_{0,p} && (\text{identity}_{\Diamond})\\
&= \texttt{id} && (\text{identity}_{\Box})
\end{aligned}$$

composition

$$(((\boxed{\circ}\overset{\uparrow}{\circ})^{\overset{\circ}{\downarrow}}_{0,p}\ u)^{\overset{\circ}{\downarrow}}_{p,q}\ v)^{\overset{\circ}{\downarrow}}_{p+q,r} = ((\boxed{((\circ\overset{\uparrow}{\diamond})^{\overset{\diamond}{\downarrow}}_{0,p}\overset{\uparrow}{\square})^{\overset{\square}{\downarrow}}_{0,p}\ u})^{\overset{\circ}{\downarrow}}_{p,q}\ v)^{\overset{\circ}{\downarrow}}_{p+q,r} \qquad \text{(lemma 29)}$$

$$= ((((\boxed{\lambda x.(x)^{\overset{\diamond}{\downarrow}}_{p,q}})\overset{\uparrow}{\square})^{\overset{\square}{\downarrow}}_{0,p}\ ((\boxed{(\circ\overset{\uparrow}{\diamond})^{\overset{\diamond}{\downarrow}}_{0,p}}\overset{\uparrow}{\square})^{\overset{\square}{\downarrow}}_{0,p}\ u))^{\overset{\square}{\downarrow}}_{p,q}\ v)^{\overset{\circ}{\downarrow}}_{p+q,r} \qquad \text{(def } (\cdot)^{\overset{\circ}{\downarrow}}_{p,q}\text{)}$$

$$= (\boxed{((((\lambda x.(x)^{\overset{\diamond}{\downarrow}}_{p,q}) \circ (\circ\overset{\uparrow}{\diamond})^{\overset{\diamond}{\downarrow}}_{0,p})\overset{\uparrow}{\square})^{\overset{\square}{\downarrow}}_{0,p}\ u)^{\overset{\square}{\downarrow}}_{p,q}\ v})^{\overset{\circ}{\downarrow}}_{p+q,r} \qquad \text{(lemma 28)}$$

$$= ((\boxed{(\lambda x.(x)^{\overset{\diamond}{\downarrow}}_{p+q,r})\overset{\uparrow}{\square}})^{\overset{\square}{\downarrow}}_{0,p+q}\ ((\boxed{(((\lambda x.(x)^{\overset{\diamond}{\downarrow}}_{p,q}) \circ (\circ\overset{\uparrow}{\diamond})^{\overset{\diamond}{\downarrow}}_{0,p})\overset{\uparrow}{\square})^{\overset{\square}{\downarrow}}_{0,p}\ u})^{\overset{\square}{\downarrow}}_{p,q}\ v))^{\overset{\square}{\downarrow}}_{p+q,r}$$
$$\text{(def } (\cdot)^{\overset{\circ}{\downarrow}}_{p+q,r}\text{)}$$

$$= (((((\boxed{\circ}\overset{\uparrow}{\square})^{\overset{\square}{\downarrow}}_{0,0}\ (\boxed{\lambda x.(x)^{\overset{\diamond}{\downarrow}}_{p+q,r}})\overset{\uparrow}{\square})^{\overset{\square}{\downarrow}}_{0,p}\ ((((\lambda x.(x)^{\overset{\diamond}{\downarrow}}_{p,q}) \circ (\circ\overset{\uparrow}{\diamond})^{\overset{\diamond}{\downarrow}}_{0,p})\overset{\uparrow}{\square})^{\overset{\square}{\downarrow}}_{0,p}\ u))^{\overset{\square}{\downarrow}}_{p,q})\ v)^{\overset{\square}{\downarrow}}_{p+q,r}$$
$$\text{(composition}_{\square}\text{)}$$

$$= (((((\boxed{(\circ)\ (\lambda x.(x)^{\overset{\diamond}{\downarrow}}_{p+q,r})})\overset{\uparrow}{\square})^{\overset{\square}{\downarrow}}_{0,p}\ (((\boxed{(\lambda x.(x)^{\overset{\diamond}{\downarrow}}_{p,q}) \circ (\circ\overset{\uparrow}{\diamond})^{\overset{\diamond}{\downarrow}}_{0,p}})\overset{\uparrow}{\square})^{\overset{\square}{\downarrow}}_{0,p}\ u))^{\overset{\square}{\downarrow}}_{p,q})\ v)^{\overset{\square}{\downarrow}}_{p+q,r}$$
$$\text{(homomorphism}_{\square}\text{)}$$

