Notre Dame Journal of Formal Logic
Volume 37, Number 2, Spring 1996

Dynamic Bracketing and Discourse Representation

ALBERT VISSER and KEES VERMEULEN

Abstract In this paper we describe a framework for the construction of entities that can serve as interpretations of arbitrary contiguous chunks of text. An important part of the paper is devoted to describing stacking cells, or the proposed meanings for bracket-structures.

1 Introduction

Motto: Sentence structure and text structure are different, but not in kind.

1.1 Dynamic brackets in action Let's start with an example. Consider the sentence,

A dog sees a cat.

To give a logical semantics for this sentence, we have to produce a meaning for the sentence. Such a meaning could be (given by) the following sentence of predicate logic.

$$\exists x(\mathrm{DOG}(x) \wedge \exists y(\mathrm{SEES}(x, y) \wedge \mathrm{CAT}(y)))$$

Even if this result were a satisfactory meaning representation, we should not be content. We do not just want correct meanings to be produced in an oracular way. We want the *process* of producing a meaning from a sentence to be systematic. Being systematic involves precise specification of the interpretation process and satisfaction of certain constraints. One such constraint is compositionality. Another such constraint is maximizing the number of meaningful components. Yet another one—subordinate to, but not a consequence of compositionality—is uniformity in the way the meanings interact. In its usual formulation Montague grammar does not meet the uniformity constraint, but we could try to set it up uniformly with function application as the fundamental mode of meaning interaction.

Traditionally the process of interpretation has two stages. The first stage, parsing, is still at the syntactical level. It consists of enriching the input sentence with syntactical structure. We analyze what the appropriate *components* are and the way

in which these components depend on each other. For example our sentence could be parsed as one of,

(i) ((a dog) (sees (a cat))) (ii) ((a dog) \ sees / (a cat))
(iii) (a(dog,sees(a cat))) (iv) (sees(a dog, a cat))

The second stage is semantical interpretation proper. Grammatical structure steers this process. It is what makes the compositionality constraint meaningful. We interpret componentwise, and the meaning of the whole is obtained from the meaning of the parts, by applying the appropriate function to the meanings of the parts in the way prescribed by the grammar. E.g., the meaning of *sees* in our example (ii) above could be a binary function which is applied to the meanings of *a dog* and *a cat*, where the slashes are indicative of argument location. Grammar is *syncategorematic* in this approach to semantics, i.e., no semantical objects are ascribed to the symbols fixing grammatical structure. In our examples: the brackets and the slashes get no meanings.

Why do we arrange the interpretation as we do? A number of the ideas—like compositionality—that go into it can be viewed as general design constraints. They do not reflect anything out there, but just fix a format for describing things. Other things could be dictated by the idea that we want to model something. Dynamic semantics as we view it is shaped by one such idea. We want the way the logical semantics is produced to model the interpretation process in humans and machines. This programmatic idea will cause us to diverge from the received idea of the role of grammar as syncategorematic steering. (Note that "modeling the interpretation process" is not among the classical aims of model theory; those aims are rather to gain understanding of validity and definability. Thus nothing we say should be constructed as criticism of model theory.)

Does the interpretation process as programmed by the grammatical analysis of, say, example (i) reflect the actual temporal interpretation process? This analysis prescribes that we first interpret *(a dog)* and *(a cat)*. Then we process *(sees (a cat))* and finally *((a dog) (sees (a cat)))*. Suppose we are hearing someone saying very slowly: *a … dog … sees … a … cat*. Our "theory" predicts that after hearing *sees* we cannot combine the meaning of *sees* with the meaning of *a dog*. But, surely, we can. The point can be strengthened by looking at very long sentences.

If we accept this argument, there are two ways to go. First we may search for a grammar that reflects the process of interpretation over time more adequately. But in our example, what else could such a grammar yield than,

((((a dog) sees) a) cat)

Is this really convincing? We also would understand something, if we missed the speakers first words and just heard: *… sees … a … cat*. Surely, interpretation satisfies the *Break-in Principle*: we can break into a piece of ongoing text at any place and still gain a measure of understanding. The second possibility is to drop the treatment of grammar as syncategorematic. Grammar is not what steers the interpretation process. It does something else, which is reflected at the semantical level. For, where else could it be reflected?

In this paper we will consider the idea that grammar is there for *categorematical steering*. In other words, *yes*, grammar plays the role of guiding the way we process information, but, *no*, grammar's role is not well placed at the transition from syntax to semantics.

The semantics that we want to develop is a version of Heim's file change semantics for indefinites (see [3],[4]).[1] In our version, the meaning of *(* is going to be *introduce a new file for storing the subsequent information*. The action that *(* means will be modeled by an appropriate mathematical object, in the style of program-semantics. Analogously, a right bracket is going to mean *eliminate the current file*.

To understand the idea of brackets as actions or program-instructions better, it helps to consider an analogy: *existential quantification*. In dynamic semantics the existential quantifier $\exists x$ is usually interpreted as the instruction *introduce a new file labeled x* (see e.g., Groenendijk and Stokhof [2]). Vermeulen [19] modified this to: *push a new file onto the stack labeled x*.[2] The stacking way of viewing the existential quantifier opens the way for introducing a companion of *exists x*, viz., *exit x*, meaning: *pop the current file from the stack labeled x*. Vermeulen's alternative predicate logic is called *DPLE*. By way of example, we produce a sentence in *DPLE*-language, written with four different notational conventions, each suggestive in its own way.

$$
\begin{array}{ll}
(a) & \exists x.P(x).\exists y.Q(x,y).\mathsf{E}x.R(y).\mathsf{E}y \\
(b) & push_x.P(x).push_y.Q(x,y).pop_x.R(y).pop_y \\
(c) & \backslash begin\{x\}.P(x).\backslash begin\{y\}.Q(x,y).\backslash end\{x\}.R(y).\backslash end\{y\} \\
(d) & [_x.P(x).[_y.Q(x,y)._x].R(y)._y]
\end{array}
$$

In contrast to predicate logic, where the existential quantifier is standardly associated with scoping brackets, *exists* and *exit* are their own brackets. As suggested by (d), $[_x$ and $_x]$ are brackets enclosing a stretch of text in which the information stored under *x* goes to a certain fixed file. But if we can view *exists* and *exit* as brackets, where these brackets are given instructions as meanings, why should we not seriously consider giving the usual brackets a similar semantics?

Our first programmatic point was the idea of modeling the interpretation process. With the example of the existential quantifier a second theme has been tacitly introduced. The aim of the dynamic interpretation of the existential quantifier was to provide a better simulation of the way anaphoric phenomena are handled in natural language. Anaphoricity is typically a text phenomenon, which exceeds the scope of individual sentences. Thus dynamic semantics aims at describing not just interpretation of sentences, but primarily interpretation of texts. Sentence interpretation just appears as a subproblem. Note that, because texts can be arbitrarily long, there is no temptation to interpret "text-brackets" like *a man* or *suppose* syncategorematically.[3] If we treat grammar categorematically, and if the syntax-to-semantics interpretation process is not guided by grammatical structure, what is the syntax-to-semantics interpretation process going to look like? Setting apart all kinds of hybrid approaches, let's look at just the most radical one. The radical answer is simply that we can interpret any stretch or chunk of text, and that the interpretation of the concatenation of chunks is a function of the interpretation of the chunks. We will call this function the merger. We will use "$\bullet$" to designate the merger.

Let's look at an example. We are going to parse *a dog sees a cat* as:

$$(.(.sub.\mathrm{a}_x.\mathrm{dog}.).\mathrm{sees}.(.ob.\mathrm{a}_y.\mathrm{cat}.).)$$

This formula is a formula of the fragment of predicate logic we are going to develop. *sub* and *ob* are markers for the argument places. We can both interpret $(.(.sub.a_x.dog.).sees$, getting as meaning, roughly, *a dog sees something*, and $sees.(.ob.a_y.cat.).)$, getting as meaning, roughly, *something sees a cat*. Obviously, to make this all work out well, we should demand that the result of merging the meaning of $(.(.sub.a_x.dog.).sees$ with the meaning of $(.ob.a_y.cat.).)$ is the same as the result of merging the meaning of $(.(.sub.a_x.dog.)$ with the meaning of $sees.(.ob.a_y.cat.).)$. Thus we demand that $\bullet$ is associative. We will conveniently add an empty meaning or *tabula rasa*. This *tabula rasa* will act as the identity for $\bullet$. So our meanings will form a monoid with *tabula rasa* as the identity. We call the interpretation process, as described, *monoidal processing*. Note that monoidal processing includes the possibility of *incremental processing*, i.e., processing strictly from left to right.

In the most radical case, where we interpret all syntax categorematically, there will be no syncategorematic syntax at all. Thus our approach has as consequence a radical unburdening of the specification language. All sentences of this language are grammatical and can be assigned meanings. Of course, some meanings are more equal than the others….

In this paper we will address the problem of interpretation from parsed sentence to semantical object. We will not consider the problem of run-time parsing. We will, however, in designing our specification language, pause to consider variants that would make the parsing easier. (See e.g., §2.4 on the use of lazy brackets.) Some of the work on incremental grammars (see e.g., Milward [12],[13]) is close in spirit to what we are aiming at.

1.2 Context and content In the previous subsection we introduced the first design feature of our approach: grammatical structure is treated as meaningful. In this subsection, we describe the second feature: the *DRT*-style representation of meanings as context/content pairs.

In Groenendijk and Stokhof's *DPL* (see [2]), dynamic meanings are actions, which are in their turn mathematically represented as input-output relations. This approach has the advantage of mathematical simplicity. It has as disadvantages that one cannot associate a good notion of information growth to it and that one cannot easily separate the statical and the dynamical aspects. We follow another dynamic tradition, *DRT* or *File-change Semantics*, in taking our meanings to be static objects (relational databases, sets of assignments), enriched by dynamic contexts (see e.g., Zeevat [22] and Kamp [9]). We claim the following advantages.

- ▷ There is a good separation between the static and the dynamic. We keep the classical ideas of a meaning as a database and of a meaning as a set of assignments.
- ▷ Our approach supports a good notion of information growth.
- ▷ We do not throw away the relational approach. From a *DRT*-style meaning a *DPL*-style relational meaning can be 'extracted'. The 'extraction'-function will be morphism of monoids, mapping $\bullet$ to relational composition, $\circ$.

- ▹ In a way similar to that of the previous item, we can associate *update functions* with our meanings.[4]

The main novelty of this paper is the machinery we develop to build the dynamic contexts. We will begin the development of our tool-kit in §3.

1.3 The local and the global In §1.1, we already mentioned the structure of larger-than-sentence discourses. Evidently, anaphoric phenomena belong to this structure. In the present paper we will give a treatment of anaphoric phenomena, which can be viewed, very roughly, as a *DRT*-version of Vermeulen's *DPLE*. We will, on the other hand, treat local sentential structure in a new way. The most salient property is that our specification language embodies a different, more natural-language like, strategy to handle argument places than Predicate Logic. In Predicate Logic terms get into the correct argument place by occurring at rigidly prescribed places after atomic predicate symbols. In our approach terms get into place by carrying the appropriate place markers (argument handlers). These place markers are analogous to prepositions in, say, Dutch or to the *casi* in, say, Latin. In our language, the following items will be essentially equivalent.

- ▹ (.(.he$_x$.*sub*.).cut.(.the$_y$.bread.*ob*.).(.with.a$_z$.knife.).)
- ▹ (.(.with.a$_z$.knife.).(.the$_y$.bread.*ob*.).(.(.he$_x$.*sub*.).cut.)

The role of *sub* and *ob* is the same as the role of *with*. E.g., *sub* is like the *casus nominativus* in Latin.

By a mechanism to be explained in §4.2, we will see that *he*$_x$ functions as a *link* between the global discourse structure (which involves a discourse referent labeled x) and the local sentential structure (which involves a discourse referent that fuses with the discourse referent associated to the argument handler *sub*). We submit that in this way our semantics for the first time correctly describes one major aspect of anaphors: that they function as places where a local and a global machinery link up. Standard *DRT* and *DPL* could not do this since their specification language uses the mechanism of Predicate Logic for handling arguments. In Predicate Logic there is nothing like the *role* of bringing an argument to its proper place. There an argument simply is in place by being written in the proper place.

1.4 On the use of categories One likely obstacle to reading the paper for the reader whose roots are in linguistics is our use of Category Theory. We feel that the use of this machinery was forced on us by the material. The categorical framework seems tailor-made for the description of the flow of files. To be more precise, we do not want just descriptions, we want descriptions such that objects described that way have certain desirable properties, the most important one being that our objects interact as the elements of a monoid. Moreover, our monoids will be monoids only *modulo* isomorphism. Again, Category Theory is the appropriate medium to describe these isomorphisms in a systematic way. So there is no way to escape Categories. Let's stress, however, that Category Theory in our paper functions just as a definitional format. We do not really use any deep or hard theory. We have added a brief introduction to categories (§3) to ease the pain. We have tried to keep the paper readable by suppressing certain essentially trivial but lengthy computations.

2 Monoids and structure: simple stacking cells

Motto: Don't be afraid of flatness!

2.1 Introduction One of the problems that one might expect for our setup is the representation of (hierarchical/constituent/component/recursive/bracket) structure: since we have set out to describe the whole interpretation process in terms of monoids, there seems to be little room to account for the hierarchical structure that is so abundantly present in most syntactic and semantic phenomena. After all, the monoidal operation is *associative*, which means that the elements of a monoid are insensitive to structure.

However, it turns out that the notion of a *stacking cell* comes to rescue the here.[5] We will see that stacking cells form a monoid, as required. But at the same time they allow us to encode the structural properties of objects. This means that we can introduce structure in the monoidal setup by using stacking cells as contexts.

As an example we will consider the following sentence:

> The quick brown fox that jumped over the lazy dog wanted the rabbit that ran.

Before we can start to interpret this sentence, it will be necessary to make some of the information about its syntactic structure explicit. Here we focus on the *constituent structure* of the sentence, which we make explicit by adding brackets, as follows:

> ((the quick brown fox ((that) jumped (over the lazy dog))) wanted (the rabbit ((that) ran)))

This is not the representation of constituent structure as it will be produced by the ultimately correct theory of syntax. But that is not the point here. The point is that even the ultimately correct representation will encode information about constituent structure in some way or other. And we will use stacking cells in the processing of that ultimately correct representation. As we do not wish to wait for that ultimately correct representation, we illustrate the use of stacking cells using the naive representation, with brackets.

Now we find ourselves confronted with a bracketed string in which different items convey different kinds of information. We have isolated the structural information in the brackets,) and (. The other elements of the string convey other kinds of information that, for now, we will group under a common heading: (truth conditional) *content*. In our (left-to-right) interpretation of this string we keep score of the different kinds of contribution of the string components at the same time. The content-like contributions will be 'added up' according to their location in the contexts: this corresponds to our view of the role of grammar in categorematic steering. Therefore we work with objects which consist of a context component, which serves to keep score of the structural information that we meet, and a content component, in which we add up the content-like information according to its place in the context. These context-content pairs have to form a monoid.

It fits into our program (as explained above) to try to construct this monoid of complex objects from simple(r) monoids: the monoid of contexts and the monoids of contents. Here we first discuss the monoid of contexts, i.e. the monoid which we will use to represent the structural, constituent-like information. In §4 we show how simple monoids can be combined into complex ones. Then, in §6, we will discuss the

content components in some detail, so that we will have all the ingredients required for the interpretation of our example.

We have used brackets to mark the boundaries of the constituents in the sentence. Thus the brackets are the elements in the example that give the information about the structure of the expression. The other elements give other kinds of information altogether. Therefore we may first concentrate on the string:

(.(.1.1.1.1.(.(.1.).1.(.1.1.1.1.).).).1.(.1.1.(.(.1.).1.).).)

instead of the complete example above. This string is obtained from the example by replacing everything but the brackets with 1, some *tabula rasa* element that is structurally neutral. This way we can concentrate on the structural information in our example.[6]

2.2 Pair representation of simple stacking cells We now have to develop a suitable monoidal representation for the kind of strings that we saw above (cf. page 327). For each substring its representation has to encode the impact of the substring on the structure in which it occurs. As a first attempt we consider the following method of representation.

We imagine ourselves working on a *stack* of constituents in each stage of the interpretation. The stack shows how deeply the constituent that we are currently working on is nested in the overall structure. For example, if our string starts with (((, we will obtain a *stack* consisting of three constituents. It is clear that a left bracket, (, indicates the beginning of a new constituent. Each left bracket causes an increase in the depth of nesting of constituents by one: it is a push action. So it seems that the contribution of each left bracket can be described by the integer +1, to indicate that it adds one new constituent to the current stack of constituents.

For the right bracket) the situation is dual: the right bracket indicates a decrease in the nesting depth by one: it is a pop action. So it seems that the contribution of the left bracket can be indicated by the integer -1. Also the stacks themselves can be represented as integers: we can map each stack to the number of levels on the stack. So the monoid of integers + addition seems a suitable candidate for the representation of bracket strings: stacks get represented by the number of push levels that they contain and strings get represented as the sum of the contributions of the brackets in the string: (()) corresponds to $1+1+1+(-1)+(-1)=1$, () to $1+(-1)=0$ etc. But this representation of bracket strings will not work. Let's compare the following two strings: () and)(. If we apply the method of representation indicated above, we find that both strings correspond to 0. Thus this method of representation suggests that both strings are structurally neutral. It will be clear that this is not true: although both strings leave the *amount* of constituents intact, they do not have the same effect on the structure at all. The string () really does have a neutral contribution to the overall structure: if we add () to some string *s*, then we will first start a new constituent with (and then finish this constituent with). As a result we end up in the same constituent where we were after *s*. But if we add)(to a string *s*, things are different. Now we will first finish a constituent (of *s*) with) and then start a *new* one with (. So)(will cause us to *switch* from one constituent to the next.

Clearly such switches will be important for the interpretation of our example. Therefore the representation of the structural contribution of bracket strings by integers is too naive: it is not only the number of brackets that matters, but also their order.

Fortunately it is possible to get away with an almost equally natural representation: we will not represent bracket strings by one integer, but by two natural numbers.[7] One number will be used to indicate the number of constituents that are closed off by the string, the other number gives the number of new, nested constituents that the string introduces. By keeping these two effects separate, we will be able to distinguish the effect of () and)(:

- ▷ (can now be represented as $\langle 0, 1\rangle$,
- ▷) as $\langle 1, 0\rangle$,
- ▷ () as $\langle 0, 0\rangle$ and
- ▷)(as $\langle 1, 1\rangle$.

We can go on and interpret arbitrary strings built up from)'s, ('s and 1's as such pairs $\langle n, m\rangle$. To get a 'monoidal' picture of this interpretation of strings we have to supply an operation of adding up—or merging—the pairs. This is achieved by the following definition:[8]

$$\langle n_1, n_2\rangle \bullet \langle m_1, m_2\rangle = \langle n_1 + (m_1 \dot{-} n_2), m_2 + (n_2 \dot{-} m_1)\rangle.$$

Some examples:

- ▷ $((((\cdot)) \leftrightsquigarrow ((,$ since $\langle 0, 4\rangle \bullet \langle 2, 0\rangle = \langle 0, 2\rangle$
- ▷ $)) \cdot ((((\leftrightsquigarrow))((((,$ since $\langle 2, 0\rangle \bullet \langle 0, 4\rangle = \langle 2, 4\rangle$
- ▷ $((\cdot)))) \leftrightsquigarrow))$, since $\langle 0, 2\rangle \bullet \langle 4, 0\rangle = \langle 2, 0\rangle$

The examples show how the second string will first pop all the constituents that the first string has introduced. Then, if the second string still has some)-brackets left, these are simply added to those of the first string: this is why we have $n_1 + (m_1 \dot{-} n_2)$ in the definition. Dually, if any (-brackets are left of the first string, then these are simply added to the second string: $m_2 + (n_2 \dot{-} m_1)$. This turns out to be the suitable view of the role of the brackets in our set up: we will represent each bracket string by two natural numbers which can be added/merged as indicated above. The first number represents the negative effect of the string, the second number its positive contribution.

It is not hard to check that this gives us a monoid. We find that the operation $\bullet$, as defined above, is associative and the tuple $\langle 0, 0\rangle$ is a unit element of the $\bullet$ operation (and hence we can use it as the 1 that we needed in our example).

Proposition 2.1 $\langle \omega \times \omega, \bullet, \langle 0, 0\rangle\rangle$ *is a monoid.*

We call such tuples $\langle n_1, n_2\rangle$ *simple stacking cells* (SSCs) and we will use them to encode the structural properties of expressions.[9] This monoid is called $\mathcal{SSC}_{pair}$, the simple stacking cells represented as pairs.

2.3 Stacking cells as partial functions There is a slightly different way of looking at SSCs that will turn out to be quite convenient later on: we can look at SSCs as partial injections on the natural numbers.

Definition 2.2

1. A simple stacking cell represented as a partial injection (SSC_{inj}) a is a partial function $a : \omega \rightarrowtail\!\!\!\!\!\rightarrow \omega$ such that:
   $dom(a) = \{n_a, n_a + 1, \ldots\}$ for some $n_a \in \omega$
   $a(n_a + k) = a(n_a) + k$ for all $k \in \omega$
2. The monoid of simple stacking cells as partial injections, $\mathcal{SSC}_{inj}$, is defined as $\mathcal{SSC}_{inj} = \langle \mathrm{SSC}_{inj}, \circ, \mathfrak{id} \rangle$, where $\circ$ stands for composition of (partial) functions[10] and $\mathfrak{id}$ is the identity function on ω.

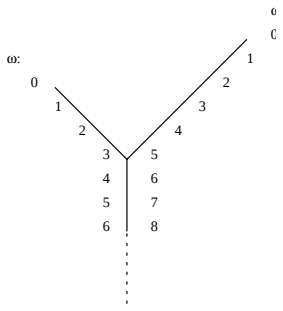


Figure 1: Simple stacking cell

Note that such a partial function a is completely fixed by the choice of n_a and $a(n_a)$ (which are equal to 3 and 5 resp. in Figure 1). In this way we get a correspondence between the partial injections as defined here and the pairs of natural numbers as introduced above.

Fact 2.3 *The mapping $\rho : SSC_{inj} \rightarrow \omega \times \omega$, defined by $\rho(a) = \langle n_a, a(n_a) \rangle$ induces an isomorphism of monoids $\rho : \mathcal{SSC}_{inj} \rightarrow \mathcal{SSC}_{pair}$.*

Once this isomorphism has been established it is no longer necessary to distinguish carefully between $\mathcal{SSC}_{inj}$ and $\mathcal{SSC}_{pair}$. In what follows we simply talk about $\mathcal{SSC}$, "the" monoid of simple stacking cells.

One clear advantage of the functional representation of simple stacking cells is the elegant definition of the monoidal operation: it is simply composition of partial functions. This is not only an advantage because it is an extremely familiar operation, but also because it is immediately clear that it is associative.

We see in Figure 2, for example, that the first cell maps 6 to 4, the second cell maps 4 to 3. Therefore in the resulting cell 6 is mapped to 3. We can also read off that the first number in the domain of the resulting cell is 5 and that 5 will be sent to 2 (via 3).

2.4 Excursion: L-monoids Above we have constructed several representations of simple stacking cells. In the constructions involved we have used the natural numbers with the usual notions of addition and cut-off substraction as a starting point. But it turns out that the constructions can already be carried out in a slightly more general situation: they work for any L-monoid.

Definition 2.4 An L-monoid is a structure $\mathcal{M} = \langle M, \bullet, \leftarrow, \mathfrak{id} \rangle$ such that $\langle M, \bullet, \mathfrak{id} \rangle$ is a monoid and the following additional requirements are met.

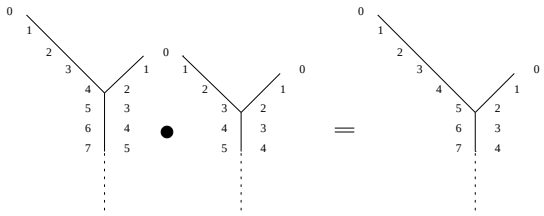


Figure 2: Merging simple stacking cells

Define: $x \leq y \iff$ for some u : $u \bullet y = x$.

L1 $\quad x \bullet z = y \bullet z \Rightarrow x = y$

L2 $\quad x \bullet y = \mathfrak{id} \Rightarrow y = \mathfrak{id}$

L3 $\quad x \bullet y \leq z \iff x \leq z \leftarrow y$

An L-monoid is a monoid with an additional operation $\leftarrow$. Condition L3 says that $\leftarrow$ is a *left-implication*, whence the L in L-monoid. In the literature (see Pratt [15], Moortgat and Oehrle [14]) the operation is also known as left residuation. It is closely related to the notion of an adjoint in category theory (cf. MacLane [10]). The analogy with implication becomes clear as soon as we consider L3 for a Boolean (or Heyting) algebra $\mathcal{B} = \langle B, \wedge, \top \rangle$: now L3 reads as $x \wedge y \leq z \iff x \leq z \leftarrow y$, so $\leftarrow$ is the Boolean implication.[11]

If we regard the $\bullet$-operation in the monoid as an operation for the addition of information, then the operation $\leftarrow$ can also be seen as a sort of directed *substraction* operation: if we take the monoid $\omega = \langle \omega, +, 0 \rangle$, then the left-implication is cut-off substraction. So in general one can try to think of $(m \leftarrow n)$ as *m minus n*.

One important example of an L-monoid is $\langle \omega, +, \dot{-}, 0 \rangle$. And the next example is not far away: we obtain an L-monoid $\mathcal{M}_\lambda = \langle M, \bullet, \leftarrow, \mathfrak{id} \rangle$ for any limit ordinal λ if we set:[12]

- $M = \{\alpha | \ \alpha < \lambda\}$
- $\alpha \bullet \beta = \beta + \alpha$
- $\mathfrak{id} = 0$.

We then find that $\alpha \leq_M \beta$ iff $\beta \leq_{ord} \alpha$. This results in the following definition for the left implication, which we will write as $\dot{-}$, a generalization of cut-off substraction for arbitrary ordinals:

- $0 \dot{-} \gamma = 0$
- $(\alpha + 1) \dot{-} \gamma = (\alpha \dot{-} \gamma) + 1$ if $\gamma \leq_{ord} \alpha$
- $(\alpha + 1) \dot{-} \gamma = (\alpha \dot{-} \gamma) = 0$ if $\alpha <_{ord} \gamma$
- $\mu \dot{-} \gamma = sup_{ord}\{\alpha \dot{-} \gamma \mid \alpha <_{ord} \mu\}$ for limit ordinals μ.

We can see our pair representation of simple stacking cells introduced above as a special case of the construction of stacking cells over an arbitrary L-monoid.

Definition 2.5 ($\mathcal{SSC}_\mathcal{M}$) For any L-monoid $\mathcal{M}$ we define the simple stacking cells over $\mathcal{M}$, $\mathcal{SSC}_\mathcal{M}$, as follows:

$$\mathcal{SSC}_\mathcal{M} = \langle M \times M, \bullet, \langle \mathfrak{id}, \mathfrak{id} \rangle \rangle$$

where $\langle x', x\rangle \bullet \langle y', y\rangle = \langle (y' \leftarrow x) \bullet x', (x \leftarrow y') \bullet y\rangle$.

The definition of $\bullet$ can be understood by direct analogy with the example of the bracket strings (substitute $+$ for $\bullet$ and $\dot{-}$ for $\leftarrow$), but we can also try to get a more general feeling for what is going on in terms of substraction and addition of information. Recall our remark above that $\bullet$ can be seen as addition of information and $\leftarrow$ as substraction of information. The pairs $\langle x', x\rangle$ tell us to first substract information x' from the context and then add information x to it. In $\langle x', x\rangle \bullet \langle y', y\rangle$ we perform such an operation twice: first for $\langle x', x\rangle$ and then for $\langle y', y\rangle$. This has the overall effect that we will substract at least x' from the context. Then we will provisionally add information x, but immediately after that we will substract y'. Finally we add information y.

In case the M we start out with is a linear order (as in the examples above), we know that either $(x \leftarrow y') = \mathfrak{id}$ or $(y' \leftarrow x) = \mathfrak{id}$. Then we can compute the overall effect of these actions by distinguishing two situations:

- $(y' \leftarrow x) = \mathfrak{id}$. Now x provides all the information that y' wants to substract. In that case some information will remain after substracting y' from x and the remaining information $(x \leftarrow y')$ can be added to the information y. We end up with $\langle x', (x \leftarrow y') \bullet y\rangle$.
- $(x \leftarrow y') = \mathfrak{id}$. Now x does not provide everything that y' asks for. In that case there is an additional request for $(y' \leftarrow x)$ from the context. Then we get the overall effect of $\langle (y' \leftarrow x) \bullet x, y\rangle$.

If M is not a linear order, a third case remains in which neither $(x \leftarrow y') = \mathfrak{id}$ nor $(y' \leftarrow x) = \mathfrak{id}$. The definition above simply summarizes all situations.

It is left to the industrious reader to check that $\mathcal{SSC}_{\mathcal{M}}$ is in fact a monoid. Thus we obtain a pairing construction which makes monoids out of L-monoids. It is easy to check that the simple stacking cells are indeed what we get if we take $\mathcal{M} = \mathcal{M}_\omega$ as a starting point.

So we see that the pairing construction generalizes to arbitrary L-monoids. Also the representation of stacking cells as partial functions can be generalized to arbitrary L-monoids. Each SSC $\langle x, y\rangle$ in $\mathcal{SSC}_{\mathcal{M}}$ gives rise to a partial mapping $\varphi_{x,y} : M \to M$ as follows:

$$\begin{aligned} dom(\varphi_{x,y}) &= \{z |\ z \leq x\} = \{z |\ \exists u :\ u \bullet x = z\} \text{ and} \\ \varphi_{x,y}(u \bullet x) &= u \bullet y. \end{aligned}$$

(Here it has to be checked that the u such that $u \bullet x = z$ is unique, which follows immediately from L1.)

We leave it to the reader to verify the following proposition.

Proposition 2.6 *The mapping $\varphi : SSC_{\mathcal{M}} \to \{\varphi_{x,y} |\ x, y \in M\}$ defined by:*

$$\varphi(\langle x, y\rangle) = \varphi_{x,y}$$

induces an isomorphism between $\mathcal{SSC}_{\mathcal{M}}$ and $\langle \{\varphi_{x,y} |\ x, y \in M\}, \circ, \varphi_{\mathfrak{id},\mathfrak{id}}\rangle$

2.5 Lazy vs. tiresome bracketing All this may seem an example of generalizing for the sake of generality, but it turns out that there are some nice ideas about the management of (linguistic) structure that can be captured in this way. As an example we discuss the idea of lazy bracketing, which will amount to the use of $\mathcal{SSC}_{\omega^2}$ instead of $\mathcal{SSC}_{\omega}$. Let's reconsider our example:

The quick brown fox that jumped over the lazy dog wanted the rabbit that ran.

Above we have explained that we have to add some information about the management of constituents if we want to interpret this sentence, and we have used brackets to make the constituent information explicit. The brackets correspond to very explicit operations on constituent structures: each bracket corresponds to pushing or popping exactly one constituent. Thus the use of brackets as indicated above gives the following tiresome picture of the left-to-right interpretation of a sentence: first we decide (or guess) at run-time exactly how many levels we have to push. (In the example this turns out to be three.) We push these levels one by one. Then we go on to interpret the sentence. Finally we pop the remaining levels one by one.

But we can also give a more easy-going picture of how things work.[13] At the beginning of each sentence, indicated by the use of a capital letter, we know that we are at a new starting point. At such a point we do not have to count the exact number of constituents required: we simply introduce "sufficiently many" constituents. Then we go on to interpret the sentence. At the end of the sentence, indicated by the use of a full stop, we know that we have reached an end point. So there is no need to be very careful in popping the remaining constituents one by one: we can simply throw away *all* remaining constituents in one sweep.

To represent this picture of the lazy management of constituents we do not bracket the example as before, but instead use the following bracketing.

[the quick brown fox ((that) jumped (over the lazy dog))) wanted (the rabbit ((that) ran]

Here [indicates the introduction of "sufficiently many" constituents and] stands for throwing away whatever remains. We call [and] *lazy brackets*. In the monoid containing lazy brackets we would expect equalities such as $[) = [, [)) = [, [))) = [$, etc. to hold, indicating that [does indeed introduce sufficiently many constituents. Dually we would like to have $(] =]$, $((] =]$, $(((] =]$, etc. Of course we also want $[] = () = 1$.

It turns out that we can adapt our definitions to model these ideas about lazy constituent management quite easily: we simply set up the whole machinery starting with the L-monoid based on ω^2 instead of ω. Then it turns out that the following way of looking at the brackets works:[14]

- $\triangleright$ $(\rightsquigarrow \langle 0, 1\rangle$
- $\triangleright$ $) \rightsquigarrow \langle 1, 0\rangle$
- $\triangleright$ $[\rightsquigarrow \langle 0, \omega\rangle$
- $\triangleright$ $] \rightsquigarrow \langle \omega, 0\rangle$.

Thus the general construction of stacking cells from L-monoids allows us to look at constituent management in nonstandard ways. Lazy brackets are just a first example of an interesting kind of variation on the operations that we may want to consider

for linguistic applications: starting from other suitable L-monoids may very well generate other interesting views on the management of (linguistic) structure. But at this point there is no time to speculate more in this direction. In what follows we will concentrate on the first kind of stacking cells: stacking cells on ω. Lazy brackets will not pop up again until our treatment of the Dutch reflexive '*zichzelf*' on page 358.

2.6 Levels of stacking cells The stacking cells will be our way of coding up structural information in a monoidal setting. Our overall goal is to use this structural information in the interpretation of (structured) expressions. These expressions not only contain information about their structure, but typically also contain other sorts of information, which we have called (truth) content above. It is important that we are able to locate this content in the correct way in the (structural) context: the content information has to be stored in the constituents of the stacking cell.

As a first step, we show how we can associate with each simple stacking cell a set of *levels* (or constituents) in such a way that we keep control over their location in the simple stacking cell. Then we can store the content items in the stacking cell by linking them to the appropriate level of the simple stacking cell. We will be able to complete this task properly only after §4, when we will have seen the Grothendieck construction, but already at this point we can go some way towards explaining the idea and showing what the problems are.

First we present a mapping $\mathfrak{L}$ that associates to each SSC its set of levels.

Definition 2.7 For each SSC a we define the set $\mathfrak{L}(a)$, the levels of a, as follows:

$$\mathfrak{L}(a) \;=\; a \;\cup\; \{\langle n, \star\rangle |\; n < n_a\} \;\cup\; \{\langle \star, n\rangle |\; n < a(n_a)\} \;\cup\; \{\langle \star, \star\rangle\}$$

(here $\star$ is some fixed new entity).

Among the levels of a we distinguish the following types:

- $\langle 0, \star\rangle,\ \langle 1, \star\rangle,\ \langle 2, \star\rangle, \ldots \langle n_a - 1, \star\rangle$: the *pop* levels (in chronological order)
- $\langle \star, a(n_a) - 1\rangle,\ \langle \star, a(n_a) - 2\rangle, \ldots, \langle \star, 0\rangle$: the *push* levels (in chronological order)
- a: the *stem* levels
- $\langle \star, \star\rangle$: a *garbage* level

We will store the content information that we find in the constituents on these levels. The pop levels correspond to the constituents that our stacking cell will close off. The push levels correspond to the constituents that the stacking cell introduces. Note that the location of a level $\langle n, m\rangle$ is fixed by n and m. For example in the representation of the string:

lazy dog))) wanted (the rabbit ((that

we will find the SSC $\langle 3, 3\rangle$. We will attach the information 'lazy dog' to the level $\langle 0, \star\rangle$, the first pop level. The information 'the rabbit' will end up at level $\langle \star, 2\rangle$, a push level, and the information 'that' will go to $\langle \star, 0\rangle$, another push level.[15]

The stem levels are levels that are structurally neutral. In this example the information 'wanted' will be stored on such a level: the example tells us 'wanted', and we

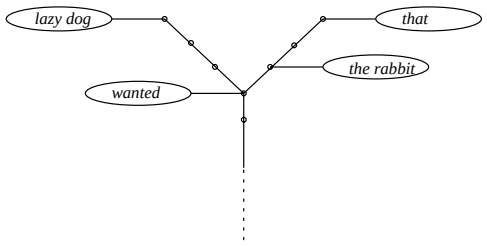


Figure 3: An inhabited stacking cell

know that this information lives in the constituent in which all the pop and push levels are nested. But the string does not give this level any structural status: it may become a push or pop level depending on the context in which the whole string occurs. Therefore we store the information that 'wanted' conveys on a structurally neutral stem level: $\langle 3, 3\rangle$.

In our setup we have provided a rather large number of stem levels. The example above does not make clear why we would ever need more than one such level, but we will see that there are cases where several stem levels are required. Although in practice we will always only use finitely many stem levels, we have chosen to add ω-many such levels, mainly for technical convenience.

So far all levels correspond directly to one of the constituents of an expression. In addition we will allow ourselves to have some extra levels, where we can store information that is not located in any of the constituents, but still belongs to the stacking cell. We will call these extra levels: *garbage levels*. Here we have just one such level, $\langle\star, \star\rangle$, but later finite sets of garbage levels will occur. At this point it is hard to be precise about the exact use of garbage levels. The real reason for introducing them is that it will considerably smooth the definitions later on: when we want to merge two simple stacking cells a and b in which we have stored information, some of the push levels of a may be popped by b. This means that these levels will not show up in the merger $a \bullet b$. But if we are not careful this will also mean that all the information that we stored on those levels is lost. It will be easy to prevent such disasters by (temporarily) storing the information of these levels in a garbage level.