$$= ((((\boxed{((\circ)\ (\lambda x.(x)^{\overset{\diamond}{\downarrow}}_{p+q,r})) \circ (\lambda x.(x)^{\overset{\diamond}{\downarrow}}_{p,q}) \circ (\circ\overset{\uparrow}{\diamond})^{\overset{\diamond}{\downarrow}}_{0,p}})\overset{\uparrow}{\square})^{\overset{\square}{\downarrow}}_{0,p}\ u)^{\overset{\square}{\downarrow}}_{p,q}\ v)^{\overset{\square}{\downarrow}}_{p+q,r} \qquad \text{(lemma 28)}$$

$$= ((((\lambda g, h.(((\circ\overset{\uparrow}{\diamond})^{\overset{\diamond}{\downarrow}}_{0,p}\ \boxed{g})^{\overset{\diamond}{\downarrow}}_{p,q}\ \boxed{h})^{\overset{\diamond}{\downarrow}}_{p+q,r})\overset{\uparrow}{\square})^{\overset{\square}{\downarrow}}_{0,p}\ u)^{\overset{\square}{\downarrow}}_{p,q}\ v)^{\overset{\square}{\downarrow}}_{p+q,r} \qquad (\equiv_{\beta,\eta})$$

$$= ((((\boxed{\lambda g, h.(g)^{\overset{\diamond}{\downarrow}}_{p,q+r} \circ (h)^{\overset{\diamond}{\downarrow}}_{q,r}})\overset{\uparrow}{\square})^{\overset{\square}{\downarrow}}_{0,p}\ u)^{\overset{\square}{\downarrow}}_{p,q}\ v)^{\overset{\square}{\downarrow}}_{p+q,r} \qquad \text{(composition}_{\diamond}\text{)}$$

$$= ((((\boxed{(\lambda P.P(\lambda x.(x)^{\overset{\diamond}{\downarrow}}_{q,r}))} \circ \boxed{(\circ) \circ (\circ) \circ (\lambda x.(x)^{\overset{\diamond}{\downarrow}}_{p,q+r})})\overset{\uparrow}{\square})^{\overset{\square}{\downarrow}}_{0,p}\ u)^{\overset{\square}{\downarrow}}_{p,q}\ v)^{\overset{\square}{\downarrow}}_{p+q,r} \qquad (\equiv_{\beta,\eta})$$

$$= ((((\lambda P.P(\boxed{\lambda x.(x)^{\overset{\diamond}{\downarrow}}_{q,r}}))\overset{\uparrow}{\square})^{\overset{\square}{\downarrow}}_{0,p}\ \boxed{((((\circ) \circ (\circ) \circ (\lambda x.(x)^{\overset{\diamond}{\downarrow}}_{p,q+r}))\overset{\uparrow}{\square})^{\overset{\square}{\downarrow}}_{0,p}\ u)})^{\overset{\square}{\downarrow}}_{p,q}\ v)^{\overset{\square}{\downarrow}}_{p+q,r}$$
$$\text{(lemma 28)}$$

$$= (((((\boxed{(\circ)} \circ \boxed{(\circ) \circ (\lambda x.(x)^{\overset{\diamond}{\downarrow}}_{p,q+r})})\overset{\uparrow}{\square})^{\overset{\square}{\downarrow}}_{0,p}\ u)^{\overset{\square}{\downarrow}}_{p,0}\ (\lambda x.(x)^{\overset{\diamond}{\downarrow}}_{q,r})\overset{\uparrow}{\square})^{\overset{\square}{\downarrow}}_{p,q}\ v)^{\overset{\square}{\downarrow}}_{p+q,r}$$
$$\text{(interchange}_{\square}\text{)}$$