In fact the use of garbage levels is just one example of a general issue in the definition of the merger of stacking cells once they are enriched with additional information. As we pointed out above, we want to add information content to the context that a simple stacking cell provides by attaching this information content to the levels of a simple stacking cell. So, as a first step, we will have to be able to work with tuples $\langle a, X_a\rangle$ where $\mathfrak{L}(a)$ gives us the constituent levels of a and X_a stands for the garbage levels of a. We want to define the monoidal operation $\bullet$ in this situation. This means that, apart from producing the right SSC $a \bullet b$, we also have to make sure that the information that we have stored on some level of a or b ends up on the right level of $a \bullet b$. In slogan:

> we have to keep track of *how levels travel*.

To do this correctly we will borrow some techniques from category theory, which will be presented in §3 and §4.

3 Categories for monoidal updating For the development of our tool-kit for building meanings, we need categories. How nice it would have been if monoids were sufficient. The reason they are not is as follows. Consider simple stacking cells or SSCs. SSCs interact with reassuring monoidal simplicity. But how can we use SSCs to describe more complicated objects? We need some way to talk about the individual "levels" of an SSC, and *we need some way to describe what happens to the levels when two SSCs interact*! In such interactions, levels merge with other levels, are sent into the garbage limbo, etc. To describe the *flow* of the levels, the category-theoretical machinery is tailor-made.

3.1 Basics This section introduces the basic concepts of category theory. The reader is referred to MacLane [10], Manes and Arbib [11], and Barr and Wells [1] for more information.

A category $\mathfrak{A}$ is a structure $\langle Ob, Ar, id, dom, cod, \circ \rangle$, where:

- *Ob* is a nonempty class, the class of objects
- *Ar* is a class, the class of arrows or homomorphisms
- *id* is a function from *Ob* to *Ar*. We will write id_a for $id(a)$
- *dom* and *cod* are functions from *Ar* to *Ob*
- $dom(id_a) = cod(id_a) = a$
- $\circ$ is a partial function from $Ar \times Ar$ to *Ar*
- $f \circ g$ is defined iff $cod(f) = dom(g)$
- If $f \circ g$ is defined, then $dom(f \circ g) = dom(f)$ and $cod(f \circ g) = cod(g)$
- $id_{dom(f)} \circ f = f \circ id_{cod(f)} = f$
- If $(f \circ g) \circ h$ is defined, then $(f \circ g) \circ h = f \circ (g \circ h)$

In what follows, identity between partial terms means: either both sides are defined and equal, or both are undefined. Thus we have, quite generally: $(f \circ g) \circ h = f \circ (g \circ h)$. We go against the mainstream tradition in category theory by reading $\circ$ in the order of the depicted arrows. Thus our $f \circ g$ "means" *first f, then g*. The reason for this deviation is that our morphisms often represent "updates." For representing updates, it is most natural to read composition in the order of application. (See below for more conventions in a similar spirit.) We will call the set of morphisms between *a* and *b*, $Hom(a, b)$, or, if we want to emphasize the dependence on the category, $Hom_{\mathfrak{A}}(a, b)$. A morphism $f : a \to b$ is an *isomorphism* if there is a $g : b \to a$, such that $f \circ g = id_a$ and $g \circ f = id_b$.

A *Functor* Θ between $\mathfrak{A}$ and $\mathfrak{B}$ is a morphism of categories between $\mathfrak{A}$ and $\mathfrak{B}$. I.e., Θ a function mapping $Ob_{\mathfrak{A}}$ to $Ob_{\mathfrak{B}}$, and $Ar_{\mathfrak{A}}$ to $Ar_{\mathfrak{B}}$, which preserves all categorical structure. So, for example, $\Theta(id_{\mathfrak{A},a}) = id_{\mathfrak{B},\Theta(a)}$ and $\Theta(f \circ_{\mathfrak{A}} g) = \Theta(f) \circ_{\mathfrak{B}} \Theta(g)$.

Example 3.1 An important example of a category will be the category $\mathfrak{Set}$, where we take:

- *Ob* is the class of all sets[16]
- *Ar* is the class of all functions from sets to sets
- id_X is the identity function on X
- $dom(f)$ is the domain of f and $cod(f)$ is the range of f.

- $\triangleright$ $\circ$ is function-composition, read in order of application

In this category the elements of the sets are treated as *featureless objects*. Their incidental features are divided out by the isomorphisms present in the category and isomorphism in the category is the intended notion of object identity. So just the "sizes" of the sets are counted relevant.

Example 3.2 Another important example of a category is given by a partial (weak) preorder or *ppo*, $\langle D, \leq \rangle$. Here the $Ob = D$ and the arrows are the "inclusions," $inc_{a,b}$, witnessing that $a \leq b$. A prominent example of a category based on a *ppo* is the category $\mathfrak{Set}_{sub}$, where the *ppo* is formed by sets with the subsetordering. Note that $\mathfrak{Set}_{sub}$ is a subcategory of $\mathfrak{Set}$. The notion of identity in $\mathfrak{Set}_{sub}$ is, however, completely different. Isomorphism in this category is ordinary identity of sets. So every incidental feature of an element counts. Another example is the category $\mathfrak{Nat}$ of the natural numbers $\{0, 1, 2, \ldots\}$, with their natural ordering.

At this point we introduce an important convention. We want to think about updating and interpreting language fragments. Composition will reflect concatenation at the level of surface syntax. Thus, as already mentioned above, we read composition in order of application. For the same reason we should use postfix notation to describe function application. However, as so often, it turns out that *jede Konsequenz zum Teufel führt*. The postfix notations indiscriminately applied look peculiar, certainly in case of binary functions. Moreover, not all functions that appear need to be considered as update functions. So a hybrid notation seems best. We will write $f(x)$, when using prefix notation, and $x[f]$, when using postfix notation. So for example if Θ is a functor from $\mathfrak{A}$ to $\mathfrak{Set}$, if $f : a \to b$ is a morphism in $\mathfrak{A}$, and if $x \in \Theta(a)$, then: $\Theta(f)(x) = x[\Theta(f)] = x[f[\Theta]]$. In a suitable context, functions from sets to sets could represent updates, whereas the functor Θ does not. So, here we would prefer the notation: $x[\Theta(f)]$. Another convention that we will use is: $\langle x, \langle y, z\rangle\rangle = \langle x, y, z\rangle$.

The objects of our categories are supposed to be informational items. The arrows fulfill two important roles. The first is that they represent ways in which one piece of information is part of another. The second is that the isomorphisms that are present fix what objects and arrows we will count as *the same*.[17] We also need an operation *merge* or $\bullet$ that enables us to glue some pieces of information together. To describe the operation or $\bullet$, we again need some extra morphisms. To motivate our choices, we first look at an example.

Example 3.3 We consider what is involved in adding the monoidal operation *disjoint union* to $\mathfrak{Set}$. In one sense this example is the *ur*-example of a monoidal operation on a category. In another sense it is somewhat misleading: disjoint union is a bifunctor. Moreover it is the direct sum or co-product of the category we are considering. These features will not be incorporated in the general case. We start by fixing a representation of disjoint union.

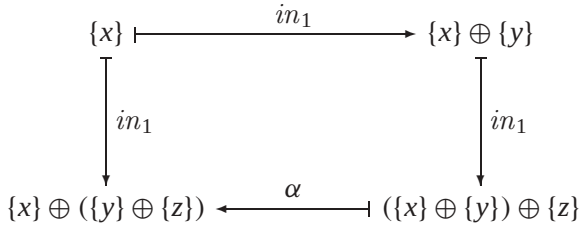
$$X \oplus Y = (\{0\} \times X) \cup (\{1\} \times Y).$$

The elements of X will have descendants in $X \oplus Y$. This descendancy relation can be described by a morphism, say in_1. For $x \in X$, we take: $x[in_1(X, Y)] := \langle 0, x\rangle$. Similarly $y[in_2(X, Y)] := \langle 1, y\rangle$. The inclusion morphisms keep track of how levels

travel when objects are fused. Disjoint union does not give us a monoid in the strict sense. We have, for example:

$$\begin{aligned}(\{x\} \oplus \{y\}) \oplus \{z\} &= \{\langle 0,0,x\rangle, \langle 0,1,y\rangle, \langle 1,z\rangle\} \\ \{x\} \oplus (\{y\} \oplus \{z\}) &= \{\langle 0,x\rangle, \langle 1,0,y\rangle, \langle 1,1,z\rangle\}.\end{aligned}$$

However, we wish to view the coding machinery we introduced to keep elements out of each other's way in taking disjoint unions as "inessential." The elements of $(X \oplus Y) \oplus Z$ are the same as those of $X \oplus (Y \oplus Z)$ *modulo* some coding. To make this idea explicit we introduce a standard isomorphism $\alpha(X, Y, Z)$ between $(X \oplus Y) \oplus Z$ and $X \oplus (Y \oplus Z)$. In our example we would have: $\langle 0,0,x\rangle[\alpha(\{x\},\{y\},\{z\})] = \langle 0,x\rangle$. It is not sufficient that there be some isomorphism: we want the correct isomorphism. For one thing, α and the in-functions will have to cooperate in appropriate ways. For example, we expect the following diagram to commute.



An easy check shows that the diagram commutes. Our monoid has a unit $\mathfrak{id}$. This is of course the empty set. We see that in_1 is an isomorphism between X and $X \oplus \varnothing$ and that in_2 is an isomorphism between Y and $\varnothing \oplus Y$. The richer structure built on $\mathfrak{Set}$, that we have described is an *m-category* according to the definition given below. We will call the enriched $\mathfrak{Set}$ again: $\mathfrak{Set}$.

After this motivating example we turn to the main definition. Our framework is rather similar to the usual notion of monoidal category, the main differences being the fact that the monoidal operation is not a functor and the presence of the *in*-functions. A structure $\mathfrak{A} = \langle Ob, Ar, id, dom, cod, \circ, \bullet, \mathfrak{id}, in_1, in_2, \alpha\rangle$, is an *m-category* if (i) $\langle Ob, Ar, id, dom, cod, \circ\rangle$ is a category and (ii) "$\langle Ob, \bullet, \mathfrak{id}, in_1, in_2, \alpha\rangle$ describes a monoid relative to the category." Our phrase (ii) means that we have a monoid only *modulo* the isomorphisms of our category. We spell (ii) out in some detail:

- $\bullet : Ob \times Ob \to Ob$
- $\mathfrak{id} \in Ob$
- $in_i : Ob \times Ob \to Ar$, where $in_i(a_1, a_2) : a_i \to a_1 \bullet a_2$. The in_i tell us in which way the a_i are embedded in $a_1 \bullet a_2$ by the operation $\bullet$
- $\alpha : Ob \times Ob \times Ob \to Ar$, where $\alpha(a,b,c) : (a \bullet b) \bullet c \to a \bullet (b \bullet c)$. Here:
  1. $in_1(a,b) \circ in_1(a \bullet b, c) \circ \alpha(a,b,c) = in_1(a, b \bullet c)$
  2. $in_2(a,b) \circ in_1(a \bullet b, c) \circ \alpha(a,b,c) = in_1(a,b) \circ in_2(a, b \bullet c)$
  3. $in_2(a \bullet b, c) \circ \alpha(a,b,c) = in_2(b,c) \circ in_2(a, b \bullet c)$
- $in_1(a, \mathfrak{id})$ is an isomorphism between a and $a \bullet \mathfrak{id}$. Similarly, $in_2(\mathfrak{id}, a)$ is an isomorphism between a and $\mathfrak{id} \bullet a$. Finally $in_1(\mathfrak{id}, \mathfrak{id}) = in_2(\mathfrak{id}, \mathfrak{id})$.[18]

How nice it would have been, if these were all the conditions we needed to impose. However, to guarantee that everything works smoothly we need some conditions of a more technical nature. For the record, we give them here.

- ▷ We need everything to behave well with respect to isomorphisms. If, for $i = 1, 2$, b_i is isomorphic to b'_i, then $b_1 \bullet b_2$ is isomorphic to $b'_1 \bullet b'_2$. The full formulation is as follows. Suppose $\beta_i : b_i \to b'_i$ is an isomorphism ($i = 1, 2$). Then there is a unique isomorphism $\beta_1 \bullet \beta_2 : b_1 \bullet b_2 \to b'_1 \bullet b'_2$ such that:

$$in_i(b_1, b_2) \circ (\beta_1 \bullet \beta_2) = \beta_i \circ in_i(b'_1, b'_2).$$

- ▷ The conditions on α are not sufficient to ensure that the correct isomorphisms are generated after repeated applications of associativity. To guarantee correct behavior ("coherence"), we have to add an extra condition. We ask that the in_i are jointly surjective, i.e., if, for $i = 1, 2$, $in_i \circ f = in_i \circ g$, then $f = g$.[19]

A functor Θ between m-categories is an *m-functor* if it preserves the additional structure. E.g. $\Theta(in_1(a, b)) = in_1(\Theta(a), \Theta(b))$.

Example 3.4

1. We add the monoidal operation *plus* to $\mathfrak{Nat}$. The further details are fixed by this choice.
2. Consider an *upper semilattice* $\mathfrak{U}$, i.e., a structure $\langle D, \leq, \vee, \perp\rangle$. Here D is a nonempty set and $\leq$ is a partial order, which is closed under taking suprema of finite sets of elements. $\vee$ is the operation of taking the supremum of two elements, and $\perp$ is the bottom. $\mathfrak{U}$ can be viewed as an m-category, by viewing $\langle D, \leq\rangle$ as a category as in Example 3.2. We take $\vee$ as the monoidal operation and $\perp$ as $\mathfrak{id}$. An important special case of this example is $\mathfrak{Set}_{sub}$, with union and empty set.

Par abus de langage we will call the resulting m-categories again $\mathfrak{Nat}$, $\mathfrak{U}$ and $\mathfrak{Set}_{sub}$.

We are now ready to introduce the last ingredient. Our semantics is intended to be *file change semantics* in the sense of Heim [4]. The objects of our categories are dynamic *whatshallwecallthems*. Using the category we can describe their interactions. We will need some way to talk about the files and the information stored there. The solution is to extend our categories with a functor $\mathfrak{R}$ from the category to $\mathfrak{Set}$. For each object a, $\mathfrak{R}(a)$ wil give the set of files "contained in" a. Thus in our example above we could take $\mathfrak{R}$ to be the identity functor in Example 3.1, the standard inclusion of $\mathfrak{Set}_{sub}$ in $\mathfrak{Set}$ in the special case of Example 3.3, and we can take $\mathfrak{R}(n) := \{m{\in}\omega \mid m < n\}$ in Example 3.2. We do not require $\mathfrak{R}$ to be an m-functor! In fact, since we want to view the elements of the category under consideration as coordinating possible unifications of referents, it is, in general, essential that $\mathfrak{R}$ is *not* an m-functor. The choice of $\mathfrak{Set}$ as category of sets of files reflects that we view a file as a featureless object, but for its connection via $\mathfrak{R}$ with the dynamic machinery.

4 The Grothendieck Construction The Grothendieck Construction can be viewed as a definitional format. It is a way of constructing objects which carries with it the guarantee that objects so constructed have such-and-such properties. In a sense

one could say that the Construction constitutes a *functional role* definition of what it is to be a context's content and what it is to be a content's context. The most salient ingredient here is that contexts transform independently of the content, but that the transformation of contents is guided by the context.[20] A good discussion of the Grothendieck Construction can be found in [1] and in Jacobs [7].

Consider an m-category $\mathfrak{A}$ and a functor Θ from $\mathfrak{A}$ to the category m-$\mathfrak{Cat}$ of m-categories. The Grothendieck construction allows us to make a new category of pairs, $\langle a, t\rangle$, where t is an object of $\Theta(a)$. The intuition is this. $\mathfrak{A}$ is a category of *contexts*. $\Theta(a)$ is the category of *contents* above a. A pair $\langle a, t\rangle$ will be a content at a context. A morphism f from a to b, will be viewed as an *embedding of contexts*. When we take the "a-object," t, under our arm, when traveling via f from a to b, t will be "transformed" into a b-object $t' := (\Theta(f))(t)$. So $(\Theta(f))(t)$ is the canonical image of t via f.

Before giving the definition let us give a kind of *ur*-example, that well conveys the flavor of what is going on.

Example 4.1 Consider a model of predicate logic with domain D. Usually the meanings of formulas are defined as sets of assignments from the set of variables *VAR* to D. However we could also wish to work with local assignments acting only on the (free) variables that are "present." Meanings now will be pairs of a finite set of variables V and a set F of assignments from V to D. Thus, e.g., the meaning of $P(v_1, \ldots, v_n)$ would be $\langle V, \{f \in D^V \mid \langle f(v_1), \ldots, f(v_n)\rangle \in I(P)\}\rangle$ where $V = \{v_1, \ldots, v_n\}$ and I is the interpretation function associated with the model. The first component, V, of such a pair is viewed as the context, the second component, F, as the content. The m-category of contexts here has as objects finite sets, V, of variables, and as arrows the inclusion functions $inc_{V,V'}$, signaling that $V \subseteq V'$. The monoidal operation is union. Above each context V we have a category $\Theta(V)$ of contents. The contents above V are the sets of assignments from V to D. The arrows of this category are the opposites of the inclusion functions, say, $cni_{F,F'}$, signaling that $F \supseteq F'$. The monoidal operation is intersection.

Suppose $V \subseteq V'$ and F is a set of assignments on V. How is F going to appear if we transport it to V'? Well, we want F to decribe the same constraint at the new context. In other words, we want F's "successor" to be the least informative object in the new context, which is constrained in the same way with respect to the old variables. Thus we take:

$$F[\Theta(inc_{V,V'})] := \langle V', \{f \in D^{V'} \mid f \restriction V' \in F\}\rangle.$$

How are we going to define the meaning of $A \wedge B$, say $\|A \wedge B\|$? Suppose $\|A\|$ is $\langle V, F\rangle$ and $\|B\|$ is $\langle W, G\rangle$. If the contexts V and W were the same this would be simple: $\|A \wedge B\| = \langle V, F \cap G\rangle$. If V and W are unequal, however, F and G live in different worlds and cannot be intersected in a sensible way. What we do is take them under the arm and take them to the nearest world where both can breathe, the world above context $V \cup W$. In this world we *can* intersect. So our new conjunction will be as follows:

$$\|A\| \wedge \|B\| \;:=\; \langle V \cup W, F[\Theta(inc_{V,V\cup W}] \cap G[\Theta(inc_{W,V\cup W})]\rangle.$$

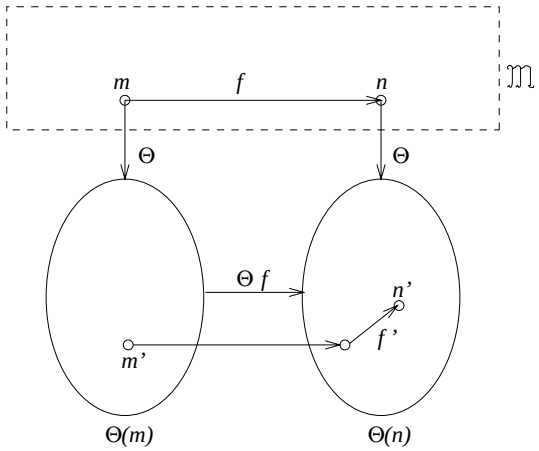


Figure 4: The Grothendieck Construction

Definition 4.2 Let an m-category $\mathfrak{A}$ and a functor $\Theta : \mathfrak{A} \to$ m-$\mathfrak{Cat}$ be given. Then we define a new m-category $\mathfrak{B} := \sum_{a \in \mathfrak{A}} \Theta(a)$ as follows:

- ▷ the objects of $\mathfrak{B}$ are the pairs $\langle a, t \rangle$ where t is an object of $\Theta(a)$
- ▷ the morphisms between $\langle a, t \rangle$ and $\langle b, s \rangle$ are the pairs $\langle f, u \rangle$ such that $f \in Ar_{\mathfrak{A}}$ and $f : a \to b$ and $u \in Ar_{\Theta(b)}$ and $u : t[\Theta(f)] \to s$
- ▷ composition of arrows is defined as follows: if $\langle f, u \rangle : \langle a, t \rangle \to \langle b, s \rangle$ and $\langle g, v \rangle : \langle b, s \rangle \to \langle c, r \rangle$, then $\langle f, u \rangle \circ \langle g, v \rangle : \langle a, t \rangle \to \langle c, r \rangle$ is the pair $\langle f \circ g, u[\Theta(g)] \circ v \rangle$.
- ▷ $\mathfrak{id}_{\mathfrak{B}} = \langle \mathfrak{id}_{\mathfrak{A}}, \mathfrak{id}_{\Theta(\mathfrak{id}_{\mathfrak{A}})} \rangle$

We introduce the new monoidal operator and the new *in*-functions. We want to define $\langle a, t \rangle \bullet \langle b, s \rangle$. On the first components, we take the obvious operations, sending a and b to $a \bullet b$. In going from a via $in_1(a, b)$, to $a \bullet b$, the object t is transformed to $t' := t[\Theta(in_1(a, b))]$. Similarly s is transformed to $s' := s[\Theta(in_2(a, b))]$. Finally—on the second component—we take $t' \bullet s'$. Thus:

- ▷ $\langle a, t \rangle \bullet \langle b, s \rangle = \langle a \bullet b, t' \bullet s' \rangle$
- ▷ $in_i(\langle a, t \rangle, \langle b, s \rangle) = \langle in_i(a, b), in_i(t', s') \rangle$.

The new α is defined in a similar way.

It requires quite a bit of tedious work to check in detail that the Grothendieck construction really preserves m-categories.

Example 4.3 Consider any two m-categories $\mathfrak{A}$ and $\mathfrak{B}$. We confuse $\mathfrak{B}$ with the following functor from $\mathfrak{A}$ to m-$\mathfrak{Cat}$: $\mathfrak{B}(a) := \mathfrak{B}$ and $\mathfrak{B}(f) := ID_{\mathfrak{B}}$, where $ID_{\mathfrak{B}}$ is the identity functor on $\mathfrak{B}$. Then $\sum_{a \in \mathfrak{A}} \mathfrak{B}(a)$ is (isomorphic to) $\mathfrak{A} \times \mathfrak{B}$.

A somewhat larger example is worked out in the Appendix. An important point is the fact that the m-category $\mathfrak{A}$ reoccurs as a sub-m-category of $\sum_{a \in \mathfrak{A}} \Theta(a)$. Consider the following mapping Φ:

- ▷ $\Phi(a) := \langle a, \mathfrak{id}_{\Theta(a)} \rangle$
- ▷ $\Phi(f) := \langle f, id_{\mathfrak{id}_{\Theta(b)}} \rangle$.

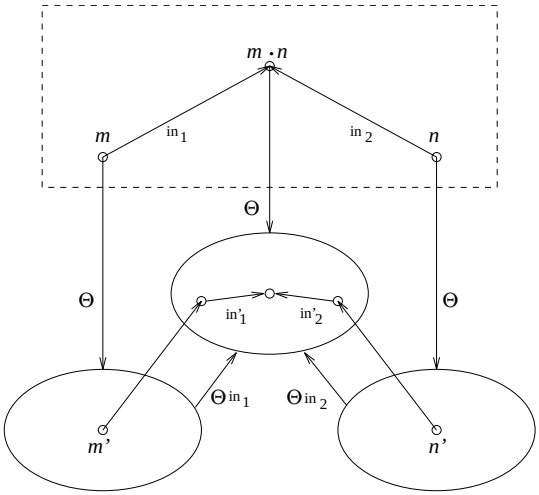


Figure 5: m-structure under the Grothendieck Construction

It is not difficult to see that Φ is an injective m-functor. Thus we are licensed to identify objects and morphisms of $\mathfrak{A}$, with their images under Φ.

We give three particularly useful specializations of the Grothendieck Construction.

4.1 Adding contents to contexts Let an m-category $\mathfrak{A}$, a functor $\mathfrak{R}$ from $\mathfrak{A}$ to $\mathfrak{Set}$, and a nonempty domain D be given. Remember that $\mathfrak{R}$ need not be an m-functor. We generalize the construction of the meanings of Example 4.1. Define the functor $\mathfrak{Ass}$ as follows.

- $\mathfrak{Ass}(a)$ is the following m-category.
  - The objects are sets of functions from $\mathfrak{R}(a)$ to D.
  - The arrows are given by the partial ordering $\supseteq$.
  - The monoidal operation is intersection of sets.
  - The rest of our category is fixed by the above.
- Let $f : a \to b$ be an $\mathfrak{A}$-morphism. We define the functor $\mathfrak{Ass}(f)$ as follows.
  - $G[\mathfrak{Ass}(f)] := \{h \in D^{\mathfrak{R}(b)} \mid \exists g \in G \forall r \in \mathfrak{R}(a)\ h(r[\mathfrak{R}(f)]) = g(r)\}$.
  - The application of $\mathfrak{R}(f)$ on the morphisms is fixed by the preceding item.

It is easy to verify that $\mathfrak{Ass}$ is a functor. We put:

- $\mathfrak{Cont}(\mathfrak{A}, \mathfrak{R}, D) := \sum_{a \in \mathfrak{A}} \mathfrak{Ass}(a)$.

We define a new $\mathfrak{R}$ on the new category by: $\mathfrak{R}(\langle u, v\rangle) := \mathfrak{R}(u)$.[21] It is not difficult to see that our Example 4.1 can be obtained by taking $\mathfrak{A}$ the category of finite sets of variables, with the inclusion functions as morphisms and union as monoidal operator. The functor $\mathfrak{R}$ of this m-category is the standard inclusion in $\mathfrak{Set}$. Note that via the standard embedding of the contexts into the context/content pairs we can identify a finite set of variables V with $\langle V, D^V\rangle$.

One could think of all kinds of variants of our construction. E.g., instead of working with sets of assignments, we could work with relational databases over the given set of referents.

4.2 Synchronic identification It will happen often that we want to say of two inhabitants of different parts of the (linguistic) structure that they are really the same. A familiar example is formed by re-entrancies in feature structures, where we want to express that two distinct expressions *share* some feature. We have not included any feature information in our linguistic examples, but already in our naive example a similar phenomenon pops up: in the interpretation of the relative 'that'. We want to say that 'that' *shares* its denotation with an expression that lives in some other constituent. Consider:

wanted (the rabbit ((that) ran)))

Here 'that' points to the same object as 'the rabbit'. We keep score of information concerning such identities by working with an equivalence relation on all the objects that occur somewhere in the relevant stacking cell.

So we will have, in the semantics, as one of the informational items an equivalence relation on a set of objects. The Grothendieck Construction can be used to describe the dividing out of equivalence relations. Let R be any binary relation on a set X. We write R^* for the transitive, reflexive, symmetric closure of R (in X). Thus R^* is the *least*, or *finest*, equivalence relation containing R. Let $\mathfrak{A}$ be an m-category, and let $\mathfrak{R}$ be a functor from $\mathfrak{A}$ to $\mathfrak{Set}$. We describe the functor $\mathfrak{E}$.

- $\mathfrak{E}(a)$ is the following category.
  - The objects are the equivalence relations on $\mathfrak{R}(a)$.
  - The morphisms are given by the subset ordering on equivalence relations considered as sets of pairs. So, we have arrows from finer to coarser equivalence relations.
  - $E \bullet E' := (E \cup E')^*$.
  - The other data on the category are fixed by the preceding items.
- Let $f : a \to b$ be an $\mathfrak{A}$-morphism. We put: $E[\mathfrak{E}(f)] := (E[\mathfrak{R}(f)])^*$, where $E[\mathfrak{R}(f)] := \{\langle r[\mathfrak{R}(f)], r'[\mathfrak{R}(f)]\rangle \mid \langle r, r'\rangle \in E\}$

We may check that $\mathfrak{E}$ is, indeed, an m-functor. Take: $\mathfrak{Eq}(\mathfrak{A}, \mathfrak{R}) := \sum_{a\in\mathfrak{A}} \mathfrak{E}(a)$. We may chose the new $\mathfrak{R}$ as follows.

- $\mathfrak{R}(\langle a, E\rangle) := \mathfrak{R}(a)/_E$,
- Let $\langle f,\ f'\rangle{:}\langle a, E\rangle \to \langle a', E'\rangle$, then: $(r/_E)[\mathfrak{R}(\langle f,\ f'\rangle)] := (r[\mathfrak{R}(f)])/_{E'}$.

It is easy to see that this definition is correct.

Example 4.4 Let $\mathfrak{A}$ and $\mathfrak{B}$ be m-categories and let $\mathfrak{R}_\mathfrak{A}$ and $\mathfrak{R}_\mathfrak{B}$ be the corresponding functors. The Cartesian product of $\mathfrak{A}$ and $\mathfrak{B}$ is defined in the obvious way. E.g., $\langle a, b\rangle \bullet \langle a', b'\rangle = \langle a \bullet a', b \bullet b'\rangle$. Take: $\mathfrak{R}_{\mathfrak{A}\times\mathfrak{B}}(\langle a, b\rangle) := \mathfrak{R}_\mathfrak{A}(a) \oplus \mathfrak{R}_\mathfrak{B}(b)$. Here $\oplus$ stands for disjoint union. The $\mathfrak{R}_{\mathfrak{A}\times\mathfrak{B}}(\langle f, g\rangle)$ are defined in the obvious way. Now the Cartesian product can be viewed as two forms of dynamic machinery $\mathfrak{A}$ and $\mathfrak{B}$ running in parallel, without any connection. Now we may define $\mathfrak{A}\|\mathfrak{B} := \mathfrak{Eq}(\mathfrak{A} \times$

$\mathfrak{B}, \mathfrak{R}_{\mathfrak{A}\times\mathfrak{B}})$. The new $\mathfrak{R}$ is defined in the obvious way. The result of our construction enables two different machineries to contribute to the identification of the same files. We will use this construction to link the global anaphoric way of identifying referents and the local grammatical way.

4.3 Storing dynamic objects on levels We turn to our final subconstruction. The idea here is to store the elements of an m-category *in* the files of another m-category. Let $\mathfrak{A}$ and $\mathfrak{B}$ be two m-categories, and let $\mathfrak{R}$ and $\mathfrak{S}$ be the corresponding functors. Suppose that for all $\mathfrak{A}$-morphisms f, $\mathfrak{R}(f)$ is *injective*. We specify the functor $\mathfrak{Q}$. Let $a \in Ob_{\mathfrak{A}}$.

- $\mathfrak{Q}(a)$ is the following category.
  - The objects are functions from $\mathfrak{R}(a)$ to the objects of $\mathfrak{B}$. We will use $\sigma, \tau, \ldots$ for the objects.
  - A morphism $\varphi : \sigma \to \tau$ is a function from $\mathfrak{R}(a)$ to the morphisms of $\mathfrak{B}$, such that: $\varphi(r) : \sigma(r) \to \tau(r)$.
  - $\sigma \bullet \tau(r) := \sigma(r) \bullet \tau(r)$.
  - The further definitions are similar.
- Suppose $f : a \to a'$. We define $\mathfrak{Q}(f) : \mathfrak{Q}(a) \to \mathfrak{Q}(a')$.
  - $(\sigma[\mathfrak{Q}(f)])(r) := \begin{cases} \sigma(r[(\mathfrak{R}(f))^{-1}]) & \text{if } r \text{ is in the range of } \mathfrak{R}(f) \\ \mathfrak{id}_{\mathfrak{B}} & \text{otherwise} \end{cases}$
  - $(\varphi[\mathfrak{Q}(f)])(r) := \begin{cases} \varphi(r[(\mathfrak{R}(f))^{-1}]) & \text{if } r \text{ is in the range of } \mathfrak{R}(f) \\ id_{\mathfrak{id}_{\mathfrak{B}}} & \text{otherwise} \end{cases}$

  Note that in these definitions the injectivity of $\mathfrak{R}$ is essential.

We define $\mathfrak{Store}(\mathfrak{A}, \mathfrak{R}, \mathfrak{B}, \mathfrak{S})$, or, briefly, $\mathfrak{B}^{*\mathfrak{A}}$, as $\sum_{a\in\mathfrak{A}} \mathfrak{Q}(a)$. In the last notation we take $\mathfrak{R}$ and $\mathfrak{S}$ to be given with the m-categories.[22] The construction gives us pairs $\langle a, \sigma\rangle$, where σ stores an object of $\mathfrak{B}$ on each referent in $\mathfrak{R}(a)$. We define the new functor, say $\mathfrak{T}$, to $\mathfrak{Set}$, as follows.

- $\mathfrak{T}(\langle a, \sigma\rangle) := \{\langle r, s\rangle \mid r \in \mathfrak{R}(a) \textit{ and } s \in \mathfrak{S}(\sigma(r))\}$
- Suppose $\langle f, \varphi\rangle : \langle a, \sigma\rangle \to \langle a', \sigma'\rangle$. Let's put $r' := r[\mathfrak{R}(f)]$. Then we may define: $\langle r, s\rangle[\mathfrak{T}(\langle f, \varphi\rangle)] := \langle r', s[\mathfrak{S}(\varphi(r'))]\rangle$

Thus $\mathfrak{T}(\langle a, \sigma\rangle)$ gives the disjoint union of the $\mathfrak{S}(\sigma(r))$ for $r \in \mathfrak{R}(a)$. A variant of this construction is the finitized version, $\mathfrak{Store}_{fin}(\mathfrak{A}, \mathfrak{R}, \mathfrak{B}, \mathfrak{S})$, where we restrict the σ to functions that are *almost everywhere*, i.e., for all but finitely many arguments, equal to $\mathfrak{id}_{\mathfrak{B}}$ and the φ to functions that are almost everywhere equal to $id_{\mathfrak{id}_{\mathfrak{B}}}$.

Example 4.5 We give a useful application of our construction. The referents or files in our applications sometimes only have an "internal" or "virtual" function. They function as indicators of places in a structure or whatever, but they are not used for further storage. It is often pleasant and even necessary to make such files invisible in the final stage. The following construction, $\mathfrak{Pres}$, does just this. We will use the construction in the next section.

Consider the m-category $\mathfrak{True}$. This is the m-category based on the upper semi-lattice $\langle \wp\{+\}, \subseteq, \cup, \varnothing\rangle$, where $+$ is an arbitrary object. $\mathfrak{R}_{\mathfrak{True}}$ is the standard inclusion of our m-category in $\mathfrak{Set}$. Consider an m-category $\mathfrak{A}$, with associated functor $\mathfrak{R}$. Define: $\mathfrak{Pres}(\mathfrak{A}) := \mathfrak{True}^{*\mathfrak{A}}$. The objects of this m-category are pairs $\langle a, \sigma\rangle$. Here σ is a function from $\mathfrak{R}(a)$ to $\{\varnothing, \{+\}\}$. Thus σ functions as a characteristic function on $\mathfrak{R}(a)$, representing a set $X_\sigma \subseteq \mathfrak{R}(a)$. Here X_σ is the set of elements of $\mathfrak{R}(a)$ that are "present." We have: $\mathfrak{T}(\langle a, \sigma\rangle) = \{\langle x, +\rangle \mid x \in X_\sigma\}$. We will also write $\mathfrak{A}_+$ for $\mathfrak{Pres}(\mathfrak{A})$. If we want the finitized version of our construction we add the subscript *fin* in the obvious way.

Example 4.6 Let $\mathfrak{True}$ be as in Example 4.5. Consider $\mathfrak{A}^{*\mathfrak{True}}$. The objects are of the form $\langle \varnothing, \varnothing\rangle$ or $\langle \{+\}, \sigma\rangle$. Since, in the second case, $dom(\sigma) = \{+\}$, we may identify σ, with $\sigma(+)$. Hence, the objects can be viewed as pairs $\langle +, a\rangle$. Thus, the result of our construction is *adding a new unit to* $\mathfrak{A}$.

Example 4.7 Let A be a finite set of items. We define multisets of the items in A as follows. We associate an m-category $\mathfrak{A}$ with A. $\mathfrak{A}$ is the category consisting of A as its single object, with as unique morphism the identity witnessing the standard inclusion of A in itself. For our monoidal operation we cannot but choose "union." The functor $\mathfrak{R}$ is the obvious inclusion of A in $\mathfrak{Set}$. We take as a category of multisets of items from A: $\mathfrak{Store}(\mathfrak{A}, \mathfrak{Set}_{fin})$. The objects of our new category are in essence functions f from A to finite sets. (We may omit the context, since it is fixed.) Moreover, e.g., $f \bullet g(a)$ is the disjoint union of $f(a)$ and $g(a)$. The new functor, say $\mathfrak{T}$, sends f to $\{\langle a, x\rangle \mid a \in A,\ x \in f(a)\}$.

In Table 1 we repeat the most important constructions introduced in this section.

5 Category of stacking cells

In this section we look at stacking cells once again. But this time we look at them in a (m-)categorical setting, adding the appropriate notions of morphism and embedding.

Recall that it is necessary and handy to enrich simple stacking cells with garbage levels. Adding the garbage levels is one of the things we have to do in order to keep track of *how levels travel*. So, instead of working with SSCs we will have to work with pairs $\langle a, X\rangle$ consisting of an SSC a and an appropriate set of garbage levels X. The resulting objects will then have as levels the levels of the SSC, as we introduced them above, as well as the garbage levels that we have added to them.

But before we define the m-category of stacking cells (with garbage), we first introduce the m-category of simple stacking cells, without garbage.