$$= (((((\circ)\overset{\uparrow}{\square})^{\overset{\square}{\downarrow}}_{0,p}\ \boxed{((((\circ) \circ (\lambda x.(x)^{\overset{\diamond}{\downarrow}}_{p,q+r}))\overset{\uparrow}{\square})^{\overset{\square}{\downarrow}}_{0,p}\ u)})^{\overset{\square}{\downarrow}}_{p,0}\ \boxed{(\lambda x.(x)^{\overset{\diamond}{\downarrow}}_{q,r})\overset{\uparrow}{\square}})^{\overset{\square}{\downarrow}}_{p,q}\ v)^{\overset{\square}{\downarrow}}_{p+q,r}$$
$$\text{(lemma 28)}$$

$$= ((((\boxed{(\circ)} \circ \boxed{(\lambda x.(x)^{\overset{\diamond}{\downarrow}}_{p,q+r})})\overset{\uparrow}{\square})^{\overset{\square}{\downarrow}}_{0,p}\ u)^{\overset{\square}{\downarrow}}_{p,q}\ (((\lambda x.(x)^{\overset{\diamond}{\downarrow}}_{q,r})\overset{\uparrow}{\square})^{\overset{\square}{\downarrow}}_{0,q}\ v))^{\overset{\square}{\downarrow}}_{p+q,r}$$
$$\text{(composition}_{\square}\text{)}$$

$$= (((\circ\overset{\uparrow}{\square})^{\overset{\square}{\downarrow}}_{0,p}\ \boxed{(((\lambda x.(x)^{\overset{\diamond}{\downarrow}}_{p,q+r})\overset{\uparrow}{\square})^{\overset{\square}{\downarrow}}_{0,p}\ u)})^{\overset{\square}{\downarrow}}_{p,q}\ \boxed{(((\lambda x.(x)^{\overset{\diamond}{\downarrow}}_{q,r})\overset{\uparrow}{\square})^{\overset{\square}{\downarrow}}_{0,q}\ v)})^{\overset{\square}{\downarrow}}_{p+q,r} \qquad \text{(lemma 28)}$$

$$= (((\lambda x.(x)^{\overset{\diamond}{\downarrow}}_{p,q+r})\overset{\uparrow}{\square})^{\overset{\square}{\downarrow}}_{0,p}\ \boxed{u})^{\overset{\square}{\downarrow}}_{p,q+r} \circ (((\lambda x.(x)^{\overset{\diamond}{\downarrow}}_{q,r})\overset{\uparrow}{\square})^{\overset{\square}{\downarrow}}_{0,q}\ \boxed{v})^{\overset{\square}{\downarrow}}_{q,r} \qquad \text{(composition}_{\square}\text{)}$$

$$= (u)^{\overset{\circ}{\downarrow}}_{p,q+r} \circ (v)^{\overset{\circ}{\downarrow}}_{q,r} \qquad \text{(def)}$$

homomorphism

$$\begin{aligned}
(f^{\overset{\uparrow}{\circ}})^{\underset{\downarrow}{\circ}}_{0,0}\ x^{\overset{\uparrow}{\circ}} &= ((f^{\overset{\uparrow}{\diamond}})^{\underset{\downarrow}{\diamond}}_{0,0}{}^{\overset{\uparrow}{\Box}})^{\underset{\downarrow}{\Box}}_{0,0}\ x^{\overset{\uparrow}{\circ}} && \text{(lemma 29)}\\
&= ((f^{\overset{\uparrow}{\diamond}})^{\underset{\downarrow}{\diamond}}_{0,0}{}^{\overset{\uparrow}{\Box}})^{\underset{\downarrow}{\Box}}_{0,0}\ x^{\overset{\uparrow}{\diamond}\overset{\uparrow}{\Box}} && \text{(def)}\\
&= ((f^{\overset{\uparrow}{\diamond}})^{\underset{\downarrow}{\diamond}}_{0,0}\ x^{\overset{\uparrow}{\diamond}})^{\overset{\uparrow}{\Box}} && (\text{homomorphism}_{\Box})\\
&= (f\ x)^{\overset{\uparrow}{\diamond}\overset{\uparrow}{\Box}} && (\text{homomorphism}_{\diamond})\\
&= (f\ x)^{\overset{\uparrow}{\circ}} && \text{(def)}
\end{aligned}$$