5.1 The category of simple stacking cells

In what follows it will be convenient to use the following notation:

> we write $a \leq b$ for $a \subseteq b$ (as partial functions $\omega \multimap \omega$) and $a^\wedge$ for the converse of a (as a partial function). $\mathfrak{id}$ is the unit of $\mathcal{SSC}$.

We collect the following useful facts (notation as on page 328).

Fact 5.1

- $a \leq b$ *iff* $n_a - n_b = a(n_a) - b(n_b) \geq 0$

$\mathfrak{Cont}(\mathfrak{A}, \mathfrak{R}, D)$	This operation was introduced in §4.1. It puts sets of assignments from $\mathfrak{R}(a)$ to D above each context a.
$\mathfrak{Eq}(\mathfrak{A}, \mathfrak{R})$	This operation was introduced in §4.2. It adds equivalence relations on $\mathfrak{R}(a)$ above each context a. The new referents assigned to a are equivalence classes of the old ones.
$\mathfrak{A} \| \mathfrak{B}$	This operation was introduced in §4.2, Example 4.4. The new objects are pairs $\langle a, b \rangle$, where a, b are from $\mathfrak{A}$, respectively $\mathfrak{B}$, together with an equivalence relation E on the disjoint union of $\mathfrak{R}(a)$ and $\mathfrak{R}(b)$. The new referents are the equivalence classes of E.
$\mathfrak{Store}(\mathfrak{A}, \mathfrak{R}, \mathfrak{B}, \mathfrak{S})$	This operation was introduced in §4.3. It stores an element of $\mathfrak{B}$ above each $r \in \mathfrak{R}(a)$.
$\mathfrak{Store}(\mathfrak{A}, \mathfrak{B})$	The same as $\mathfrak{Store}(\mathfrak{A}, \mathfrak{R}, \mathfrak{B}, \mathfrak{S})$, where we assume $\mathfrak{R}$ and $\mathfrak{S}$ to be given with $\mathfrak{A}$ and $\mathfrak{B}$.
$\mathfrak{B}^{*\mathfrak{A}}$	The same as $\mathfrak{Store}(\mathfrak{A}, \mathfrak{B})$.
$\mathfrak{Pres}(\mathfrak{A})$	This operation was introduced in §4.3, example 4.5. It stores $\{+\}$, for *present*, or $\varnothing$, for *absent*, on each $r \in \mathfrak{R}(a)$.

Table 1: Special cases of the Grothendieck Construction

- $n_{a^\wedge} = a(n_a)$ *and* $a^\wedge(n_{a^\wedge}) = n_a$
- $(.)^\wedge$ *is monotonic (with respect to* $\leq$*) and*
  - $\bullet$ *is monotonic in both arguments (with respect to* $\leq$*)*
- $(a \bullet b)^\wedge = b^\wedge \bullet a^\wedge$
- $a^{\wedge\wedge} = a$
- $a \bullet a^\wedge \leq \mathfrak{id}$ *and* $a^\wedge \bullet a \leq \mathfrak{id}$

Now we are ready to introduce $\mathcal{SSC}$, the category of simple stacking cells. We already know the objects of this category, the simple stacking cells, and also the merger has been discussed above. So the crucial thing to add is an appropriate notion of morphism. Here we are led by the following minimal requirement: we want to know *how levels travel* when simple stacking cells are merged, so we will need to keep track of the way that stacking cells get merged in between other stacking cells. This means that whenever a stacking cell a gets embedded in some context $b_1 \bullet - \bullet b_2$, then we want to have a morphism from a to the resulting stacking cell $b_1 \bullet a \bullet b_2$ that witnesses this embedding. Therefore we will at least need a morphism:

$$\varphi_{a,b_1,b_2} : a \to b_1 \bullet a \bullet b_2$$

for any choice of b_1 and b_2. We will denote such a morphism φ_{a,b_1,b_2} by $\langle b_1^\wedge, b_2 \rangle$ to limit the use of subscripts in our notation.[23] The morphisms $\langle b_1^\wedge, b_2 \rangle$ will be the only morphisms in the category of simple stacking cells $\mathcal{SSC}$.

Definition 5.2 $\mathcal{SSC}$, the m-category of simple stacking cells, has as objects the simple stacking cells and as morphisms $\varphi : a \to a'$ pairs $\langle b, b' \rangle$ such that $b^\wedge \bullet a \bullet b' = a'$. Composition and identities are as follows:

- ▹ for each a $\langle \mathfrak{id}, \mathfrak{id} \rangle : a \to a$ is the identity on a
- ▹ for $\langle b, b' \rangle : a \to a'$ and $\langle c, c' \rangle : a' \to a''$, $\langle b, b' \rangle \circ \langle c, c' \rangle : a \to a''$ is given by $\langle b \bullet c, b' \bullet c' \rangle$
- ▹ $in_1(a, a') = \langle \mathfrak{id}, a' \rangle$
- ▹ $in_2(a, a') = \langle a^\wedge, \mathfrak{id} \rangle$
- ▹ $\alpha(a, a', a'') = \langle \mathfrak{id}, \mathfrak{id} \rangle : (a \bullet a') \bullet a'' \to a \bullet (a' \bullet a'')$.

We leave it to the reader to check in detail that this does indeed define an m-category. As an example we consider the composition of morphisms: given $\langle b, b' \rangle : a \to a'$ and $\langle c, c' \rangle : a' \to a''$, we know that $a' = b^\wedge \bullet a \bullet b'$ and $a'' = c^\wedge \bullet a' \bullet c'$. Substitution now gives: $a'' = c^\wedge \bullet b^\wedge \bullet a \bullet b' \bullet c'$. By Fact 5.1 this can be written as $(b \bullet c)^\wedge \bullet a \bullet b' \bullet c'$ as required.

It may be useful to note that $\varphi : a \to a'$ is an isomorphism of $\mathfrak{SSC}$ iff $a = a'$ and $\varphi = \langle \mathfrak{id}, \mathfrak{id} \rangle$. So $\mathfrak{SSC}$ has very few isomorphisms.

In what follows we will use φ as a variable over morphisms in $\mathfrak{SSC}$.

5.2 How some levels travel Before we go on to extend the stacking cells with garbage levels, we take some time to check how the nongarbage levels travel when we merge two stacking cells. For each morphism $\langle b, b' \rangle : a \to a'$ we give a corresponding mapping $\mathfrak{L}(\langle b, b' \rangle) : \mathfrak{L}(a) \to \mathfrak{L}(a')$ as follows: for an arbitrary level $\langle n, n' \rangle \in \mathfrak{L}(a)$ we set

$$\langle n, n' \rangle [\mathfrak{L}(\langle b, b' \rangle)] = \langle b(n), b'(n') \rangle,$$

where we read $b(n) = \star$ if $n \notin dom(b)$ and $b'(n') = \star$ if $n' \notin dom(b')$. Of course it has to be checked that this does indeed define a mapping $\mathfrak{L}(a) \to \mathfrak{L}(a')$. This is a matter of case-checking.

Intuitively $\mathfrak{L}(\langle b, b' \rangle)$ has to describe what happens to the levels of $\mathfrak{L}(a)$ when a gets merged with $b^\wedge$ and b'. In this process many things can happen: for example, a push-level $\langle \star, n \rangle \in \mathfrak{L}(a)$ could simply become a push-level $\langle \star, b'(n) \rangle \in \mathfrak{L}(a \bullet b')$ and then stay a push-level $\langle \star, b'(n) \rangle \in \mathfrak{L}(b^\wedge \bullet a \bullet b')$. But it can also happen that a push level $\langle \star, n \rangle \in \mathfrak{L}(a)$ gets popped in $a \bullet b'$. Then it will be mapped to the garbage level, $\langle \star, \star \rangle$, of $a \bullet b'$ and then to the garbage level of $b^\wedge \bullet a \bullet b'$. For stem- and pop-levels we have to distinguish similar cases. It turns out that the formula $\langle b(n), b'(n') \rangle$ (with the notation convention as indicated) gives a concise presentation of all the cases.[24]

Note that all the levels that "disappear" in the merger $b^\wedge \bullet a \bullet b'$ are sent to the garbage level $\langle \star, \star \rangle$. If we had not added this garbage level, we would not know where to send such "disappearing levels" which would force us to work with partial functions at this point. But by the introduction of $\langle \star, \star \rangle$ we can keep all the functions total. Now it is easy to check the following.

Fact 5.3 *$\mathfrak{L}$ as defined above is a functor from $\mathfrak{SSC}$ to $\mathfrak{Set}$, the category of sets (with arbitrary mappings as morphisms).*

5.3 Traveling with garbage Now we come to the crucial step of adding more garbage (levels) to the picture. By adding a set of garbage levels we make a real stacking cell out of a simple stacking cell.

Above we have already smuggled in one garbage level, which enabled us to keep working with total mappings in the category of sets. The trick was to map all levels that were in danger of getting lost to the garbage level. This way no information needs to get lost, since it can all be sent to the garbage level. So *in a sense* information can be preserved, but as all the information ends up on the same level, we will get confused as to which information belongs together. In order to keep the information from different "disappearing levels" separate we need more than one garbage level.

So we start using pairs $\langle a, X\rangle$ where a is a simple stacking cell, as before, and X is a finite set of garbage levels. We simply call such pairs (not-so-simple) *stacking cells*.

Important examples of such stacking cells will be:

- $\mathfrak{push} = \langle\langle 0, 1\rangle, \varnothing\rangle$
- $\mathfrak{pop} = \langle\langle 1, 0\rangle, \varnothing\rangle$
- $\mathfrak{garb} = \langle \mathfrak{id}, \{\langle \Lambda, 0\rangle\}\rangle$.

These three are about the most basic stacking cells one can think of: $\mathfrak{push}$ consists of just one push level and no garbage. Similarly $\mathfrak{pop}$ consists of just one pop level without any garbage. $\mathfrak{garb}$ is the stacking cell that just has one garbage level and no real "structural" contribution. We have called the garbage level of $\mathfrak{garb}$ $\langle \Lambda, 0\rangle$, a pair consisting of the empty string Λ and the natural number 0. Later on it will become clear why it is convenient to assume that garbage levels have this kind of shape.

Whenever we merge two stacking cells $\langle a, X\rangle$ and $\langle a', X'\rangle$, the result is of the form $\langle a \bullet a', Y\rangle$. Here Y contains (i) the garbage levels X, (ii) the garbage levels X' and (iii) new garbage levels that are produced by the merger $a \bullet a'$. The new garbage levels are the levels that "disappear" in the merging process. This happens when a push level $\langle \star, n\rangle$ of a meets a pop level $\langle n, \star\rangle$ of a'. Each time this happens, we introduce a new garbage level and call it $\langle \Lambda, n\rangle$.

Since it is essential that we keep distinct garbage levels distinct, we will always have to take the *disjoint* union of garbage sets. There are, of course, several implementations of disjoint union around, each of which would do equally well for our purposes. But to keep things readable we prefer an implementation that does not introduce a lot of confusing brackets. To achieve this we assume that all garbage levels are pairs $\langle \sigma, x\rangle$, where σ is some string of 0's and 1's. We introduce the two shift operations Sh_0 and Sh_1 on sets of such elements. These operations are defined by:

$$Sh_i(X) = \{\langle i\sigma, x\rangle \mid \langle \sigma, x\rangle \in X\}.$$

The shift operations allow us to discriminate between elements of different origin without introducing lots of brackets. This is a clear advantage in the examples that follow later. Now we can implement disjoint union of garbage sets X and Y as follows:

$$X \oplus Y = Sh_0(X) \cup Sh_1(Y)$$

This gives us all the (notational) ingredients we need to introduce the garbage levels properly.

Definition 5.4 For each a, a' we define $G(a, a')$, the garbage introduced by merging a and a':

$$G(a, a') = \{\langle \Lambda, n\rangle |\ \langle \star, n\rangle \in \mathfrak{L}(a)\ \&\ \langle n, \star\rangle \in \mathfrak{L}(a')\ \&\ n \in \omega\}.$$

For a morphism $\varphi = \langle b_1^\wedge, b_2\rangle : a \to b_1 \bullet a \bullet b_2$ we define $G(\varphi)$, the garbage introduced by φ as:

$$G(\varphi) = G(b_1, a \bullet b_2)\ \cup\ Sh_1(G(a, b_2)).$$

Note that in defining $G(\varphi)$ we have—as it were—chosen a bracketing for $b_1 \bullet a \bullet b_2$. Here we see why we need to worry about the presence of suitable isomorphism: the existence of a "coherent" isomorphism α implies that such choices do not really matter in the end.[25]

Throughout this section it will be helpful to keep Figure 6 in mind. There we see three stacking cells $\langle a, X\rangle$, $\langle a', X'\rangle$, and $\langle a'', X''\rangle$. The sets X, X', and X'' are indicated by the little clouds below the simple stacking cells. Now when we merge $\langle a', X'\rangle$ and $\langle a'', X''\rangle$, for example, this will produce as new garbage $\{\langle \Lambda, 0\rangle\}$.

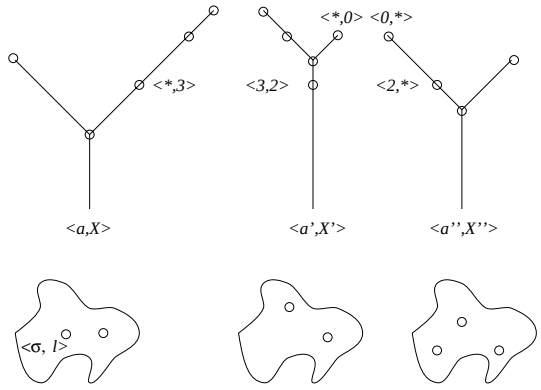


Figure 6: Merging with garbage

Now we can take as morphisms in the category of stacking cells, $\mathfrak{SC}$, pairs:

$$\langle \varphi,\ f\rangle :\ \langle a, X\rangle \to \langle a', X'\rangle$$

such that $\varphi : a \to a'$ is a morphism of $\mathfrak{SSC}$ and $f : G(\varphi) \oplus X \to X'$. The identity arrows simply are:

$$\begin{array}{lll} \langle id_a, \imath_{\varnothing\oplus X}\rangle : & \langle a, X\rangle \to \langle a, X\rangle & \\ \text{where } \imath_{\varnothing\oplus X} : & \varnothing \oplus X \to X & \text{is defined by } \imath_{\varnothing\oplus X}(\langle 1\sigma, x\rangle) = \langle \sigma, x\rangle, \end{array}$$

and composition of arrows is defined as:

$$\langle \varphi,\ f\rangle \circ \langle \psi, g\rangle = \langle \varphi \circ \psi, h\rangle,$$

where $h : G(\varphi \circ \psi) \oplus X \to X''$ is specified as follows.

We may assume that $\varphi = \langle b_1^\wedge, b_2\rangle : a \to b_1 \bullet a \bullet b_2$ and $\psi = \langle c_1^\wedge, c_2\rangle : b_1 \bullet a \bullet b_2 \to c_1 \bullet b_1 \bullet a \bullet b_2 \bullet c_2$. So $G(\varphi \circ \psi) = G(c_1 \bullet b_1, a \bullet b_2 \bullet c_2)\ \cup\ Sh_1(G(a, b_2 \bullet c_2))$. Now we distinguish the following cases:

- ▷ $h(\langle 1\sigma, x\rangle) = g(\langle 1\sigma', x'\rangle)$
  Here we write: $\langle \sigma', x'\rangle = f(\langle 1\sigma, x\rangle)$
  (in this case $\langle \sigma, x\rangle \in X$)
- ▷ $h(\langle 0, n\rangle) = g(\langle 1\sigma', x'\rangle)$
  in case $\langle 0, n\rangle \in dom(f)$
  Here we write: $\langle \sigma', x'\rangle = f(\langle 0, n\rangle)$
  (in this case $\langle \Lambda, n\rangle \in G(b_1, a \bullet b_2)$)
- ▷ $h(\langle 0, n\rangle) = g(\langle 0, n\rangle)$ in case $\langle 0, n\rangle \notin dom(f)$
  (in this case $\langle \Lambda, n\rangle \in G(c_1 \bullet b_1, a \bullet b_2 \bullet c_2) \backslash G(b_1, a \bullet b_2)$)
- ▷ $h(\langle 01, n\rangle) = g(\langle 1\sigma', x'\rangle)$
  in case $\langle 01, n\rangle \in dom(f)$
  Here we write: $\langle \sigma', x'\rangle = f(\langle 01, n\rangle)$
  (in this case $\langle \Lambda, n\rangle \in G(a, b_2)$)
- ▷ $h(\langle 01, n\rangle) = g(\langle 01, n\rangle)$ in case $\langle 01, n\rangle \notin dom(f)$
  (in this case $\langle \Lambda, n\rangle \in G(a, b_2 \bullet c_2) \backslash G(a, b_2)$)

Here the first case defines h on levels that initiate from X, and the second and third clauses consider garbage levels that are produced in the merger of $c_1 \bullet b_1$ and $a \bullet b_2 \bullet c_2$. The fourth and fifth clauses take care of the garbage that originates from merging a with $b_2 \bullet c_2$. Basically what we have to do is to keep in mind what could happen if we merge the five (!) simple stacking cells a, b_i, and c_i in two different ways: either we first merge a with the b_i and then later add the c_i. This is what happens if we do φ first and then ψ. Or else we first merge the b_i with the c_i and then merge the result with a. This is what happens if we compute $\varphi \circ \psi$ "right away." The definition is hard to read, and perhaps it is good advice to skip it and concentrate on our discussion of Figure 6 in §5.5, in which we see a case where *three* stacking cells are merged. However it can be checked that our definition does indeed produce a category.

Fact 5.5 *We have defined a category of stacking cells* $\mathfrak{SC}$.

It is important to note that the isomorphisms $\langle \varphi, f\rangle : \langle a, X\rangle \to \langle a', X'\rangle$ of this category are of the form: $\langle \varphi, f\rangle = \langle id_a, f\rangle$, where f is a bijection $f : Sh_1(X) \to X'$. This shows that isomorphism conditions ('coherence') in this category arise only at the level of the garbage sets: we have only to check that appropriate canonical bijections of garbage sets can be defined (cf. §5.5 for more details).

To extend this category into an m-category we have to specify the merger, the embeddings and the appropriate isomorphisms. We will not do this in full detail here: we just specify the merger of stacking cells and leave the other details to §5.5.

$$\langle a, X\rangle \bullet \langle a', X'\rangle = \langle a \bullet a', G(a, a') \cup (X \oplus X')\rangle$$

5.4 Levels again Now all that remains to be done is to extend the level functor $\mathfrak{L} : \mathfrak{SSC} \to \mathfrak{Set}$ to a level functor $\mathfrak{SC} \to \mathfrak{Set}$. We will use $\mathfrak{L}$ as notation for both functors. On objects we simply take:

$$\mathfrak{L}(\langle a, X\rangle) = X \oplus (\mathfrak{L}(a) \backslash \{\langle \star, \star\rangle\}).$$

So we collect the "real" levels of a and the garbage levels X of $\langle a, X\rangle$.

On morphisms $\langle \varphi, f\rangle : \langle a, X\rangle \to \langle a', X'\rangle$ we take:

$\mathfrak{L}(\langle \varphi, f\rangle) : \mathfrak{L}(\langle a, X\rangle) \to \mathfrak{L}(\langle a', X'\rangle)$

- $\langle 1n, m\rangle[\mathfrak{L}(\langle \varphi, f\rangle)] = \langle 1n', m'\rangle$
  in case $\langle n, m\rangle \in \mathfrak{L}(\langle a\rangle)$ and $\langle n, m\rangle[\mathfrak{L}(\langle \varphi\rangle)] = \langle n', m'\rangle \neq \langle \star, \star\rangle$
- $\langle 1n, m\rangle[\mathfrak{L}(\langle \varphi, f\rangle)] = \langle 0\sigma', k'\rangle$ in case $\langle n, m\rangle \in \mathfrak{L}(\langle a\rangle)$, $\langle n, m\rangle[\mathfrak{L}(\langle \varphi\rangle)] = \langle \star, \star\rangle$
  Now $\langle n, m\rangle$ gives rise to a tuple $\langle \sigma, k\rangle \in G(\varphi)$, where $\langle \sigma, k\rangle[f] = \langle \sigma', k'\rangle$
- $\langle 0\sigma, x\rangle[\mathfrak{L}(\langle \varphi, f\rangle)] = \langle 0\sigma', x'\rangle$
  in case $\langle \sigma, x\rangle \in X$ and $\langle \sigma, x\rangle[f] = \langle \sigma', x'\rangle$.

It can be checked that this does indeed make $\mathfrak{L}$ into a functor $\mathfrak{SC} \to \mathfrak{Set}$. In other words, we can check the following.

Fact 5.6 $\langle \varphi, f\rangle \circ \langle \psi, g\rangle[\mathfrak{L}] = \langle \varphi, f\rangle[\mathfrak{L}] \circ \langle \psi, g\rangle[\mathfrak{L}]$ *and* $id_{\langle a, X\rangle}[\mathfrak{L}] = id_{\langle a, X\rangle[\mathfrak{L}]}$.

Note that both $\langle a, X\rangle$ and $\langle \varphi, f\rangle$ are determined by their $\mathfrak{L}$-images. So we can regard $\mathfrak{SC}$ as a subcategory of $\mathfrak{Set}$.

5.5 How levels really travel Finally we look at our example again to see in some more detail how levels really travel when three stacking cells are merged in the category $\mathfrak{SC}$. We recall the following observation about $\mathfrak{SC}$.

Fact 5.7 *The isomorphisms* $\langle \varphi, f\rangle : \langle a, X\rangle \to \langle a', X'\rangle$ *of* $\mathfrak{SC}$ *are of the form:* $\langle \varphi, f\rangle = \langle id_a, f\rangle$, *where* f *is a bijection* $f : Sh_1(X) \to X'$.

So to check that suitable isomorphisms are present, we have only to look at the mappings of the garbage levels. This can be illustrated with our example Figure 6.

There are two different ways of merging these three stacking cells. We can either first merge the two leftmost stacking cells and then merge the result with $\langle a'', X''\rangle$, or we can first merge the two rightmost stacking cells and merge the result with $\langle a, X\rangle$. In the stacking cell component we will not notice any difference between the two approaches, since $a \bullet (a' \bullet a'') = (a \bullet a') \bullet a''$. But there will be a difference in terms of the garbage sets produced. To ensure the presence of suitable isomorphisms ("coherence"), we need a canonical bijection between the two garbage sets that the two different bracketings produce. (Recall that $\langle b, Y\rangle \bullet \langle b', Y'\rangle = \langle b \bullet b', G(b, b') \cup (Y \oplus Y')\rangle$.) Let's say that X_l is the garbage set obtained by left association of the brackets and X_r the set obtained by right association. We need a bijection $\alpha : X_l \to X_r$.

Here it helps to distinguish the following four cases:

1. $x \in X_l$ originates from one of the garbage sets X, X' or X''
2. $x \in X_l$ originates from a push level of a' that becomes garbage when a' and a'' are merged
3. $x \in X_l$ originates from a pop level of a' that becomes garbage when a and a' are merged
4. $x \in X_l$ originates from a stem level of a' that does not become garbage popped until the second merge step.

An example of Case 1 is given in the picture by the element $\langle \sigma, l\rangle \in X$. This will end up as $\langle 00\sigma, l\rangle \in X_l$, but as $\langle 0\sigma, l\rangle \in X_r$. So the bijection α will have to map $\langle 00\sigma, l\rangle$ to $\langle 0\sigma, l\rangle$. The general prescription for levels of type 1 is:

$$\begin{array}{lll} \langle 00\sigma, l\rangle & \rightsquigarrow_\alpha & \langle 0\sigma, l\rangle \\ \langle 01\sigma, l\rangle & \rightsquigarrow_\alpha & \langle 10\sigma, l\rangle \\ \langle 1\sigma, l\rangle & \rightsquigarrow_\alpha & \langle 11\sigma, l\rangle. \end{array}$$

An example of an element of type 2 is given in the picture by $\langle\star, 0\rangle$. This will end up in X_l as $\langle\Lambda, 0\rangle$, but in X_r it will appear as $\langle 1, 0\rangle$. So α will have to map $\langle\Lambda, n\rangle$ to $\langle 1, n\rangle$ in such a case.

By duality we need not consider 3 as a separate case.

The fourth case arises for the level $\langle 3, 2\rangle$ in the picture. This will end up as $\langle\Lambda, 2\rangle \in X_l$, but as $\langle\Lambda, 3\rangle \in X_r$. So α will have to map elements of the form $\langle\Lambda, n\rangle$ to $\langle\Lambda, a'(n)\rangle$ in these cases.

This gives a complete description of $\alpha : X_l \to X_r$. We will not go into the business of proving that this does indeed induce all the isomorphisms that are required.

6 Constructing meaningful monoids In this section we will put the machinery to work to construct some useful monoids. Remember that the $\mathfrak{Store}$-construction works only if the $\mathfrak{R}$-images of the morphisms of the context category are injective. Let's say that such categories have the injectivity property. We will start our constructions with $\mathfrak{Sc}$ and with categories of sets where the morphisms correspond to the subset ordering. The images of the morphisms of these categories are surely injective. It is not difficult to check that $\mathfrak{Cont}$, and $\mathfrak{Store}$ preserve the property. On the other hand $\mathfrak{Eq}$ does not preserve the injectivity property. Thus, we have to take care not to apply $\mathfrak{Store}$ after $\mathfrak{Eq}$!

6.1 Managing variables In this subsection we study a semantics for toy languages corresponding to the $\wedge\exists$-fragment of Predicate Logic. In particular, we indicate how to use this semantics both to simulate the $\wedge\exists$-fragment of Vermeulen's Sequence Semantics and of Vermeulen's Referent Systems (see Vermeulen [19],[18], and Hollenberg and Vermeulen [6]). We start by introducing some auxiliary objects and some useful notational conventions. Consider the m-category $\mathfrak{Sc}_{fin,+}$ of stacking cells where only finitely many levels are present. Remember that, via the standard embedding, we consider the objects of $\mathfrak{Sc}$ as occurring in $\mathfrak{Sc}_{fin,+}$. Note that for $a \in \mathfrak{Sc}$, we have that $\mathfrak{R}_{\mathfrak{Sc}}(a)$ is an infinite set, but that $\mathfrak{R}_{\mathfrak{Sc}_{fin,+}}(a) = \varnothing$. We define (suppressing the obvious subscripts): $\mathfrak{id}^+ := \langle \mathfrak{id}, \sigma\rangle$, where $\sigma(\langle 0, 0\rangle) := \{+\}$ and $\sigma(\langle n, n\rangle) := \varnothing$ for $n \neq 0$.

In the definitional format we use, the foregoing definition looks like this:

$$\mathfrak{id}^+ := \langle_{\mathfrak{s}}\ \mathfrak{id}, \{\langle 0, 0\rangle : \{+\}\}\ {}_{\mathfrak{s}}\rangle.$$

The salient points are these. First we indexed our brackets to indicate the relevant instance of the Grothendieck Construction. We use $\mathfrak{c}$ for $\mathfrak{Cont}$, $\mathfrak{e}$ for $\mathfrak{Eq}$, and $\mathfrak{s}$ for $\mathfrak{Store}$. Secondly we use an alternative notation for pairing in the description of the function σ. Finally we suppress both the constructions that add a unit of the relevant category and the function assignments of units: they are the default. Define further:

- $\mathfrak{push}^+ := \mathfrak{push} \bullet \mathfrak{id}^+$
- $\mathfrak{pop}^+ := \mathfrak{id}^+ \bullet \mathfrak{pop}$
- $\mathfrak{garb}^+ := \mathfrak{push} \bullet \mathfrak{id}^+ \bullet \mathfrak{pop}$

▷ $\mathfrak{block} := \mathfrak{pop} \bullet \mathfrak{push}$.

Note that $\mathfrak{garb}^+$ is $\mathfrak{id}$ plus one garbage level, where the garbage level is the only level present. We proceed by considering the category

$$\mathfrak{Varstack} := (\mathfrak{Sc}_{fin,+})^{*\mathfrak{Var}_{fin}}.$$

Here $\mathfrak{Var}_{fin}$ is the m-category of finite sets of variables with the subset ordering and union and as associated functor the obvious inclusion in $\mathfrak{Set}$.[26] For any $a \in \mathfrak{Sc}_{fin,+}$, define $a_x := \langle_{\mathfrak{s}} \{x\}, \{x : a\}\ _{\mathfrak{s}}\rangle$. Let $x^+ := (\mathfrak{id}^+)_x$. Finally we introduce the category of meanings for our fragment of Predicate Logic. Let D be any nonempty set. We take: $\mathfrak{Varman} := \mathfrak{Cont}(\mathfrak{Varstack}, D)$. This category is designed to handle *variable management*. Its elements are of the form $\langle a, F\rangle$, where a is an element of $\mathfrak{Varstack}$ and where F is a set of assignments from $\mathfrak{R}(a)$ to D. The (finitely many) referents of $\langle a, F\rangle$ are located above variables x in the outer context of a. They occur at levels of a stacking cell, which forms the inner context. The general form of the referents—in our standard way of coding—is $\langle x, \langle\langle u, v\rangle, +\rangle\rangle$ or, briefly, $\langle x, \langle u, v\rangle, +\rangle$, where x is a variable and u, v are in $\omega \cup \{*\}$, so that $\langle u, v\rangle$ is the level of a stacking cell.

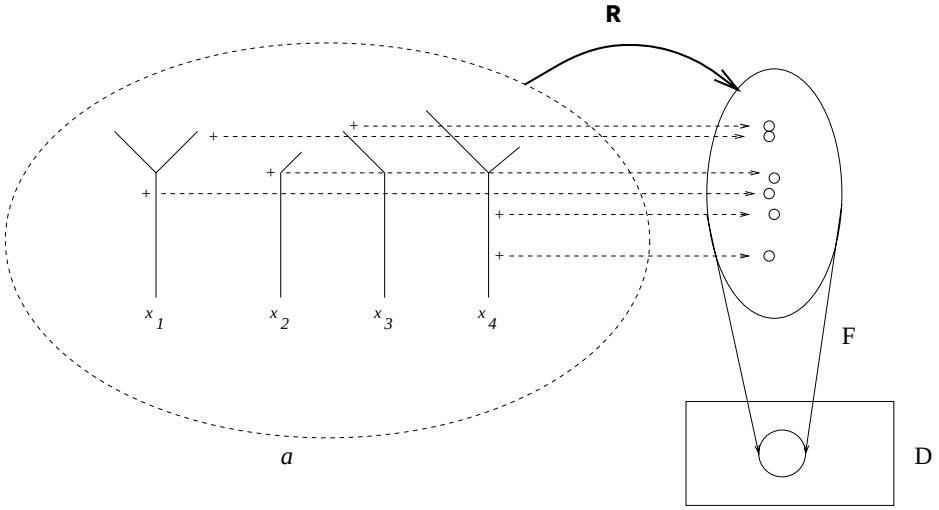


Figure 7: An object of $\mathfrak{Varman}$

We describe the language of the $\wedge\exists$-fragment corresponding to Sequence Semantics. To simplify inessentially, we consider only a language with a binary predicate symbol P and a unary predicate symbol Q. The atomic formulas are $[_x, {}_x]$, $P(x, y)$ and $Q(x)$, where x and y are variables. The language $\mathfrak{L}$ is the smallest set containing the atomic formulas, such that if $A \in \mathfrak{L}$ and $B \in \mathfrak{L}$, then $A.B \in \mathfrak{L}$. An example of a formulas is: $[_x.Q(x).[_y.P(x, y).{}_x].Q(y).{}_y]$. Let a standard first order model $\mathfrak{M} = \langle D, I\rangle$ for our language be given. $[_x$, the "left square bracket for x," is going to mean: *create a discourse environment in which an occurence of a file labeled x will be counted as* new. Counting as new, here, means that the file is not going to be identified with the current file—if there is one—labeled x. To put it in a different way: $[_x$ means *declare x*. Similarly ${}_x]$ will mean *end the discourse environment in which the current file (if any) for x is active*. We specify the *DRT*-style meanings corresponding to Sequence Semantics for our fragment. Remember our convention that

$\langle a, \langle b, c\rangle\rangle = \langle a, b, c\rangle$, etc. We put $r_x := \langle x, \langle 0, 0\rangle, +\rangle$. Thus r_x is the unique discourse referent of x^+.

- $[\![[_x]\!] := \mathfrak{push}_x$
- $[\![]_x]\!] := \mathfrak{pop}_x$
- $[\![P(x, y)]\!] := \langle_{\mathfrak{c}}\ x^+ \bullet y^+, \{\, f \in D^{\{r_x, r_y\}} \mid \langle f(r_x),\ f(r_y)\rangle \in I(P)\}\ _{\mathfrak{c}}\rangle$
- $[\![Q(x)]\!] := \langle_{\mathfrak{c}}\ x^+, \{\, f \in D^{\{r_x\}} \mid \langle f(r_x)\rangle \in I(Q)\}\ _{\mathfrak{c}}\rangle$
- $[\![A.B]\!] := [\![A]\!] \bullet [\![B]\!]$

Note that where $\mathfrak{push}_x$ represents *declare x*, $\mathfrak{push}_x^+$ would rather represent *declare and initialize x*. We will not give the precise correspondence of our semantics as presented here to Sequence Semantics as defined in [19].[27] Sequence Semantics as defined there cannot make the distiction between declaring and initializing, so in one respect our present semantics is more refined. We give two examples of interpretations. Let $g_x := \langle x, \langle \Lambda, 0\rangle, +\rangle$. Thus g_x is the unique discourse referent of $\mathfrak{garb}_x^+$. We have

$$[\![[_x.Q(x)._x]]\!] = \langle_{\mathfrak{c}}\ \mathfrak{garb}_x^+, \{\, f \in D^{\{g_x\}} \mid \langle f(g_x)\rangle \in I(Q)\}\ _{\mathfrak{c}}\rangle.$$

Note that g_x, the file "containing" possible witnesses for the truth of our formula is a hidden, noninteractive level for possible texts surrounding the formula. Thus we simulate the usual hiding of quantified formulas. Still in our setup the information stored in g_x is not really thrown away.[28] We turn to our second example. Let:

- $r_y := \langle y, \langle 0, 0\rangle, +\rangle$
- $s_x := \langle x, \langle *, 0\rangle, +\rangle$
- $t_x := \langle x, \langle *, 1\rangle, +\rangle$
- $(\mathfrak{push}^+)^2 := \mathfrak{push}^+ \bullet \mathfrak{push}^+ = \langle_{\mathfrak{s}}\ \langle\langle 0, 2\rangle, \varnothing\rangle, \{\langle *, 0\rangle{:}\{+\}, \langle *, 1\rangle{:}\{+\}\}\ _{\mathfrak{s}}\rangle$.

We have:

$$\begin{aligned}[\![[_x Q(x)[_x P(x, y)]]\!] = \\ &\langle_{\mathfrak{c}}\langle_{\mathfrak{s}}\{x, y\}, \{\ x{:}(\mathfrak{push}^+)^2,\ y{:}\mathfrak{id}^+\ \}_{\mathfrak{s}}\rangle, \\ &\{\, f \in D^{\{r_y, s_x, t_x\}} \mid \langle f(t_x)\rangle \in I(Q) \text{ and } \langle f(s_x),\ f(r_y)\rangle \in I(P)\}_{\mathfrak{c}}\rangle\end{aligned}$$

We close this subsection with a brief look at Referent Systems (see [18]). Referent Systems can be simulated by replacing $[_x$ and $_x]$ in our language by a single symmetric bracket $\|_x$. The semantics is as before, except that: $[\![\|_x]\!] := \mathfrak{block}_x$. So, when using $[\![\|_x]\!]$ to declare x, we simultaneously pop the current discourse environment (if any) in which x may have a value. Thus in the Referent Systems semantics, stacking never happens. We leave it to the reader to compute, e.g., $[\![\|_x.Q(x).\|_x]\!]$ and $[\![\|_x Q(x)\|_x P(x, y)]\!]$. In the last example the file-discourse referent corresponding to the first occurrence of $\|_x$ will be a garbage level. This in contrast to the second example for Sequence Semantics, where the file corresponding to the first occurrence of $[_x$ was t_x, a visible file.

6.2 Managing argument structure In this subsection we treat a version of the $\wedge\exists$-fragment of Predicate Logic that looks suspiciously like a fragment of English. We call our fragment: Semantics for Argument Management, or, briefly, *SAM*. Suppose we would like to represent the meaning of a natural language like anaphor, say $he_x/she_x/it_x$, in our version of Sequence Semantics as described above. It would

seem that x^+ is the perfect candidate for the job. It is a "free floating" variable, that signals the presence of an object labeled x. On closer inspection, however, this object would not really have a sensible role to play. How could x^+ ever interact in an interesting way with the meanings of a text? We have for example:

$$x^+ \bullet [\![Q(x)]\!] = [\![Q(x)]\!] \bullet x^+ = [\![Q(x)]\!].$$

One could say that (the interpretation of) the internal x of $Q(x)$ already does the work. Note also that $x^+ = [\![x = x]\!]$. Thus, naively, we seem to be close to providing anaphor-like meanings, but we just cannot reach our goal. This malaise is shared by theories like *DPL* and *DRT*. It is curious that, where these theories are advertised as providing a semantics for anaphoric reference, they fail to give a semantics that represents the *role* of anaphors, like $he_x/she_x/it_x$. The reason they cannot do it is simple. The specification language takes its format for handling arguments from Predicate Logic. *This format itself is already a solution of the problem that anaphors solve in natural language, namely to link up local and global information management.* Since in Predicate Logic the problem is already solved, one cannot well represent an alternative solution in the same language. In predicate language the arguments of a predicate are always specified in fixed places immediately following it. There can be no intervening material or changes of order (*salva significatione*). The meaning of, e.g., $P(x, y)$ is specified as one package. There is no further analysis of the way P, x, and y interact in terms of representable semantical operations. Thereby we miss the chance to tell the story about anaphors, about how they provide a link between the local and the global .... We will now give a semantics in which the way arguments are treated is more like the way it happens in natural language.