interchange

$$\begin{aligned}
(u)^{\underset{\downarrow}{\circ}}_{p,0}\ x^{\overset{\uparrow}{\circ}} &= (((\lambda z.(z)^{\underset{\downarrow}{\diamond}}_{p,0})^{\overset{\uparrow}{\Box}})^{\underset{\downarrow}{\Box}}_{0,p}\ u)^{\underset{\downarrow}{\Box}}_{p,0}\ x^{\overset{\uparrow}{\diamond}\overset{\uparrow}{\Box}} && \text{(def)}\\
&= ((\lambda f.f(x^{\overset{\uparrow}{\diamond}}))^{\overset{\uparrow}{\Box}})^{\underset{\downarrow}{\Box}}_{0,p}\ (((\lambda z.(z)^{\underset{\downarrow}{\diamond}}_{p,0})^{\overset{\uparrow}{\Box}})^{\underset{\downarrow}{\Box}}_{0,p}\ u) && (\text{interchange}_{\Box})\\
&= (((\lambda f.f(x^{\overset{\uparrow}{\diamond}}))^{\overset{\uparrow}{\Box}})^{\underset{\downarrow}{\Box}}_{0,p} \circ ((\lambda z.(z)^{\underset{\downarrow}{\diamond}}_{p,0})^{\overset{\uparrow}{\Box}})^{\underset{\downarrow}{\Box}}_{0,p})\ u && \text{(def } \circ)\\
&= (((\lambda f.f(x^{\overset{\uparrow}{\diamond}})) \circ (\lambda z.(z)^{\underset{\downarrow}{\diamond}}_{p,0}))^{\overset{\uparrow}{\Box}})^{\underset{\downarrow}{\Box}}_{0,p}\ u && \text{(lemma 28)}\\
&= ((\lambda v.(v)^{\underset{\downarrow}{\diamond}}_{p,0}\ x^{\overset{\uparrow}{\diamond}})^{\overset{\uparrow}{\Box}})^{\underset{\downarrow}{\Box}}_{0,p}\ u && (\equiv_{\beta,\eta})\\
&= ((\lambda v.((\lambda f.fx)^{\overset{\uparrow}{\diamond}})^{\underset{\downarrow}{\diamond}}_{0,p}\ v)^{\overset{\uparrow}{\Box}})^{\underset{\downarrow}{\Box}}_{0,p}\ u && (\text{interchange}_{\diamond})\\
&= ((((\lambda f.fx)^{\overset{\uparrow}{\diamond}})^{\underset{\downarrow}{\diamond}}_{0,p})^{\overset{\uparrow}{\Box}})^{\underset{\downarrow}{\Box}}_{0,p}\ u && (\equiv_{\eta})\\
&= ((\lambda f.fx)^{\overset{\uparrow}{\circ}})^{\underset{\downarrow}{\circ}}_{0,p}\ u && \text{(lemma 29)}
\end{aligned}$$

$\Box$

References

Atkey R (2009) Parameterized notions of computation. Journal of Functional Programming 19(3-4):335–376

Babaev AA, Soloviev SV (1982) A coherence theorem for canonical morphisms in cartesian closed categories. Journal of Soviet Mathematics 20:2263–2279

Barendregt H, Dekkers W, Statman R (2013) Lambda Calculus with Types. Cambridge University Press, Cambridge

Barker C, Shan C (2014) Continuations and Natural Language, Oxford Studies in Theoretical Linguistics, vol 53. Oxford University Press

Benton PN, Bierman GM, de Paiva V (1998) Computational types from a logical perspective. Journal of Functional Programming 8(2):177–193

Büring D (2004) Crossover situations. Natural Language Semantics 12(1):23–62

Church A (1940) A formulation of the simple theory of types. Journal of Symbolic Logic 5(2):56–68

Cooper R (1983) Quantification and Syntactic Theory. D. Reidel, Dordrecht

De Groote P, Pogodalla S, Pollard C (2011) About parallel and syntactocentric formalisms: A perspective from the encoding of convergent grammar into abstract categorial grammar. Fundamenta Informaticae 106(2-4):211–231

Fairtlough M, Mendler M (1997) Propositional lax logic. Information and Computation 137(1):1–33

de Groote P (1994) A CPS-translation of the $\lambda\mu$-calculus. In: Tison S (ed) Proceedings of the Colloquium on Trees in Algebra and Programming — CAAP'94, Springer, Lecture Notes in Computer Science, vol 787, pp 85–99

de Groote P (2001a) 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

de Groote P (2001b) Type raising, continuations, and classical logic. In: van Rooy R, Stokhof M (eds) Proceedings of the Thirteenth Amsterdam Colloquium, University of Amsterdam, pp 97–101