Let a finite set *AH* be given. The elements of this set are the argument handlers: *sub, ob, val, with, of* .... Let $\mathfrak{AH}$ be the m-category of subsets of *AH* with subset and union. Its associated functor is the inclusion in $\mathfrak{Set}$. We build up our target category in steps. First we make $\mathfrak{AH}^{*\mathfrak{S}\mathfrak{c}}$, in which sets of argument handlers are stored on levels of stacking cells. This category represents the local grammatical structure of sentences together with the arguments present at the various sentential levels. The discourse referents are *arguments on levels*. Define for $X \subseteq AH$:

$$\mathfrak{id}^X := \langle_s \mathfrak{id}, \{ \langle 0, 0\rangle{:}X \} _s\rangle.$$

Sometimes—as discussed in the introduction—the same object occurs on different levels. Thus we need the category $\mathfrak{Loc} := \mathfrak{Eq}(\mathfrak{AH}^{*\mathfrak{S}\mathfrak{c}})$. This category will be sufficient to handle local, sentential structure. To handle global, anaphoric structure we use $\mathfrak{Varman}$. Finally local and global have to be linked. To do this we work in the category: $\mathfrak{Sam} := \mathfrak{Varman} \| \mathfrak{Loc}$ (see Example 4.4).[29]

The language $\mathfrak{L}$ is defined as follows. The atomic formulas are:

Brac	(,)
Link	who, that, *sub*, *ob*, of, with, …
Phor	a_x, the_x, he_x, she_x, it_x, $Mary_x$, …(for $x \in VAR$)
CN	mother, father, child, horse, knife, …
Adj	angry, brown, …
Verb	cuts, sees, walks, is, …

Formulas are the smallest class containing the atomic formulas and closed under the rule: if A and B are formulas, then so is $A.B$. In other words, formulas are strings of atomic formulas with separating dots. To increase readability we will often omit the dots. Let an ordinary first order model $\mathfrak{M} = \langle D, I\rangle$, be given. I assigns relations to the elements of *CN*, *Adj*, and *Verb*. E.g., I(mother) could be a binary relation, representing *x is the mother of y*. I(cuts) could be a ternary relation, representing *x cuts y with z*. We could as well take I(cuts) to be a 5-ary relation, representing *x cuts y with z in place p at time t*, etc. The elements of *Brac*, *Phor*, and *Link* are treated as logical constants: their meanings are at most dependent on the domain of the model.[30]

Before we can proceed to specify the interpretations of the *CN*, the *Adj*, and the *Verb*, we have to introduce some notational conventions and simplifications. If one of the components of a pair from the Cartesian product underlying the $\|$-construction is a unit of the appropriate kind, then we will omit it and just exhibit the other component. This cannot lead to confusion, since the "inner context" of the first component always is a set of variables and the "inner context" of the second component always is a stacking cell. Moreover—in virtue of the specific categories going into this product—if a component is a unit, then its contribution to the ultimate set of referents is empty. Thus, *a fortiori*, $\mathfrak{Eq}$ restricted to this contribution to the referents can only be trivial. We will assume that an element $x \in X$, goes to $\langle 0, x\rangle \in X \oplus Y$. Similarly $y \in Y$ goes to $\langle 1, y\rangle$. We will often omit singleton parentheses. Finally $[\![r = s]\!] := \{\langle r, s\rangle\}^*$, where we take the $(.)^*$ in the appropriate set. $[\![r = s, t = u]\!]$, etc., is similarly defined. We give sample interpretations of the atoms of our language in Table 2.

(	$\mathfrak{push}$
)	$\mathfrak{pop}$
who	$\langle_{\mathfrak{e}} \langle_{\mathfrak{s}} \mathfrak{id}, \{\langle 0,0\rangle{:}val,\ \langle 2,2\rangle{:}val\}\ _{\mathfrak{s}}\rangle, [\![r_1 = r_2]\!]\ _{\mathfrak{e}}\rangle$ $r_1 := \langle\langle 0,0\rangle, val\rangle,\ r_2 := \langle\langle 2,2\rangle, val\rangle$
of	$\langle_{\mathfrak{e}} \langle_{\mathfrak{s}} \mathfrak{id}, \{\langle 0,0\rangle{:}val,\ \langle 1,1\rangle{:}of\}\ _{\mathfrak{s}}\rangle, [\![r_1 = r_3]\!]\ _{\mathfrak{e}}\rangle$ $r_3 := \langle\langle 1,1\rangle, of\rangle$
a_x	$\langle_{\mathfrak{e}} \langle \mathfrak{push}_x^+, \mathfrak{id}^{\{val\}}\rangle, [\![r_4 = r_5]\!]\ _{\mathfrak{e}}\rangle$ $r_4 := \langle 0, x, \langle *, 0\rangle, +\rangle,\ r_5 := \langle 1, \langle 0,0\rangle, val\rangle$
the_x	$\langle_{\mathfrak{e}} \langle \mathfrak{id}_x^+, \mathfrak{id}^{\{val\}}\rangle, [\![r_6 = r_5]\!]\ _{\mathfrak{e}}\rangle$ $r_6 := \langle 0, x, \langle 0,0\rangle, +\rangle$
mother	$\langle_{\mathfrak{c}} \mathfrak{id}^{\{0,of\}}, \{f{\in}D^{\{r_5,r_7\}} \mid \langle f(r_5),\ f(r_7)\rangle \in I(\text{mother})\}\ _{\mathfrak{c}}\rangle$ $r_7 := \langle 1, \langle 0,0\rangle, of\rangle$
knife	$\langle_{\mathfrak{c}} \mathfrak{id}^{\{val\}}, \{f{\in}D^{\{r_5\}} \mid \langle f(r_5)\rangle \in I(\text{knife})\}\ _{\mathfrak{c}}\rangle$
angry	$\langle_{\mathfrak{c}} \mathfrak{id}^{\{val\}}, \{f{\in}D^{\{r_5\}} \mid \langle f(r_5)\rangle \in I(\text{angry})\}\ _{\mathfrak{c}}\rangle$
cuts	$\langle_{\mathfrak{c}} \mathfrak{id}^{\{sub,ob,with\}}, \{f{\in}D^{\{r_8,r_9,r_{10}\}} \mid \langle f(r_8),\ f(r_9),\ f(r_{10})\rangle \in I(\text{cuts})\}\ _{\mathfrak{c}}\rangle$ $r_8 := \langle 1, \langle 0,0\rangle, sub\rangle,\ r_9 := \langle 1, \langle 0,0\rangle, ob\rangle,\ r_{10} := \langle 1, \langle 0,0\rangle, with\rangle$
is	$\langle_{\mathfrak{c}} \mathfrak{id}^{\{sub,ob\}}, \{f{\in}D^{\{r_8,r_9\}} \mid f(r_8) = f(r_9)\}\ _{\mathfrak{c}}\rangle$

Table 2: SAM's atomic interpretations

The recursive clause for interpretation is as expected: $[\![A.B]\!] := [\![A]\!] \bullet [\![B]\!]$.

We explain the definitions of Table 2. The brackets are easy: they push or pop levels of the local grammatical structure. Let's look at *mother*. [[mother]], has no

links to the global anaphorical machinery and does not change local syntactical structure. Thus the set of variables in the first component of the context is empty and the stacking cell in the second is the unit.[31] The argument handlers *val* and *of* are stored at the top level of the unit stacking cell in the second component of the context. These argument handlers give the *roles* of things standardly associated with motherhood. First there is the value, mother herself, tagged *val*. Then there are her children, tagged *of*.[32] E.g., the discourse referent $\langle 1, \langle 0, 0\rangle, \mathit{of}\rangle$ can be understood as follows. The first component 1 signals that we are in the second, the "local" component of the context. The $\langle 0, 0\rangle$ signals that we are at the top level of the unit. The *of* shows that we are looking at the argument *of*, stored at the top level.

Using the meanings introduced so far, we can already interpret a child reciting consecutively things she sees.

$$(.\text{horse}.).(.\text{mother}.).(.\text{dog}.).(.\text{cat}.)$$

The interpretation will have the effect of *there is a horse, there is a mother, there is a dog, there is a cat*. We do not analyze the *deixis* present in the child's words in our interpretation—our framework is too poor for that—but just the fact that she notes the existence of the consecutive items. The argument *of*, associated with *mother*, does not occur in the child's utterance. In the interpretation this has the effect of existentially quantifying out the argument. Thus, *mother* means *mother of someone*. If we had omitted the brackets separating the items, the effect would have been to identify the items. E.g. *(.mother.horse.)*, says that something is both a mother and a horse.

The interpretations of *knife* and *angry* do not bring anything new. In the interpretation of *cuts* and *is*, we have the special roles *sub* and *ob* of subject and object. We turn to the interpretations of the links. These interpretations serve to identify files across syntactical levels.[33] *that* is like *who*, and *sub, ob, with,* etc., are like *of*. We illustrate the way the linking works by means of an example.

Example 6.1 We assume that *runs* corresponds to a unary predicate. Let's consider the term *U:=(man ((who sub) runs))*. *who* occurs in a term *T:=(who sub)*. *T* occurs inside the sentence *S:=((who sub) runs)*. *S* occurs, in its turn, in *U*. To each of these components correspond "levels" of the stacking cell in the interpretation. These levels are introduced by the three left brackets and popped into garbage by the corresponding right brackets. Let's call these levels: *t*, corresponding to *T*, *s*, corresponding to *S*, and *u* corresponding to *U*. On *t*, a referent *val* is stored. This referent is the result of the dynamic fusing of the referent *val* stored on the upper level of $[\![\text{who}]\!]$ and the referent *val* stored on the upper level of $[\![sub]\!]$. On *s* we find the referent *sub*. It is the result of fusing the referent *sub* on the level $\langle 1, 1\rangle$ of $[\![sub]\!]$ with the referent *sub* of the top level of $[\![\text{man}]\!]$. In $[\![sub]\!]$ the referent *val* of the top level is "synchronically" identified with the referent *sub* one level below. So the referents *val* on *t* and *sub* on *s* are identified. On the level *u*, we find again a referent *val*. It is the result of fusing the referent *val* of the top level of $[\![\text{man}]\!]$, with the referent *val* of the level $\langle 2, 2\rangle$ of $[\![\text{who}]\!]$. Moreover by synchronic identification, the referent *val* on $\langle 2, 2\rangle$ in $[\![\text{who}]\!]$ is identified with the referent on $\langle 0, 0\rangle$ in $[\![\text{who}]\!]$. Hence *val* on *t*, *sub* on *s* and *val* on *u* are identified. We give the result of computing the meaning of *U* incrementally from left to right. Remember that different ways of computing the se-

mantics of our sentence will give different representations of the discourse referents. The existence of the isomorphisms α guarantees that this is harmless. Define:

- $r_1 := \langle 000, 0\rangle$, $r_2 := \langle 0, 0\rangle$, $r_3 := \langle \Lambda, 0\rangle$
- $X := \{r_1, r_2, r_3\}$
- $\sigma := \{r_1{:}\{val\},\ r_2{:}\{sub\},\ r_3 : \{val\}\}$
- $r_4 := \langle r_1, val\rangle$, $r_5 := \langle r_2, sub\rangle$, $r_6 := \langle r_3, val\rangle$
- $E := [\![r_4 = r_5,\ r_4 = r_6]\!]$
- $r_7 := \{\langle 1, \{r_4, r_5, r_6\}\rangle\}$
- $F := \{\, f \in D^{\{r_7\}} \mid f(r_7) \in I(\text{man}) \cap I(\text{runs})\}$.

We have: $[\![U]\!] = \langle_{c}\ \langle_{e}\ \langle_{s}\ \langle\langle 0, 0\rangle, X\rangle, \sigma\ _{s}\rangle, E\ _{e}\rangle, F\ _{c}\rangle$. By our conventions we suppressed the first component of the context. If instead we had computed $[\![U]\!]$ as [[(man ((who sub]] • [[) runs))]], the result would have been the same but for the fact that, for i=1,2,3, r_i would have been $\langle \Lambda, i-1\rangle$.

The phores operate like the links, only they link files or discourse referents of the global machinery to files or discourse referents of the local machinery. On the global side the machinery is simply Sequence Semantics. The meaning of he_x, she_x, and it_x is taken to be the same as the meaning of the_x. Our choice of treating *the* as an anaphor is not undisputed. There are plenty of examples that seem to undermine this theory.[34] We will not go into that discussion here. How do we treat names? In fact our semantics provides various options. The one we prefer is viewing names as "frozen anaphors." So the meaning of a name is like the meaning of *he*. Our present framework is too poor to model the *frozenness* of names fully. For that we would need the notion of state, which is not treated in this paper. We can, however, make one simple adaptation to get part of the desired effect. We set aside some labels for variables to function as subscripts of names. We exclude these labels from occurring as subscripts of *a* and *the*. They can occur as subscripts of *he, she*, and *it*.[35]

Claim We submit that we are the first to describe correctly *in the semantics* the role of the phores as links. This does not mean, however, that we claim that our solution is a fully correct representation of the meanings of phores in natural language. First, it seems that Sequence Semantics allows lots of structure one never meets in natural language. For example, nothing seems to correspond to the stacking under *x* in: ((*sub* a_x dog) sees (*ob* a_x cat)). Secondly, it could well be that fusing in natural language happens more by higher order inference than by label. These defects are a problem for all approaches we know.

We have described the various meanings provided by our semantics, and we have touched briefly on some further topics such as the problem of definite descriptions and the proper treatment of names. It is time to have a look at some sample sentences in our language.

(a) ((*sub* Mary_x) cuts (the_y *ob* bread) (with a_z sharp knife))
(b) ((with a_z knife sharp) cuts (*ob* the_y bread) (Mary_x *sub*))
(c) ((*sub* Mary_x) cuts (the_y *ob* bread))
(d) ((*sub* Mary_x) cuts (with a_z sharp knife))
(e) ((a_x woman *sub*) sees (a_y horse *ob*)) ((she_x *sub*) beats (it_y *ob*))

(f) ((the$_x$ *sub* man ((who *sub*) sees (the$_y$ *ob* brown horse)) cuts (the$_z$ *ob* bread))

(g) ((Mary$_x$ *sub*) ((she$_x$ *sub*) is (*ob* angry)) ((she$_x$ *sub*) is (hungry *ob*)) cuts (the$_y$ bread *ob*))

Example (a) illustrates one advantage of our approach: the interpretation of such a sentence can proceed in the order in which it is given. (a) and (b) are equivalent in meaning, since we may interchange, *salva significatione* (*modulo* some specifiable isomorphism), the order of items as long as no bracket (local or global) intervenes to which these items are sensitive. Leaving out argument places as in (c) or (d) has the effect of having a hidden existentially quantified argument. Thus (c) means something like: *Mary cuts the bread with something.* Note that we can as well suppress the subject, which is unusual in English (but it is common in Latin). (e) works like the usual *DPL/DRT*-example of anaphoric reference. E.g., in the interpretation, the *horse* will be fused with *it*. Example (g) illustrates the fact that in our approach sentences can be interrupted for other sentences. These will be "laid over" the interrupted sentence.

We end this section with a brief remark on one possible extension of the fragment and its problems.[36] We could try to add a semantics for the Dutch reflexive *zichzelf* (or German *sichselbst*). One possible interpretation of *zichzelf* is to make it just a link between subject and object.[37] So:

$$[\![\text{zichzelf}]\!] := \langle_{\mathfrak{s}}\, \mathfrak{id}, \{\langle 0,0\rangle{:}\{sub, ob\}\}\, {}_{\mathfrak{s}}\rangle.$$

Under this analysis, we would represent *John snijdt zichzelf* (English: *John cuts himself*) as: ((John$_x$ *sub*) snijdt zichzelf). Alternatively, we could make *zichzelf* a term. Thus, we could take: $[\![\text{zichzelf}]\!] := [\![sub]\!]$. Under this analysis, we would represent our sample sentence by: ((John$_x$ *sub*) snijdt (zichzelf *ob*)). However, consider such sentences as:

John ziet een foto van zichzelf

John ziet een foto van een foto van zichzelf

(meaning roughly: *John sees a picture of himself* and *John sees a picture of a picture of himself* respectively). Here it seems that the meaning of *zichzelf* is able to search for the appropriate level for linking. Our approach in its present form allows us only to build links across a specified fixed number of levels. One possible way out is as follows. We change our semantics in such a way that we put sentences always between lazy brackets (see §2.4). Thus we would rewrite the above example (f) as:

▷ [the$_x$ *sub* man [who *sub*) sees (the$_y$ *ob* brown horse] cuts (the$_z$ *ob* bread]

Obviously, we should make adaptations, e.g., for the meaning of *who*. (The second term-label *val* should be stored on $\langle\omega,\omega\rangle$ instead of on $\langle 2,2\rangle$.) We treat *zichzelf* as a term and store *sub* in its interpretation on all levels $\langle i,i\rangle$ for $i<\omega$. Moreover by $\mathfrak{Eq}$, we identify these *subs*. Obviously, our operation will cause spurious *subs* to occur on various term levels, but these can do no harm, since they will not fuse with other labels on the term levels. Consider the sentence:

▷ [John$_x$ *sub*) ziet (een$_y$ foto *ob* (van een$_z$ foto (van zichzelf]

We obtain the effect that we search on each lower level for something to fuse with *sub* until we reach the sentence level. Hence we will correctly identify *John* with *zichzelf*. Our "theory" simply predicts that *zichzelf* is always fused with the subject of the nearest sentence level below. E.g. *zichzelf* will be fused with *de vrouw* in: *De man, aan wie de vrouw een foto van zichzelf gaf, glimlachte* (Here we see a major difference with English *himself*: *The man, who gave the woman a picture of himself, smiled.* In this sentence, we certainly do *not* want *the woman* and *himself* to be identified).

The main point of our elaboration is the mechanism of *searching for the appropriate level* can be implemented in our framework.

7 Concluding remarks This paper describes techniques to construct meaning-objects for monoidal processing. One starts with simple objects—like finite sets or stacking cells. By iterating the Grothendieck Construction more elaborate objects are constructed. The great advantage of the Grothendieck construction is that the appropriate monoidal behavior is automatically preserved. We introduced stacking cells as an interpretation of bracket structures. Subsequently, we outlined the interpretation of a fragment using the construction methods of the paper. This fragment incorporates a linking mechanism to describe what anaphors do. We claim that only some such mechanism can pretend to truly constitute a semantics of anaphoric reference. If this claim is correct, we have given an example that monoidal semantics can provide *faithful modeling* of meanings.

The purpose of our paper is more to point in the right direction, than to establish a definitive, rigid framework once and for all. More questions are evoked than answered. We distinguish three kinds of extension of our work: (i) mathematical improvement of the framework as it stands, (ii) extending the fragment developed here within the boundaries of the present framework, and (iii) extending the framework with essentially new elements both to increase expressive power and to incorporate some further philosophical ideas. We briefly sketch some ideas for extensions of the three different kinds.

We mention some directions of local improvement of the framework as it stands. First, one would like to incorporate a smooth construction of the "file-set functor" $\mathfrak{R}$. Second, it would be good to be able to construct stacking cells from even simpler objects. One of the authors (Visser) is currently working on a proposal to represent stacking cells as multisets of morphisms of an appropriate category.

We mention some ideas for extending the fragment that seem to be in the scope of the methods developed so far. There are many interesting phenomena that we would like to include. For example, we would like to have a way of working with an expandable number of argument places (cf. *the horse of Sir John* in §6.2). Currently we are working on a uniform treatment of the semantics of *and* in sentences such as *John eats the bread with a fork and the pudding with a spoon* and *John hates and Mary loves Marc*, etc. Another extension of the fragment could consist of a treatment of error messages to ensure the propagation of information about local errors in the syntax, e.g., in cases where term levels are erroneously fused with sentence levels.

Finally, we discuss some possible extensions of the framework. The present paper *is* a study in dynamics. Therefore we would like to include into our framework more of the salient ingredients occurring in the literature on dynamics. First and fore-

most, we can extend our framework with a good notion of state and state transition, so that a formulation of our semantics in update style (cf. Veltman [17]) and in the relational format (cf. [2]) becomes available. There is an elegant way to do this, which will be presented elsewhere. Another important ingredient we intend to include is a rational reconstruction of dynamic implication. It is our prejudice that (dynamic) implication should be an adjunction in an appropriate category of partial information states. We suspect that finding this category will become possible once a proper notion of presuppositional ordering (in which presuppositions are counted as *negative* information) has been added to the framework (cf. [21] for preliminary investigations).

Clearly, our work is just an initial step towards the grand aim of a theory of monoidal processing. But we think that it exhibits very well the sort of thing one has to do and the sort of question one has to answer. We submit that we have shown—by means of examples—that such a theory can provide a powerful setting in which the phenomena of discourse processing can be fruitfully discussed.

Appendix Semantics for binary notations In this appendix we show how to use the Grothendieck Construction to give a semantics for binary notations.[38] Binary notations are the usual designations of numbers in binary like "101," which stands for *five*. Our problem is that we want to assign meanings to these notations that make concatenation of notations a meaningful operation. So consider a language containing binary notations plus a *symbol* $*$ for concatenation. Let's first consider the option of interpreting a notation as the number it designates. E.g., $[\![101]\!] = 5$. The problem is that we would have to put, e.g., $[\![01]\!] = [\![1]\!] = 1$. But,

$$[\![11 * 01]\!] = [\![1101]\!] = 13 \neq 7 = [\![111]\!] = [\![11 * 1]\!].$$

So we cannot interpret $*$ compositionally under this semantics. At the other extreme we could interpret binary strings autologically: *as themselves*. This would surely lead to a compositional semantics, but we would lose the central idea that these notations are supposed to stand for numbers. Our solution is to interpret notations as pairs $\langle m, n\rangle$, where m is the length of the notation and where n is its customary value. We show how this semantics can be assembled using the Grothendieck Construction.

We start by specifying our m-category of contexts. This is the m-category $\mathfrak{Nat}_{loc}$ of located unary strings or located tally numbers.

- ▹ The objects are natural numbers $\{0, 1, 2, \ldots\}$. These can be viewed as unary strings or tally numbers.
- ▹ The morphisms are triples $\langle m, n, k\rangle$, with $m + n \leq k$. A morphism $\langle m, n, k\rangle$ tells us that m is embedded in k at location n. Here n is the number of 1s in k occurring after m. Like this:

$$\underbrace{1\ldots1\overbrace{1\ldots1}^{m}\overbrace{1\ldots1}^{n}}_{k}.$$

- ▹ $dom(\langle m, n, k\rangle) = m$, $cod(\langle m, n, k\rangle) = k$, $id_m = \langle m, 0, m\rangle$

▷ $\langle m, n, k\rangle \circ \langle k, p, q\rangle = \langle m, n+p, q\rangle$

$$\underbrace{1\ldots1\underbrace{1\ldots1}_{m}\underbrace{1\ldots1}_{n}\underbrace{1\ldots1}_{p}}_{k}\Big\}_{q}$$

▷ $m \bullet n = m+n$, $in_1(m,n) = \langle m, n, m+n\rangle$, $in_2 = \langle n, 0, m+n\rangle$

$$\underbrace{\underbrace{1\ldots1}_{m}\underbrace{1\ldots1}_{n}}_{m+n}.$$

▷ α is just the appropriate identity, since $+$ gives us a standard monoid.

▷ $\mathfrak{id} = 0$

Take $\Theta(n) = \mathfrak{Nat}$ and $p[\Theta(\langle n, m, k\rangle)] = p \cdot 2^m$. It is easy to see that this defines an m-functor. We get:

$$\begin{array}{rcl}\langle i, n\rangle \bullet \langle j, m\rangle & = & \langle i+j, n[\Theta(in_1(i,\,j))] + m[\Theta(in_2(i,\,j))]\rangle \\ & = & \langle i+j, n[\Theta(\langle i,\,j,\,i+j\rangle)] + m[\Theta(\langle j, 0, i+j\rangle)]\rangle \\ & = & \langle i+j, n\cdot 2^j + m\rangle.\end{array}$$

A good alternative way of representing the objects we constructed is as follows. Consider the pair $\langle i, n\rangle$. Write n in binary, and precede it by infinitely many 0s. Put a pointer above the place followed by i digits. We represent, for example, $\langle 3, 41\rangle$ by: $\ldots 000\underset{\downarrow}{1}01001$. Note that, e.g., $\langle 4, 9\rangle \bullet \langle 3, 41\rangle$ can be computed by:

$$\begin{array}{cccccccccc} & & & & & & \downarrow & & & \\ & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ & & \downarrow & & & & & & & \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & & & \\ \hline & & \downarrow & & & & & & & \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{array}$$

Thus the context of the second component of the merge has the effect of a shift. We interpret binary notations by the pair of their length and their value. The second component is the classical content, the first a dynamic context that causes a shift in interaction. Evidently, $[\![\sigma * \tau]\!] = [\![\sigma]\!] \bullet [\![\tau]\!]$. So our semantics produces the desired effect. Note that $\mathfrak{id} = \langle 0, 0\rangle \neq \langle 1, 0\rangle = [\![0]\!]$.

Acknowledgments Both during the process of theory formation and that of document preparation we benefited from the help and influence of colleagues and friends. We thank Patrick Blackburn, Jan van Eijck, Tim Fernando, Claire Gardent, Marco Hollenberg, Marcus Kracht, Emiel Krahmer, and Henk Zeevat for enlightening discussions. We found intellectual inspiration in the work of Johan van Benthem, Jeroen Groenendijk, Remco Scha, Martin Stokhof, Frank Veltman, and Henk Zeevat. We were helped both with matters of notation and matters of LaTeX by Jan Friso Groote, Karst Koymans, Jaco van de Pol, Alex Sellink,

Jan Springintveld, and Freek Wiedijk. We thank Maarten de Rijke and Patrick Blackburn for encouraging us to write the paper and for their comments on the penultimate version.

NOTES

1. Our semantics is also closely related to Kamp's *DRT* (see [8],[9]) and to Seuren's Discourse Semantics (see [16]).
2. See also Hollenberg and Vermeulen [6].
3. If we say, *suppose* …, we introduce an imagined world. Thus, supposing opens a stretch of discourse which is interpreted with respect to this new imagined world. The idea that *suppose* is "pushing into fantasy" comes from a suggestive discussion by Hofstadter [5], p. 128. It was studied by Zeinstra [23].
4. Both relations and update functions can be associated to our meanings in a mathematically elegant way. We will substantiate these claims elsewhere.
5. Stacking cells were introduced by Visser [21],[20].
6. Note that we write . here between each of the elements of the strings. This is our official notation, but, as usual, we will allow ourselves to omit . if no confusion can arise.
7. There is a strong analogy with the construction of the integers as pairs of natural numbers!
8. Here $\dot{-}$ stands for *cut-off* substraction: $x \dot{-} y = x - y$ if $y \leq x$ and $x \dot{-} y = 0$ else.
9. The reader may wish to verify that the monoid of simple stacking cells is in fact the free monoid over two generators) and (with equation $() = 1$. We will prefer to work with the more concrete representations in this paper.
10. We always read $f \circ g$ as *first f and then g* (cf. p. 335).
11. By the symmetry of $\wedge$ the Boolean implication also is a *right* implication. Note however that in a Boolean algebra L1 does not hold.
12. Recall that addition of ordinals is not in general commutative.
13. We were led to this view on things by remarks of Henk Zeevat on "discourse popping."
14. In fact this gives us the free monoid over four generators), (,], and [satisfying the additional equations: $[) = [(= [, (] =)] =]$, and $[] = () = 1$.
15. In fact 'that' will be interpreted as a link between levels (cf. the previous section), so strictly speaking we cannot say that 'that' is located on one particular level.
16. The distinction between large categories, in which the objects or the arrows do not form a set, and small ones, in which objects and arrows do form sets, plays no role in this paper. Categories like $\mathfrak{Set}$ could, in our applications, always be replaced by appropriate small categories.
17. It could be argued that to assign to the morphisms this double task is in some sense impure. The point is strengthened by the fact that our monoidal operation is a bifunctor with respect to the categories restricted to isomorphisms, but not with respect to the full categories. The reason that we have the two roles in one and the same category, is pragmatic: things seem to work out well this way.
18. Thus $in_1(a, \mathfrak{id})$ and $in_2(\mathfrak{id}, a)$ have the role of ρ_a^{-1}, respectively λ_a^{-1} of [10], p. 158.

19. Our conditions imply that an m-category is a monoidal category, when we restrict it to isomorphisms and, thus, that it is *coherent*. See [10], pp. 157–166, for further explanation.

20. The Grothendieck Construction is, thus, reminiscent of the "central dogma of molecular genetics," viz., that information can flow from nucleic acids to proteins, but cannot flow from protein to nucleic acid.

21. In a more definitive treatment we should expect to derive the new $\mathfrak{R}$ systematically. In this paper we will content ourselves by introducing them *ad hoc*.

22. Note that the operation is *not* exponentiation, even if there are some similarities.

23. This notation is very convenient, but please keep in mind that there is a difference between $\langle b_1^{\wedge}, b_2\rangle : a \to a'$ and $\langle b_1^{\wedge}, b_2\rangle : c \to c'$.

24. Note that the convention ensures that $b(\star)$ and $b'(\star)$ are read as $\star$.

25. We have chosen to take the garbage produced by the merger as the basic notion and to define the garbage introduced by a morphism as a derived notion. But this is merely a matter of choice: it can be checked that $G(\varphi)$ consists of all the levels that are sent to $\langle\star,\star\rangle$ by $\mathfrak{L}(\varphi)$. To be precise: there is a bijection between $G(\varphi)$ and $\langle\star,\star\rangle[\mathfrak{L}(\varphi)]^{-1}\backslash\{\langle\star,\star\rangle\}$. This suggests an alternative way of introducing garbage formally, where the garbage produced by a morphism is the basic notion and the garbage introduced by the merger is defined in terms of it.

26. There is a slight inelegance in using $\mathfrak{Var}_{fin}$ in the definition of $\mathfrak{Varstack}$. It is that a variable v can be "absent" in two ways in $\langle V, \sigma\rangle$, viz., either if $v \notin V$ or if $v \in V$, but if $\sigma(v) = \mathfrak{id}$. We can get around this defect as follows. As contexts, start with the m-category having the set of all variables as single element, with subset and union and with the usual inclusion in $\mathfrak{Set}$ as associated functor. Then apply the $\mathfrak{Store}_{fin}$ construction with the set of variables as single context and the elements of $\mathfrak{Sc}_{fin,+}$ as stored objects.

27. We hope to elaborate on this elsewhere. In fact, one can show that under quite general conditions the $\mathfrak{Cont}$ construction yields a semantics from which a relational *DPL*-style semantics can be derived in a natural way. To do this we need the additional notion of *state*, which is developed in [21].

28. A *garbage disposal* construction would be a useful addition to our framework.

29. In fact, it suffices to apply the construction $\mathfrak{Eq}$ only once. We prefer the current set-up, because it allows us to consider the local identifications in isolation of the global anaphoric machinery.

30. We will also treat *is* as a logical constant.

31. *Sam*-meanings of which all embedded simple stacking cells are the unit (i.e., the embedded stacking cells are unit plus garbage) are called *conditions*. *Sam*-expressions, whose meanings are conditions are likewise called conditions. Thus *mother* is a condition, but *(* is not. When the embedded simple stacking cell of a first component is the unit, we speak of a *global condition* and when the embedded simple stacking cell of a second component is the unit of a *local condition*. Thus, sentences and terms are (or stand for) local conditions.

32. A disadvantage of our framework in its present form is that we have to choose the arguments associated with a given word in advance. E.g., not every horse has an owner, but to make sense of *the horse of Sir John*, we would have to add an argument *of* to the interpretation of *horse*. But, adding the argument licenses the inference of the existence of an owner, whenever we speak of a horse. We feel confident that it will be possible to manufacture more flexible versions of our framework lacking this defect.

33. The meaning of *who*, e.g., is a condition according to the definition of Note 7. Note that this usage does not quite correspond to the usual idea of a condition as a test.

34. For example: *the winner will get one thousand guilders*.

35. Note that the meaning of *((sub Hesperus$_x$) is (ob Hesperus$_x$))* is different from the meaning of *((sub Hesperus$_x$) is (ob Phosphorus$_y$))*, since fusion of discourse referents is different from contentual identity.

36. We were made aware of the problems concerning *zichzelf* by Claire Gardent.

37. Note that we assume here that *zichzelf* always fuses with *sub*s. It was pointed out to us that this is not anadequate assumption about, for example, English *himself*, as is clear from: *Mary gave John a picture of himself*, where *himself* is the indirect object of the nearest sentence level. However, it seems that what we give is a good approximation of the meaning of Dutch *zichzelf* or German *sichselbst*.

38. The appendix is our answer to a question posed by Theo Janssen.

REFERENCES

[1] Barr, M., and C. Wells, *Category Theory for Computing Science*, Prentice Hall, New York, 1989. Zbl 0714.18001 MR 92g:18001 3.1, 4

[2] Groenendijk, J., and M. Stokhof, "Dynamic predicate logic," *Linguistics and Philosophy*, vol. 14 (1991), pp. 39–100. 1.1, 1.2, 7

[3] Heim, I., *The Semantics of Definite and Indefinite Noun Phrases*, Ph.D. thesis, University of Massachusetts, Amherst, 1982. 1.1

[4] Heim, I., "File change semantics and the familiarity theory of definiteness," pp. 164–189 in *Meaning, Use and Interpretation of Language*, edited by R. Bäuerle, C. Schwarze, and A. von Stechow, De Gruyter, Berlin, 1983. 1.1, 3.1

[5] Hofstadter, D. R., *Gödel, Esher, Bach: An Eternal Golden Braid*, Basic Books, New York, 1979. 7

[6] Hollenberg, M., and C. F. M. Vermeulen, "Counting variables in a dynamic setting," Logic Group Preprint Series LGPS-125, Department of Philosophy, Utrecht University, December 1994. 6.1, 7

[7] Jacobs, B. P. F., *Categorical Type Theory*, Ph.D. thesis, Catholic University Nijmegen, 1991. 4

[8] Kamp, H., "A theory of truth and semantic representation," pp. 277–322 in *Formal Methods in the Study of Language*, edited by J. Groenendijk et al., Mathematisch Centrum, Amsterdam, 1981. MR 84h:03069 7

[9] Kamp, H., and U. Reyle, *From Discourse to Logic*, vol. I, II, Kluwer, Dordrecht, 1993. 1.2, 7

[10] MacLane, S., *Categories for the Working Mathematician*, Number 5 in Graduate Texts in Mathematics, Springer, Berlin, 1971. Zbl 0705.18001 MR 50:7275 2.4, 3.1, 7, 7

[11] Manes, E., and M. Arbib, *Arrows, Structures and Functors: the Categorical Imperative*, Academic Press, New York, 1975. Zbl 0374.18001 MR 51:638 3.1

[12] Milward, D., "Dynamics, dependency grammar and incremental interpretation," pp. 1095–1099 in *Proceedings of COLING 92*, 1992. 1.1

[13] Milward, D., "Dynamic dependency grammar," *Linguistics and Philosophy*, vol. 17 (1994), pp. 561–606. 1.1

[14] Moortgat, M. J., and R. Oehrle, "Adjacency, dependency and order," pp. 447–466 in *Proceedings of 9th Amsterdam Colloquium*, ILLC Publications, 1994. 2.4

[15] Pratt, V., "Action logic and pure induction," pp. 97–120 in *Logics in AI — European Workshop JELIA '90*, edited by J. van Eijck, Springer, Berlin, 1991. Zbl 0814.03024 MR 92d:03016 2.4

[16] Seuren, P., *Discourse Semantics*, Blackwell, Oxford, 1985. 7

[17] Veltman, F., "Defaults in update semantics," pp. 28–65 in *Conditionals, Defaults and Belief Revision*, edited by H. Kamp, Dyana Deliverable R2.5A, Edinburgh, 1991. 7

[18] Vermeulen, C. F. M., "Merging without mystery, variables in dynamic semantics," forthcoming in *Journal of Philosophical Logic*. Zbl 0825.03016 MR 96b:03044 6.1, 6.1

[19] Vermeulen, C. F. M., "Sequence semantics for dynamic predicate logic," *Journal of Logic, Language and Information*, vol. 2 (1993), pp. 217–254. Zbl 0802.03024 MR 95e:03097 1.1, 6.1, 6.1

[20] Visser, A., "Lazy and quarrelsome brackets," Logic Group Preprint Series 82, Department of Philosophy, Utrecht University, November, 1992. 7

[21] Visser, A., "Actions under presuppositions," pp. 196–233 in *Logic and Information Flow*, edited by J. van Eijck and A. Visser, MIT Press, Cambridge, 1994. MR 1295068 7, 7, 7

[22] Zeevat, H., "A compositional approach to DRT," *Linguistics and Philosophy*, vol. 12 (1991), pp. 95–131. 1.2

[23] Zeinstra, L., "Reasoning as discourse," Master's thesis, Philosophy Department, Utrecht University, 1990. 7

Department of Philosophy
Utrecht University
P.O. Box 80.126
3508 TC Utrecht
The Netherlands
email: Albert.Visser@phil.ruu.nl
email: Kees.Vermeulen@phil.ruu.nl