Hunter T (2010) Relating movement and adjunction in syntax and semantics. PhD thesis, University of Maryland

Johnson K (2000) How far will quantifiers go? In: Martin R, Michaels D, Uriagereka J (eds) Step by Step: Essays on Minimalist Syntax in Honor of Howard Lasnik, MIT Press, Cambridge, Massachusetts, chap 5, pp 187–210

Kanazawa M (2007) Parsing and generation as datalog queries. In: Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics (ACL), Association for Computational Linguistics, Prague, pp 176–183

Kanazawa M (2009) The pumping lemma for well-nested multiple context-free languages. In: Diekert V, Nowotka D (eds) Developments in Language Theory, Lecture Notes in Computer Science, vol 5583, Springer, Berlin, Heidelberg, pp 312–325

Kanazawa M (2016) Parsing and generation as Datalog query evaluation. The IfCoLog Journal of Logics and their Applications

Keller WR (1988) Nested cooper storage: The proper treatment of quantification in ordinary noun phrases. In: Reyle U, Rohrer C (eds) Natural Language Parsing and Linguistic Theories, no. 35 in Studies in Linguistics and Philosophy, D. Reidel, Dordrecht, pp 432–447

Kobele GM (2006) Generating copies: An investigation into structural identity in language and grammar. PhD thesis, University of California, Los Angeles

Kobele GM (2012) Importing montagovian dynamics into minimalism. In: Béchet D, Dikovsky A (eds) Logical Aspects of Computational Linguistics, Springer, Berlin, Lecture Notes in Computer Science, vol 7351, pp 103–118

Kreisel G, Krivine JL (1967) Elements of Mathematical Logic (Model Theory). North-Holland, Amsterdam

Larson RK (1985) Quantifying into NP. available at: `http://semlab5.sbs.sunysb.edu/~rlarson/qnp.pdf`

May R, Bale A (2005) Inverse linking. In: Everaert M, van Riemsdijk H (eds) The Blackwell Companion to Syntax, vol 2, Blackwell, Oxford, chap 36, pp 639–667

McBride C (2011) Kleisli arrows of outrageous fortune, available at: `https://personal.cis.strath.ac.uk/conor.mcbride/Kleisli.pdf`

McBride C, Paterson R (2008) Applicative programming with effects. Journal of Functional Programming 18(1):1–13

Melliès PA (2016) The parametric continuation monad. Mathematical Structures in Computer Science FirstView:1–30, DOI 10.1017/S0960129515000328

Moggi E (1991) Notions of computation and monads. Information and Computation 93(1):55–92

Montague R (1973) The proper treatment of quantification in ordinary English. In: Hintikka J, Moravcsik J, Suppes P (eds) Approaches to Natural Language, D. Reidel, Dordrecht, pp 221–242

Parigot M (1992) $\lambda\mu$-calculus: An algorithmic interpretation of classical natural deduction. In: Voronkov A (ed) Logic Programming and Automated Reasoning, Springer-Verlag, Berlin Heidelberg, Lecture Notes in Computer Science, vol 624, pp 190–201

Paterson R (2012) Constructing applicative functors. In: Gibbons J, Nogueira P (eds) Mathematics of Program Construction, Lecture Notes in Computer Science, vol 7342, Springer, Berlin, Heidelberg, pp 300–323

Pollard C (2011) Covert movement in logical grammar. In: Pogodalla S, Quatrini M, Retoré C (eds) Logic and Grammar: Essays Dedicated to Alain Lecomte on the Occasion of his 60th Birthday, Lecture Notes in Artificial Intelligence, vol 6700, Springer, pp 17–40

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

Stabler EP (1997) Derivational minimalism. In: Retoré C (ed) Logical Aspects of Computational Linguistics, Lecture Notes in Computer Science, vol 1328, Springer-Verlag, Berlin, pp 68–95

Stabler EP, Keenan EL (2003) Structural similarity within and among languages. Theoretical Computer Science 293:345–363

Vijay-Shanker K, Weir D, Joshi A (1987) Characterizing structural descriptions produced by various grammatical formalisms. In: Proceedings of the 25th Meeting of the Association for Computational Linguistics, pp 104–111